Imperial College London
Department of Computing

Practical Deep Learning

Hao Dong

Submitted for PhD Thesis

October, 2018

Abstract

Deep learning is experiencing a revolution with tremendous progress because of the availability of large datasets and computing resources. The development of deeper and larger neural network models has made significant progress recently in boosting the accuracy of many applications, such as image classification, image captioning, object detection, and language translation. However, despite the opportunities they offer, existing deep learning approaches are impractical for many applications due to the following challenges. Many applications exist with only limited amounts of annotated training data, or the collected labelled training data is too expensive. Such scenarios impose significant drawbacks for deep learning methods, which are not designed for limited data and suffer from performance decay. Especially for generative tasks, because the data for many generative tasks is difficult to obtain from the real world and the results they generate are difficult to control. As deep learning algorithms become more complicated increasing the workload for researchers to train neural network models and manage the life-cycle deep learning workflows, including the model, dataset, and training pipeline, the demand for efficient deep learning development is rising.

Practical deep learning should achieve adequate performance from the limited training data as well as be based on efficient deep learning development processes. In this thesis, we propose several novel methods to improve the practicability of deep generative models and development processes, leading to four contributions. First, we improve the visual quality of synthesising images conditioned on text descriptions without requiring more manual labelled data, which provides controllable generated results using object attribute information from text descriptions. Second, we achieve unsupervised image-to-image translation that synthesises images conditioned on input images without requiring paired images to supervise the training, which provides controllable generated results using semantic visual information from input images. Third, we deliver semantic image synthesis that synthesises images conditioned on both image and text descriptions without requiring ground truth images to supervise the training, which provides controllable generated results using both semantic visual and object attribute information. Fourth, we develop a research-oriented deep learning library called TensorLayer to reduce the workload of researchers for defining models, implementing new layers, and managing the deep learning workflow comprised of the dataset, model, and training pipeline. In 2017, this library has won the best open source software award issued by ACM Multimedia (MM).

Copyright Declaration

Declaration of Originality

This thesis presents my work in the Department of Computing at Imperial College London between January 2015 and September 2018. Parts of the work were done in collaboration with other researchers and open-source communities:

- Chapter 5: Semantic image synthesis is a joint work with Simiao Yu. My key contribution is the adversarial loss and network architecture.
- Chapter 6: I established TensorLayer, designed the abstraction and led its development.

I declare that the work presented in this thesis is my own, except where acknowledged above.

Acknowledgements

First, I would like to offer my sincere thanks to my supervisor Prof. Yike Guo for his helpful advice and active support over the four years. Further gratitude to Yike for leading an excellent example of hardworking and creative thinking. Besides his unique insights of adversarial learning and his humour made my office life so enjoyable.

Second, my thanks go to my second supervisor Prof. Paul M Matthews for his willingness to explain to me many brain concepts. In particular, I would like to thank him for providing me with financial support through the OPTIMISE Portal.

I also would like to express my gratitude to the following people who have helped me along my way of finishing my PhD: Simiao Yu, Akara Supratak, Chao Wu, Pan Wang, Guang Yang, Douglas Mcllwraith, Jingqing Zhang, Kai Sun, Felix Liang, Diana O'Malley, Yuanhan Mo and other members of Imperial Data Science Institute. A special thanks to Simiao Yu who shared his desk with me for two years.

Furthermore, I thank the people from the open-source community especially Luo Mai, Jonathan Dekhtiar and Guo Li for their helpful contributions to TensorLayer.

Most importantly, I want to thank my parents. I cannot thank them enough for giving me a comfortable environment with love and visiting me frequently in the past four years. I also would like to thank the rest of my family.

Dedication

This dissertation is dedicated to my parents, Yanying He and Cai Dong, for their constant love and support throughout my life.

'We think too small, like the frog at the bottom of the well. He thinks the sky is only as big as the top of the well. If he surfaced, he would have an entirely different view.'

Mao Zedong

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Deep learning is a subset of artificial intelligence that uses multi-layered neural networks to create autonomous learning from big data and perform tasks, such as image and text recognition [2]. Over the previous few years, deep learning has unlocked treasure troves of big data to drive advancements in healthcare, creating efficiencies in the power grid, improving agricultural yields, and discovering solutions to climate change. A deep neural network is a large trainable machine with multiple layers containing potentially millions of parameters to learn hierarchical representations and predict instance labels. This framework requires significant amounts of annotated data to train the parameters. Through progressive learning, the deep neural networks grind away and find nonlinear relationships in the data without requiring users to perform feature engineering [2].

Deep learning gained more attention when models outperformed previous methods [3] by more than 10% during the ImageNet visual recognition challenge in 2012 [4]. Deep learning next achieved success by demonstrating better performance compared to many traditional learning methods supporting applications in language translation [5], image segmentation [6], object detection [7], action recognition [8], and image captioning [1]. The success of deep learning is due to two primary reasons. First, the world-wide digitisation of information enables the creation of many large public labelled datasets, such as ImageNet [4] and MSCOCO [9], that enables researches to improve models collaboratively. Second, graphics processing units (GPUs) and tensor processing units (TPUs) provide superior processing speeds in deep learning computation that are orders of magnitude faster than conventional

central processing units (CPUs). To date, advanced deep learning algorithms have outperformed non-deep learning methods in many applications [2] as well as demonstrated comparable performance with humans in specific tasks such as image classification [10], clinical diagnosis [11], game playing (*e.g.,* AlphaGo) [12], and image captioning [1] [1].

The success of deep learning is partially attributed to rich datasets with abundant annotations [2]. Unfortunately, collecting and annotating such large-scale training data in practice is prohibitively expensive and time-consuming. This problem is more severe in generative tasks as the training data for many generative tasks can be difficult to obtain from the real world [13]. In some cases, it is even impossible to collect such data. For example, to modify the gender in the image of a given face, it is impossible to collect face images of different genders from the same person to supervise training a model. Another example is synthesising images conditioned on both images and text descriptions as with a dataset of just 1,000 images and 1,000 text descriptions, one million combinations (*i.e.,* $1,000 \times 1,000$) exist, and it is impractical to create one million ground truth images manually for supervised training. This problem also exists in synthesising images conditioned on text descriptions where an image could have infinite matching text descriptions. Having more training data is desirable to increase the robustness and generalisation of a deep model. However, it can be too costly to label more text descriptions or define a specific rule to augment the text descriptions manually [14, 15]. Therefore, for practical deep learning, exploring efficient deep learning algorithms for applications with limited training data is crucial.

Along with the problem of limited training data, deep generative models provide results that are difficult to control [16, 17], which limits their applicability to real-world scenarios. Since first proposed in 2014, the generative adversarial network (GANs) frameworks have achieved superior performance compared with other methods [2, 18]. For example, DCGAN [19] is the first to utilise the power of the deep neural networks to synthesise 64×64 face and bedroom images. The recent work on ProgressiveGAN [20], successfully synthesised 1024×1024 face images. However, randomly synthesising data in an open space makes it difficult for the deep generative models to be leveraged for many real-world applications because controlling the generated results to meet a specified requirement is challenging [17]. For example, instead of generating a random face of a lady, a real-world application may require the model to generate a lady with sunglasses. Therefore, the GAN research community began to study how to control the synthesis results by inputting auxiliary information into a vanilla

[1] `http://cocodataset.org`

GAN framework [16, 17]. ACGAN [16] synthesises images conditioned on discrete class labels such as car, bird or flower. A class label is input into the GAN network that outputs corresponding images containing cars, birds or flowers. Compared to discrete class labels as the input condition, text descriptions and images can be utilised to control the generated results by providing more auxiliary information. A text description can provide the information of a class label and the object attributes, such as the colour, location, action and background. Images can provide semantic visual information that the text description cannot provide. For example, images can include the pixel-level precise shape, colour, and location of an object. Therefore, developing the use of text descriptions and images for auxiliary information can achieve more controllable image synthesis compared to vanilla GAN-based frameworks and discrete label conditioned frameworks (*e.g.,* ACGAN).

To improve the practicability of deep generative models, we proposed several methods to address the challenges of limited training data and uncontrolled generated results. We first improve the visual quality of synthesised images for the text-to-image synthesis problem from limited training data problem by generating more text descriptions based on an image captioning model [1] to generate more training data and provide object attribute information for controlling the generated results. Second, we achieve unsupervised image-to-image translation for synthesising one type of image conditioned on another without requiring paired images to supervise training, which enables tasks, such as face gender translation, that cannot be performed through a supervised model [13]. This approach provides controllable generated results by using the semantic visual information from the conditioned input images. Third, we study the semantic image synthesis which synthesises images conditioned on images and text descriptions without requiring ground truth images to serve as the supervision, providing controllable generated results based on semantic visual information from the input images as well as object attribute information from the input text descriptions.

In addition to the development of algorithms for efficiently using data for generative tasks, this thesis designs a deep learning library, called TensorLayer, to facilitate the deep learning development. TensorLayer provides tools to rapidly build neural network models and routines to manage datasets, models and training pipelines. Deep learning development tends to revolve around experiments where researchers frequently experiment with different datasets, model architectures, and training methods [2]. The deep learning workflow represents a pipeline for developing a deep learning model, including creating and acquiring the dataset, training the model based on the training pipeline, evaluating its performance, and saving the model for future use [21]. Apart from model training, researchers

often need to archive the model and dataset for versioning, sharing, further retrieval, provenance [21], experiment reproduction, and training multiple models concurrently to speed up the hyper-parameter selection [2, 22].

Tools supporting model building and life-cycle management for a deep learning workflow increases its practicability. However, for model building, existing libraries such as Keras [2] and Pytorch [23] require researchers to spend extra effort when defining models and implementing new layers to deal with the layer output. Also, these libraries cannot exactly restore a certain part of the deep models based on different use cases, which can result in initialising unused layers and unnecessary computer memory consumption. For life-cycle management, existing tools, such as AzureML [3] and SeaHorse [4], offer a GUI-based interfaces with predefined templates that require researchers to complete each component of the template before execution. While such functionality is desirable for production purposes, it restricts the flexibility of deep learning researches because most algorithms are developed on-the-fly instead of being predefined.

To facilitate deep learning development, our research-oriented library, TensorLayer, delivers an abstraction to build deep models without the need to deal with the output shape of the layers manually. Moreover, TensorLayer provides the pre-trained model abstraction that can exactly restore a certain part of the deep models based on variety of use cases to avoid initialising unused layers and unnecessary computer memory consumption. For life-cycle management, TensorLayer abstracts the deep learning workflow into the model, dataset and task components, where a task contains the training details and results of an experiment. TensorLayer also supports running multiple tasks concurrently to speed model training. We evaluate TensorLayer by comparing it with other representative libraries. TensorLayer won the best open source software award issued by ACM Multimedia (MM) in 2017.

1.2 Contributions and Thesis Organisation

This thesis improves the practicability of deep learning by exploring efficient deep learning algorithms for applications with limited training data and designing a library for improving deep learning development. The specific contributions include the following: 1) Improve the visual quality of synthesised images for text-to-image synthesis without requiring additional domain knowledge. The generated

[2] `https://github.com/keras-team/keras`
[3] `https://studio.azureml.net`
[4] `https://seahorse.deepsense.ai`

results are controlled by using object attribute information implied from the input text descriptions. 2) Achieve unsupervised image-to-image translation without requiring paired images to supervise the training. The generated results are controlled with the semantic visual information of the input images. 3) Propose a method for semantic image synthesis without requiring ground truth images to supervise the training. The generated results are controlled by the object attribute information of the input text descriptions and the semantic visual information of the input images. 4) Develop a research-oriented deep learning development library, called TensorLayer, to support researchers in building models, implementing new layers, managing the deep learning workflow, and running multiple trainings concurrently. I will introduce the above contributions with more details in the following:

Efficient text-to-image synthesis

In Chapter 3, we propose a method to improve the image's visual quality for text-to-image synthesis for which the input text description contains object attribute information to describe the images. The challenge is that labelled text descriptions are limited in the training set. For example, the well-known image captioning dataset, MSCOCO, only has five text descriptions for each image [9]. However, in an ideal scenario, an image would have an infinite number of text descriptions that could be matched, *i.e.,* the text and image are highly multi-modal [24].

To alleviate the problem of limited labelled text descriptions, we use a state-of-the-art image captioning model with comparable performance to human [1] to synthesise more text descriptions for each image in the dataset. When we create more text descriptions to train the text-to-image generator, the training data includes more information and covers more variance of the data, which increases the robustness and generalisation of the model. Therefore, the visual quality of the synthesised images is improved. Moreover, we pre-trained an image captioning model on an image dataset that included labelled text descriptions and used the image captioning model to generate text descriptions on another dataset that did not have labelled text descriptions. Then, we leveraged the generated text to learn the text-to-image synthesis. To the best of our knowledge, this approach is the first that can provide a text-to-image synthesis from an image dataset that does not include labelled text descriptions.

Efficient image-to-image translation

In Chapter 4, we propose an unsupervised image-to-image translation method to synthesise images conditioned on input images, in which the input image can provide semantic visual information for the synthesis. The challenge of image-to-image translation is that paired images are expensive or even impossible to be collected in practice. For example, it is impossible to collect face images representing different genders of the same person. In this case, no supervised information can be collected to train the portrait gender transformation model, precluding the supervised methods for the portrait gender transformation task.

To achieve image-to-image translation without supervision, we develop a two-step (unsupervised) learning method to translate images between different domains by using unlabelled images without specifying any relationship to avoid the cost of acquiring labelled data. We verify the proposed method using applications of portrait gender transformation, face swapping and image inpainting. We also analysis the limitation of our approach and how the subsequent studies, such as CycleGAN [25], address this limitation. As our proposed method can translate images between two domains by using one generator, in contrast, the CycleGAN requires two generators to perform the translations, *i.e.,* one generator for domain A to B and another for domain B to A.

Efficient semantic image synthesis

In Chapter 5, we propose a novel method for semantic image synthesis where the synthesised images are conditioned on both the input images and text descriptions. This task is an extension of image-to-image translation by using text descriptions to control the image translation. Specifically, we utilise the object attribute information from the text description and the semantic visual information from the image to provide controllable image synthesis. For example, given an input image of a white flower with a text description, "this is a yellow flower", the input image is translated into a yellow flower while maintaining other semantic visual information, such as the flower's shape and background.

The challenge of semantic image synthesis is that the ground truth for the images after translations remains unknown, and it is impractical to create such labelled ground truth images manually. For example, given n images and m text descriptions, we can have $n \times m$ pairs of combination, meaning $n \times m$ ground truth images must be created manually if we use supervised learning. To solve this problem of unknown ground truth images, we propose a novel neural network model along with an

adversarial loss, which only requires the paired input image and text description for training. Our results demonstrate that the method can synthesise images to match the input text descriptions and maintain the semantic visual information of the input image.

Efficient deep learning development

In Chapter 6, our research-oriented library, TensorLayer, reduces the workload of researchers in building neural network models and managing the life-cycle of the deep learning workflow including the model, dataset and training pipeline. Specifically, we design a model abstraction method that frees researchers from dealing with the output shape of the layers that reduces the the workload when defining models and implementing new layers. For life-cycle management, we abstract the deep learning workflow into the three components of the model, dataset, and task, where the task contains the training details and results of an experiment. Our life-cycle management supports the deep learning workflow and facilitates the experiments by enabling multiple tasks to be run concurrently on multiple nodes (*e.g.,* different machines or GPUs) to speed up the deep learning development. We evaluate TensorLayer by comparing it with other libraries and demonstrate its efficacy using two case studies of training multiple models concurrently for speeding up hyper-parameter selection and running multiple data generators for speeding up the training of deep reinforcement learning.

1.3 Publications

My PhD study has led to a number of publications as follows:

- **Chapter 3: Learning Text to Image Synthesis with Textual Data Augmentation.** *International Conference on Image Processing (ICIP).* ***(Oral)****. 2017. H. Dong, J. Zhang, D. Mcllwraith, Y. Guo.*

- **Chapter 4: Unsupervised Image-to-Image Translation with Generative Adversarial Networks.** *arXiv 2017. H. Dong, P. Neekhara, C. Wu, Y. Guo.*

- **Chapter 5: Semantic Image Synthesis via Adversarial Learning.** *International Conference on Computer Vision (ICCV). 2017. H. Dong*, S. Yu*, C. Wu, Y. Guo.*

- **Chapter 6: TensorLayer: A Versatile Library for Efficient Deep Learning Development.** *ACM Multimedia (MM). 2017. H. Dong, A. Supratak, L. Mai, F. Liu et al.*

In addition, the following publications over the course of this thesis and have been impacted but not directly contributed to this thesis.

- **Mixed Neural Network Approach for Temporal Sleep Stage Classification.** *IEEE Transaction on Neural Systems and Rehabilitation Engineering (TNSRE). 2017. H. Dong, A. Supratak, W. Pan, C. Wu, P. M. Matthews, Y. Guo.*

- **DeepSleepNet: a Model for Automatic Sleep Stage Scoring based on Raw Single-Channel EEG.** *IEEE Transaction on Neural Systems and Rehabilitation Engineering (TNSRE). 2017. A. Supratak, H. Dong, C. Wu, Y. Guo.*

- **Automatic Brain Tumor Detection and Segmentation Using U-Net Based Fully Convolutional Networks.** *Medical Image Understanding and Analysis (MIUA).* ***(Oral)****. 2017. H. Dong*, G. Yang*, F. Liu*, Y. Mo, Y. Guo.*

- **Deep Learning using TensorLayer.** *Publishing House of Electronic Industry (PHEI). 2018. ISBN: 9787121326226. H. Dong, Y. Guo, G. Yang et al*

- **Dropping Activation Outputs with Localized First-layer Deep Network for Enhancing User Privacy and Data Security.** *IEEE Transaction on Information Forensics and Security (TIFS). 2017. H. Dong, Z. Wei, C. Wu, Y. Guo.*

- **Generative Creativity: Adversarial Learning for Bionic Design.** *arXiv. 2018. S. Yu. H. Dong, W. Chao, Y. Guo.*

- **DAGAN: Deep De-Aliasing Generative Adversarial Networks for Fast Compressed Sensing MRI Reconstruction** *IEEE Transaction on Medical Imaging (TMI). 2017. G. Yang*, S. Yu*, H. Dong et al.*

- **Deep De-Aliasing for Fast Compressive Sensing MRI.** *arXiv. 2017. S. Yu*, H. Dong*, G. Yang, G. Slabaugh et al.*

- **TensorDB: Database Infrastructure for Continuous Machine Learning.** *International Conference on Artificial Intelligence (ICAI). 2017. F. Liu, A.Oehmichen, J. Zhang et al.*

- **Survey on Feature Extraction and Applications of Biosignals.** *Machine Learning for Health Informatics. Springer. 2016. A. Supratak, C. Wu, H. Dong et al.*

- **DropNeuron: Simplifying the Structure of Deep Neural Networks.** *arXiv. 2016. W. Pan, H. Dong, Y. Guo.*

- **A New Soft Material based In-the-ear EEG Recording Technique.** *Engineering in Medicine and Biology Society (EMBC).* ***(Oral)****. 2016. H. Dong, P. M. Matthews, Y. Guo.*

* Indicates co-first author.

Chapter 2

Background

In this chapter, we describe the fully connected layer, activation and loss functions, and convolutional and recurrent neural networks that are widely used throughout this thesis. The training details of neural networks are also explained, including gradient descent, error back-propagation, regularisation, and transfer learning. Finally, we introduce the generative adversarial network (GAN) [26] to review the vanilla GAN, deep convolutional GAN [19], and conditional GAN [17].

2.1 Deep Neural Networks

Deep learning is a class of machine learning methods based on deep neural networks [2]. In 2012 during the image classification challenge event, ImageNet [4], a new neural network design called Alexnet [3] outperformed previous non-deep learning methods by 10.8%. Since then many deep learning methods achieve state-of-the-art performance on machine learning tasks such as vision [1, 27, 28], image processing [29,30], and natural language processing [5]. The most common neural networks applied today are fully connected networks, convolutional neural networks (CNN), and recurrent neural network (RNN) [2]. The architectures of those neural networks are described in the following along with the concepts of encoding, decoding, and latent space.

2.1.1 Fully connected network

A neural network neuron is the fundamental element of the neural networks [2] as shown in Figure 2.1 with three inputs and one output. This neuron is represented by Equation (2.1), where x are the

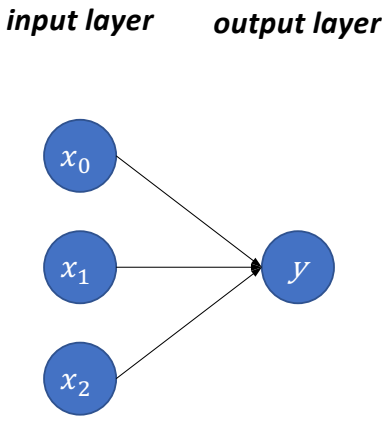

Figure 2.1: An example of a single neural network neuron.

input values, w are the weights, and y is the network output. Each arrow in Figure 2.1 represents one weight value. As the output directly connects to the inputs, this neuron can be considered as a network with a single layer.

$$y = x_0 * w_0 + x_1 * w_1 + x_2 * w_2 \tag{2.1}$$

Given varied weights, the output will be sensitive to different inputs. For example, y may be a score determining if we are to play football. If y is large, then we play. To determinate this score, x_0 represents the weather, x_1 is the expense of the football field rental, and x_2 is the distance to the field. These inputs are considered the features with respect to the output. If the whether is the most critical factor, then we can set w_0 to a large positive value and set w_1 and w_2 to smaller values. If w is set to zero, then the corresponding input feature is discarded.

Expanding from this single neural network neuron, a network can have multiple outputs. Figure 2.2 shows an example of three outputs each of which can be computed by Equation (2.1) to which a bias value may be added. A bias value allows the output value to be shifted higher or lower to better fit the input data. Equation (2.2) illustrates how to compute each output y_i, where i is the index of outputs. Because every input of the layer is linked to all its outputs, this layer is referred to as a fully connected layer [2]. In practice, a fully connected layer is implemented with a matrix multiplication as in Equation (2.3) shows.

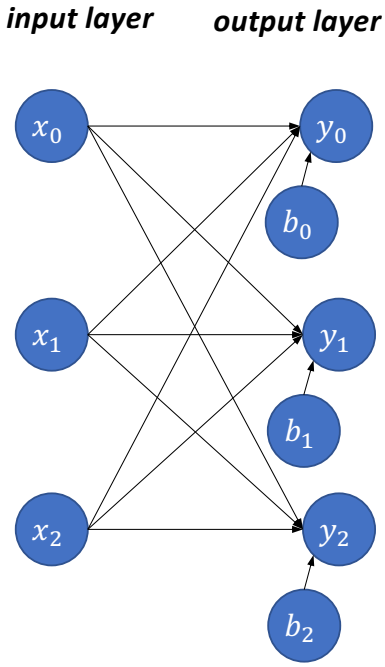

Figure 2.2: An example of three neural network neurons.

$$y_i = x_0 * w_{0,i} + x_1 * w_{1,i} + x_2 * w_{2,i} + b_i \tag{2.2}$$

$$y = x * W + b \tag{2.3}$$

where x are the input values, W is the weight matrix representing the connections, b are the bias values, and y includes the output values. Considering 2.2 as an example, since the input has three values, x can be represented by a 1×3 vector, *i.e.,* a row vector with one row and three columns. For the output with three values, y can also be represented by a 1×3 vector. The weight matrix W and bias b are a 3×3 matrix and a 1×3 vector, respectively, and are called the network or model parameters. Then, by using multiple neurons, we can obtain multiple outputs. For example, the outputs can represent if we should play football, basketball or tennis. Here, the network outputs three scores for three classes of sports.

A multi-layer perceptron (MLP) [31, 32] extends from a single, fully connected layer, and consists of at least two fully connected layers. Figure 2.3 presents a MLP consisting of two more fully connected

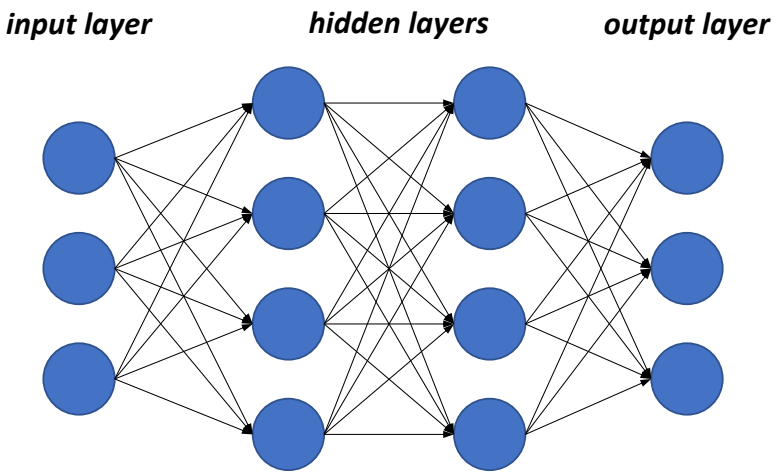

Figure 2.3: A multi-layer perceptron with two hidden layers.

layers compared with a single fully connected layer. The biases are not drawn to simplify the figure. The layers between the inputs and outputs are called "hidden" because they cannot be directly accessed from outside the network.

By stacking a new layer on top of an existing layer, the new layer is considered to use the output of the previously existing layer as its input features [2]. Therefore, compared with a single fully connected layer, MLP can fit more complex input data. In other words, MLP can have more representational capability than a single layer.

2.1.2 Activation functions

Continuing with our example, for a given neural network, the output y can represent specific scores, such as the probability of playing football. To represent the probability from 0% to 100%, it is of common practice to apply a function to scale the output to a value between 0 to 1. Therefore, the network output is non-linear such that it is not a linear combination of the inputs. Also, in order to represent a more complex function by using a neural network, the network hidden outputs can be non-linear [2]. Activation functions provide the non-linearity on the layers outputs, and their design remains an active researched area. The following four activation functions are applied most frequently.

Sigmoid function

The logistic sigmoid is a traditional activation function that provides non-linearity to a neural network. With an output range between 0 to 1, the sigmoid is used for the output of binary classifiers that result in one probability value of either 0 or 1. For example, applying the sigmoid function to the signal neuron in Figure 2.1 provides a simple binary classifier that receives three input features. Given a probability value between 0 to 1, a threshold value (*e.g.,* 0.5) is to set so that if the value is greater than the threshold, then it classifies the input features as a positive sample. The sigmoid function is defined as follows where z denotes the output of a layer.

$$f(z) = \frac{1}{1 + e^{-(z)}} \tag{2.4}$$

Hyperbolic tangent function

Similar to the sigmoid, the hyperbolic tangent (tanh) also scales the output layer to a limited range of values. With an output range between -1 to 1, this function is often used for regression, such as for an output image with pixel values between -1 to 1. As well as being applied to the network outputs, the sigmoid and hyperbolic tangent functions are used in the hidden layers [33] to provide non-linearity to the network. The hyperbolic tangent function is defined as follows.

$$f(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{2.5}$$

Softmax function

Again, with Figure 2.2 as an example network with three outputs and three inputs, with these multiple outputs, multi-class classification can be performed, *i.e.,* classify the input into one of three or more classes. The softmax function is designed for the output layer of a multi-class classifier to not only limit all outputs to 0 to 1, as with the sigmoid, but also ensure the sum of each output equals 1, *i.e.,* the sum of all probabilities must be 100%. The softmax function is defined as follow.

$$f(z)_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}} \tag{2.6}$$

where K is the number of classes or outputs. Softmax first applies an exponential function e^z to each output and then normalises each by dividing it by the sum of all outputs. The fully connected layer is often used as the output layer for neural network classifiers. In practice, using the softmax activation, a network can output a vector to represent the probabilities of each class whereas with the logistic sigmoid, a network can output a single value to represent one probability for binary classification.

Rectified linear unit

The rectified linear unit (ReLU), also known as a rectifier [33], is a function that sets the negative values to zero for the purpose of feature selection. For example, in MLP, the output values of a layer are a combination of the input values. If an outputs is unnecessary for the final task, such as in classification, then ReLU allows the network to set this output to zeros. The ReLU function is defined as the following.

$$f(z) = \begin{cases} 0 & , when\ z <= 0 \\ z & , when\ z > 0 \end{cases} \tag{2.7}$$

A recent study [33] showed that ReLU has a better performance on the hidden layers compared to that of the sigmoid and hyperbolic tangent. So, this activation function is becoming a default choice for deep neural networks [10, 34, 35].

However, merely setting negative values to zero will lead to information loss. A solution was proposed with the leaky ReLU [36], defined in Equation (2.8), where α is a small positive value to control the slope (*e.g.,* 0.1 and 0.2) so that the information from the negative values can pass through to the next layer.

$$f(z) = \begin{cases} \alpha * z & , when\ z <= 0 \\ z & , when\ z > 0 \end{cases} \tag{2.8}$$

In addition, the parametric ReLU (PReLU) [37] was also proposed to consider α as a trainable parameter.

2.1.3 Loss functions

In deep learning, loss functions are defined to quantify an error, known as the loss value, between the predicted and targeted (*i.e.,* ground truth) outputs. The loss value is used as the goal for optimising the neural network parameters, such as the weights and biases. Specifically, optimising a given neural network minimises the defined loss value $\mathcal{L}$ by updating the network parameters θ. Gradient descent [38] is commonly used to update the parameter by computing the partial derivatives of the loss with respect to the network parameters, which can be written as $\frac{\partial \mathcal{L}}{\partial \theta}$. Details about gradient descent and neural network training is included in Section 2.2.

For the classification task discussed above, logistic regression and cross-entropy losses are commonly used. For regression problems where the target outputs are continuous values, such as the predicted temperatures or image pixels, the mean squared error (MSE) and mean absolute error (MAE) are mostly used. These regularly utilised loss functions are described below.

Logistic regression loss

Logistic regression loss is commonly used for binary classification. For a network with one output value, such as in Figure 2.1 with a sigmoid function, minimising the logistic regression loss is equivalent to guiding the network toward an output of 0 or 1 based on its input. The loss function is defined as follows.

$$\mathcal{L} = -\frac{1}{m}\sum_{i=1}^{m}(y^i \log(\hat{y}^i) + (1 - y^i)\log(1 - \hat{y}^i)) \tag{2.9}$$

where y is the binary target label of 0 or 1, $\hat{y}$ is the predicted probability value from the sigmoid output, and m is the number of classes. A smaller loss represents a smaller gap between the target y and prediction $\hat{y}$. If y and $\hat{y}$ are equal, then the loss is zero.

Cross-entropy loss

Cross-entropy loss is used for training multi-class classifiers, which requires the neural network to output multiple values that represent the probabilities of each class instead of a single value as in

binary classification. The cross-entropy loss is defined as follows.

$$\mathcal{L} = -\frac{1}{m}\sum_{i=1}^{m}\log(\hat{y}^i_{y_i}) \tag{2.10}$$

where m denotes the number of output values, y and $\hat{y}$ are the target and prediction, and $\hat{y}^i_{y_i}$ represents the predicted probability of the target label.

$\mathcal{L}_p$ norm

Given a vector x, $p-$norm [2] can measure its scale such that a vector with larger values represents a large scale, and is defined as the following, where p is an integer greater or equal to 1.

$$\begin{aligned} ||x||_p &= (\sum_{i=1}^{n}|x_i|^p)^{1/p} \\ i.e., ||x||_p^p &= \sum_{i=1}^{n}|x_i|^p \end{aligned} \tag{2.11}$$

In deep learning, the $p-$norm measures the difference between two vectors written as $\mathcal{L}_p$, as in Equation (2.12), where y and $\hat{y}$ are the target and prediction, respectively.

$$\mathcal{L}_p = ||y-\hat{y}||_p^p = \sum_{i=1}^{n}|y_i-\hat{y}_i|^p \tag{2.12}$$

Mean squared error

The mean squared error (MSE) is the averaged $\mathcal{L}_2$ norm and is used for regression problems in which the output of the neural networks contains continuous values, such as the pixels of an image or a scalar value. The MSE is defined as follows.

$$\mathcal{L} = \frac{1}{m}\sum_{m}^{i=1}(y^i - \hat{y}^i)^2 = \frac{1}{m}||y - \hat{y}||_2^2 \tag{2.13}$$

where m is the number of examples, and y and $\hat{y}$ are the target and prediction, respectively.

Mean absolute error

Similar to MSE, the mean absolute error (MAE) is the averaged $\mathcal{L}_1$ norm, known as the least square error, The MAE is also used for regression problems and is expressed as follows.

$$\mathcal{L} = \frac{1}{m}\sum_{m}^{i=1}|y^i - \hat{y}^i| = \frac{1}{m}||y - \hat{y}|| \tag{2.14}$$

where m is the number of examples, and y and $\hat{y}$ are the target and prediction, respectively. Both MSE and MAE minimise the difference between y and $\hat{y}$. MSE offer a better mathematical property making it easier to compute the partial derivative as required for gradient descent. In contrast, due to the absolute value term, MAE requires more complicated computation of the partial derivative. In addition, when the difference between y and $\hat{y}$ is greater than 1, MSE can result in a larger error compare to MAE (*i.e.,* $|y - \hat{y}|$ vs $|y - \hat{y}|^2$) making the network more sensitive to this prediction [2].

2.1.4 Encoder and decoder

Latent representation

This thesis focuses on visible data, such as images and text descriptions, and latent representations, such as the hidden output of a neural network. Latent representations are the structures and features of corresponding values of visible data. The intrinsic meaning of latent representations cannot be directly understood as they are the unobserved abstraction of the visible data, and each value represents specific features of the visible data. For example, facial expressions in an images can be modified by changing the values in the corresponding latent representation (*e.g.,* the latent variables), which means the content of the visual data can be controlled by changing these variables. In deep learning, visible

data and latent representations can be converted into each other via different neural networks. The latent space (or distribution) represents the set of all latent variables for a set of visible data. For example, all possible text descriptions and images can be converted into text and image latent spaces, respectively.

Encoder

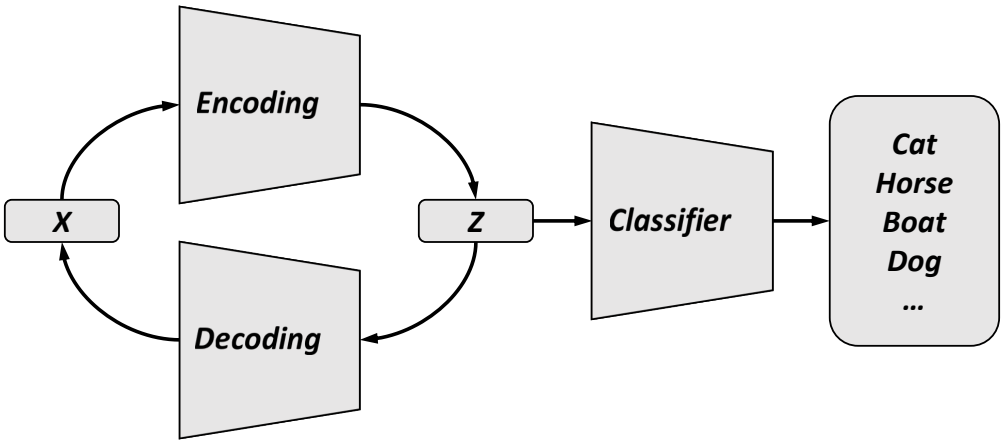

Figure 2.4: Illustration of encoding and decoding where x and z denote visible data and its latent representation, respectively.

Deep learning has been outperforming traditional methods for many image tasks [2, 39], such as object detection [40], image classification [3, 10], and image segmentation [41, 42]. These types of tasks train an encoding network to extract features from a given visual input for subsequent tasks. The simple example in Figure 2.4 shows an image classification tasks with an image x as input that outputs the probability of different classes (*e.g.,* cat, horse or boat). These discriminative tasks are successfully with this approach when the approximate capacity of the neural network finds a good latent representation z for the visible input data. In deep learning, the process of transforming visible input data, such as an image, text or video, into a latent representation, alternatively known as embedding or hidden representation, is referred to as "encoding".

Decoder

In deep learning, the decoder imposes a decoding network that generates the corresponding visual data from its latent representation. This routine is the inverted process of the encoding network

as exemplified as the inverted image encoding shown in Figure 2.4 to generate the image. Another example is image captioning (*i.e.,* generating a text description for an input image) with an inverted process of text-to-image generation to synthesis images conditioned on the given text descriptions. In a later chapter, we introduce the convolutional neural network (CNN) and recurrent neural network (RNN) for encoding and decoding images and text descriptions, respectively, as well as how to combine a CNN and generative adversarial network (GAN) for image generation.

2.1.5 Convolutional neural networks

Convolution

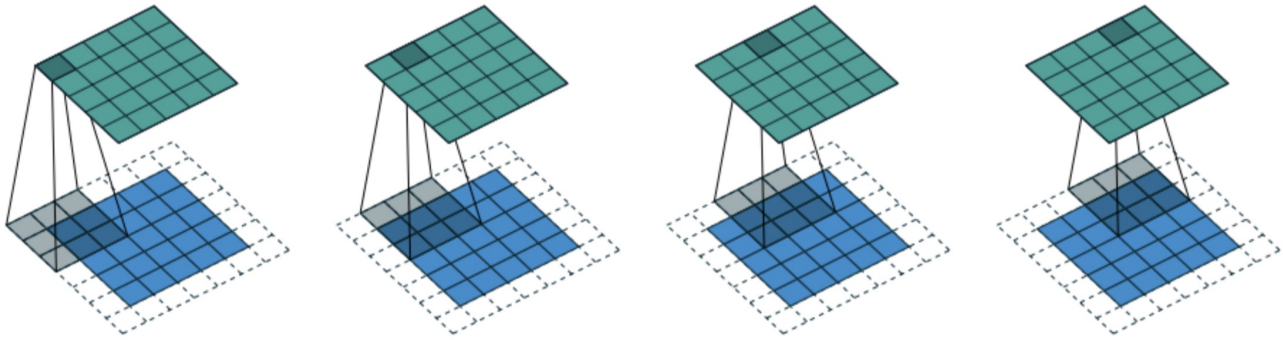

Figure 2.5: An example of 2D convolution where the blue squares denote the input values, dotted lines represent the padding values, and green squares denote the output values [43].

The convolutional neural networks (CNNs) is widely used to solve computer vision problems [34, 40, 42, 44]. Mathematically, the convolutional operation measures how two signals overlap as one passes over. For example, the edge detection in digital image processing is a specific type of convolutional operation with a pre-defined filter that passes over an image to identify the edges that match the filter pattern. Specifically, when a pre-defined filter passes over the image, the values of the filter and the values of a local patch are multiplied element-wisely and summed into a single output value. If the pattern of a local patch is similar to the filter, then the output value is large.

Similar to edge detection, CNNs apply many filters to an image and obtain many feature maps. Just as with the weights and biases of fully connected networks, the filters are the network parameters optimised according to a loss function. Figure 2.5 shows an example of a 2D convolution on a 5×5 matrix with a filter size of 3×3, a stride of 1 and padding of 1 [43]. The stride represents the number of values to skip when the filter passes over the input, while a larger stride leads to a smaller

output feature map. The padding represents the number of values appended to the edge of input before convolution. The dotted lines in the figure illustrate how to pad zero on the edge of the input so the output and input are the same size.

Unlike the fully connected layer, a CNN filter can be applied to different locations of the input. This allows the CNN to largely reduces the number of parameters while providing space invariance. Similar to MLP, multiple CNN layers can be stacked together to build a deep CNN offering the advantage of encoding features layer-by-layer. The first layer learns the features of input data followed by the second layer that learns higher-level features based on the features extracted from the first layer. Deeper CNNs have a layer output can represent higher-level features with a larger receptive field [43]. Therefore, a value from the feature map can represent a larger region of the input data, which is also why a larger image requires deeper CNN encoding.

In recent years, many studies focused on designing effective deep CNN architectures, such as VGG [45], MobileNet [46], SqueezeNet [47], Residual network (ResNet) [10], and Inception network [48]. ResNet [10] was the first to outperform humans at the ImageNet [4] classification challenge and to successfully train a network with one thousand convolutional layers. In this thesis, VGG, Inception network, and the concept of ResNet are featured for specific purposes.

To understand the features learned by a deep CNN, Figure 2.6 visualise the features learned by a VGG network by maximising the activation values of different CNN layers. The shallow layers are only capable of learning the edge features. Going to deeper layers, the receptive field becomes larger, and it begins to learn texture features and object parts. For example, the "conv5" in the bottom right of Figure 2.6 begin to show the profiles of a dog and cat. This visualisation demonstrates that objects can be conceptualised by deep CNN similar to the human visual system.

Pooling

Figure 2.7 shows the architecture of a VGG16 network with 13 CNN layers. The blue rectangles denote the CNN layers, and the white rectangles denote the max pooling layers that reduce the size of the feature map by calculating the averaged or maximum location values. At the end of the model, fully connected layers are applied to output 1,000 probabilities corresponding to different classes of the ImageNet dataset. Specifically, pooling is a down-sampling operation that applies aggregate statistics for local features. The output of a CNN layer typically includes many feature maps, which leads to

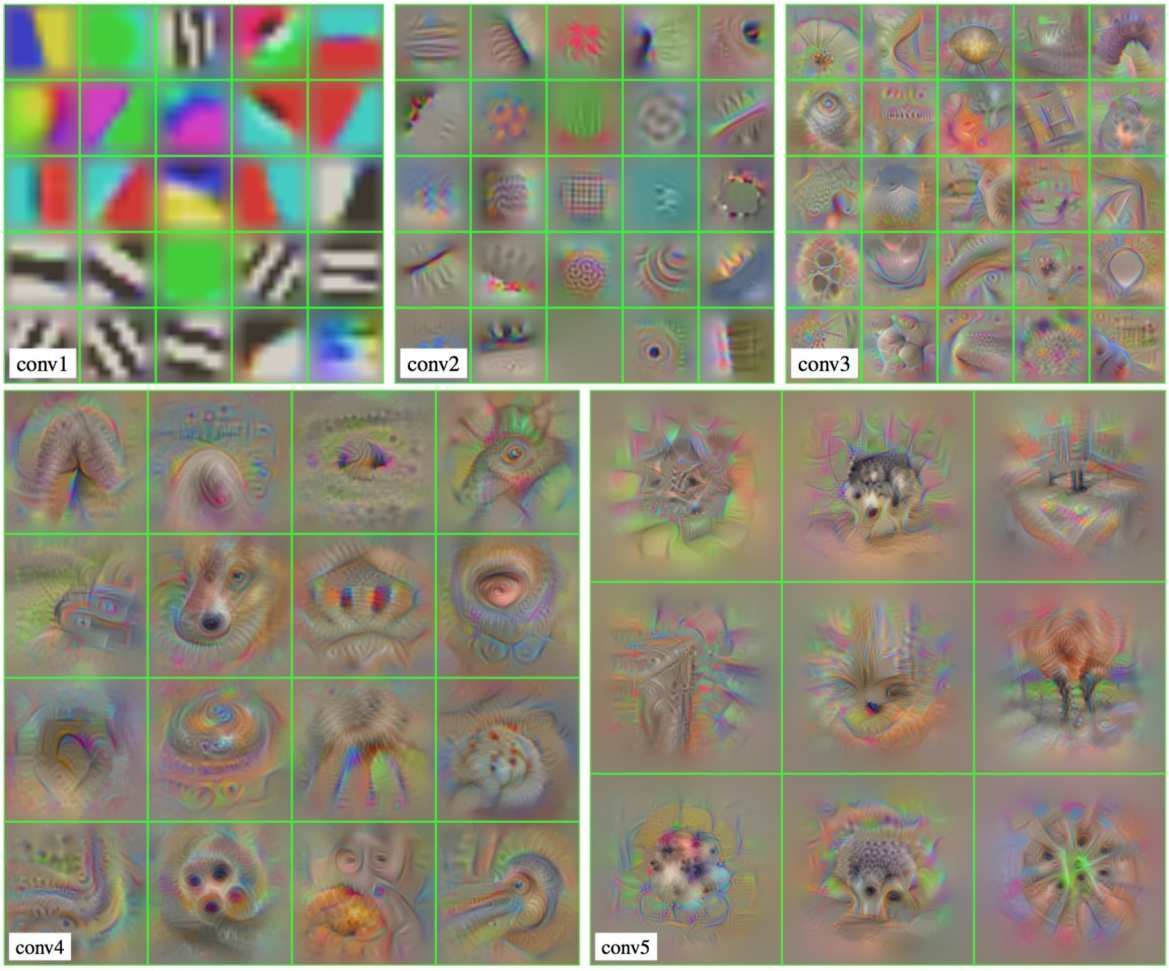

Figure 2.6: Activation maximisation of the first filters of each convolutional layer in a VGG. The notations of "conv1" through "conv5" distinct hidden layers, where a larger number represents a deeper hidden layer. Image replicated from [49].

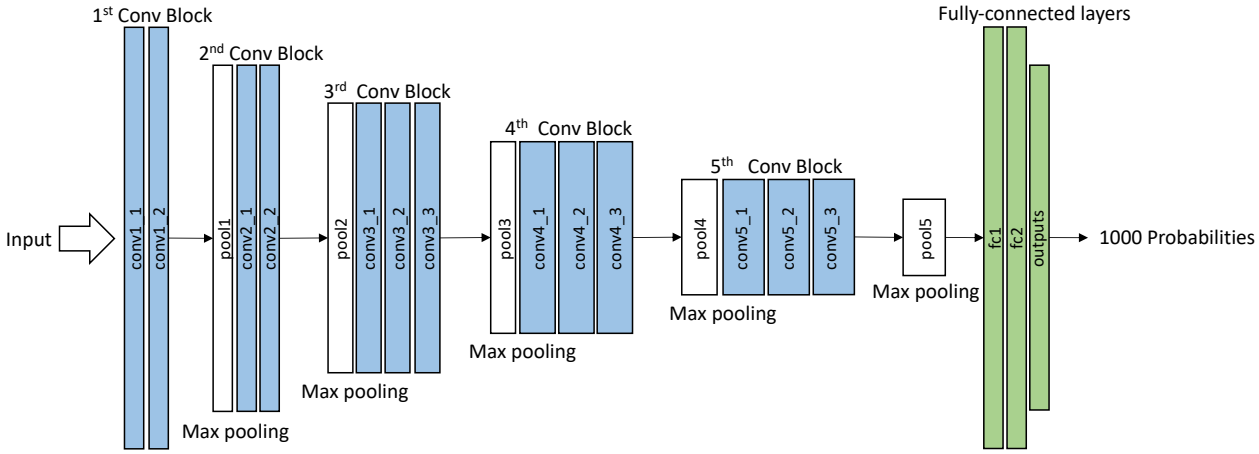

Figure 2.7: The VGG16 architecture with blue rectangles representing the convolutional layers, white rectangles denote the max pooling layers and green rectangles represent fully connected layers.

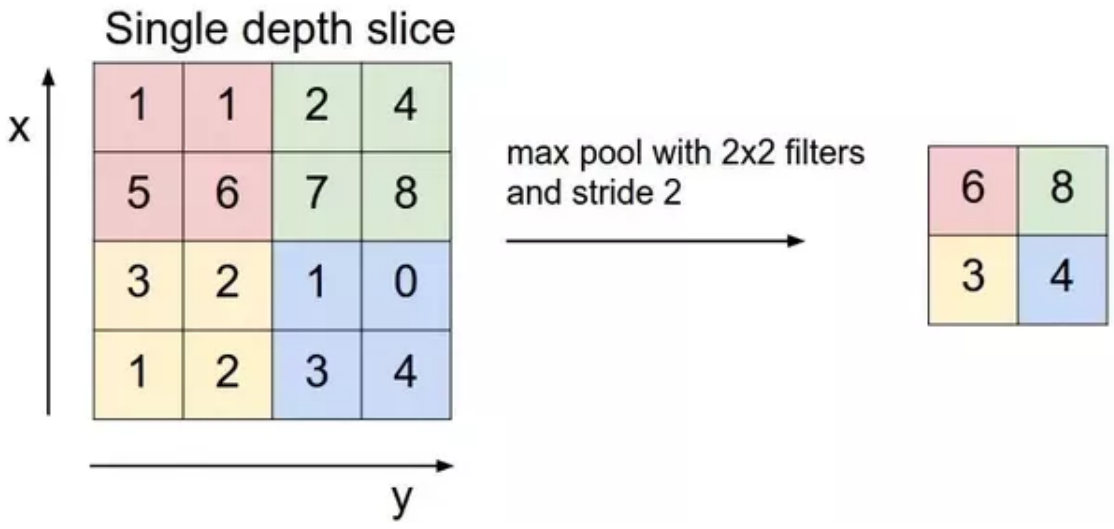

Figure 2.8: An example of 2D max pooling. Image is replicated from http://cs231n.github.io

high computational costs for the next layer. The pooling operation reduces the size of the feature maps by using the maximum or averaged values from the local patches that correspond to the max and mean poolings. Figure 2.8 provides an example of max pooling with a filter size of 2×2, stride 2, and no padding.

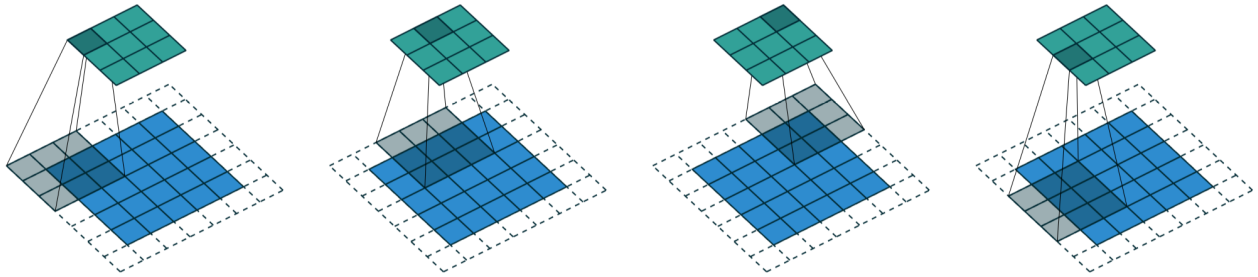

Figure 2.9: An example of 2D convolution with a stride of 2. The blue squares denote the input values, dotted lines represent the padding values, and green squares denote the output values. Image is taken from [43].

Another way to down-sample the feature maps is to set the stride value of the convolution layers higher than 1. Figure 2.5 includes an example where the stride is equal to 1. If the stride is set to 2, as in Figure 2.9 shows, then the filter will skip two values resulting in an output of 3×3. Even though skipping values during convolution leads to information lost, this approach is still used in generative adversarial networks (GAN), as described in Section 2.3. This method works well because by using the stride for down-sampling, the CNN can learn its spatial down-sampling function [19], which stabilises the GAN training. In this thesis, a convolution with a stride of 2 is used for the down-sampling operation in the GAN.

Deconvolution

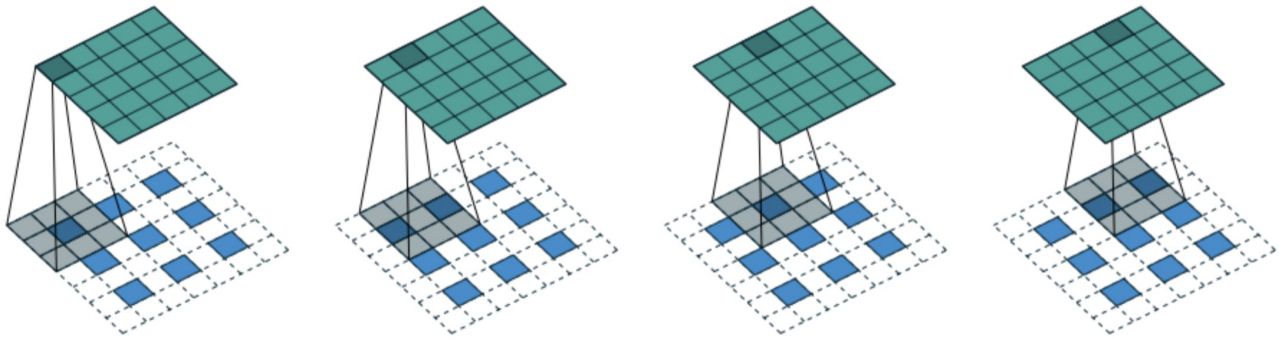

Figure 2.10: An example of 2D transposed convolution. Image is taken from [43].

Convolution and pooling are down-sampling operations that encode the image into high-level features. To decode the encoded features, up-sampling operations are required. Deconvolution, also known as transposed convolution [43], is typically applied for up-sampling. Figure 2.10 demonstrates the operation of 2D deconvolution, which is normal convolution with the exception that zeros are inserted between the input values by a given stride number. As in the figure, the size of the input feature map is 3 × 3, and with a 3×3 filter and a stride of 2 (*i.e.,* insert one zero between the input values), then the output size is 5 × 5. Apart from deconvolution, there are different types of up-sampling operations, such as the sub-pixel convolution [29] for super-resolution tasks and the resize convolution for reducing the checker-board artefact [50].

2.1.6 Word embedding

In this thesis, "text" represents text descriptions, *i.e.,* sentences with multiple words, a "word" represents a single word. To represent text in a computer, a string format is used. However, to perform computation with text, each word of the text must be converted into a numerical representation.

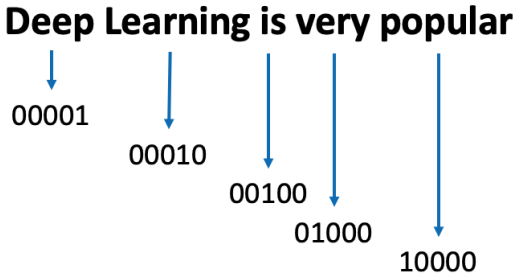

Figure 2.11: An example of a one-hot vector for words.

One-hot encoding is a simple way to represent words as shown in Figure 2.11, in which each word is assigned a unique bit "1" in a vector. To encode the entire sentence into the text representation (*i.e.,* text embedding), the list of text embeddings are feed into a recurrent neural network (RNN), as will be introduced in the next subsection. The drawback of one-hot encoding is its vector length is equal to the number of words in the vocabulary of the context of the application, which could contain thousands of words [9]. So, one-hot vector can lead to challenges of dimensionality [2].

Another common method to represent text is called bag-of-words [51]. Instead of converting words into a numerical format, the bag-of-words approach directly uses the word frequencies to represent the text. For example, given the text "we like deep learning, do we?", we can assign based on occurrences of each word, "we" = 2, "like"=1, "deep"=1, "learning'=1 and "do"=1. The drawback of bag-of-words is that it does not contain any temporal information, *i.e,* the semantic meaning from the order of the words is ignored.

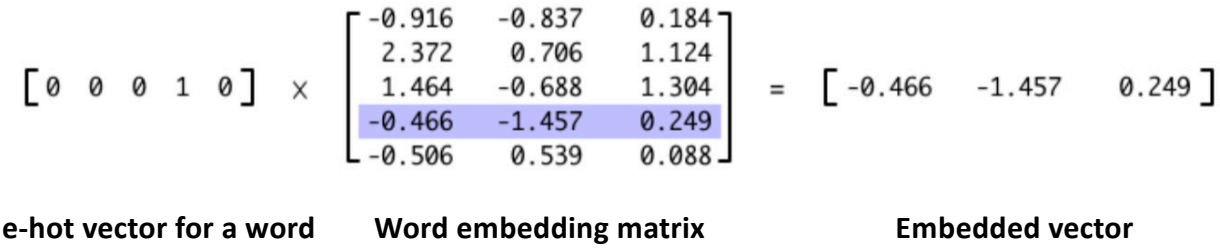

Figure 2.12: An example of word embedding.

Neural network-based word representation methods have been widely used more recently [1, 5]. For enhancing the one-hot-vector, a word embedding vector with floating-point numbers to represent a word can contain more information and largely reduce the length of the one-hot vector. Figure 2.12 shows an example of word embedding that multiplies the one-hot vector with the word embedding matrix to encoded the word into a vector of 3 values, which can be fed into the remainder of the network, such as a softmax layer for classifying the sentiment of a word. The row and column numbers of the word embedding matrix equal to the number of words in the vocabulary and the embedding size, respectively. The embedding size is often much smaller than the vocabulary size, which depends on the application and data. For example, one study [1] used a embedding size of 512 to represent the words in a vocabulary containing thousands of words.

Figure 2.13: A visualisation of word embedding space using t-SNE.

In practice, we would not use a one-hot vector to represent a word and multiply it with the word embedding matrix. Instead, we assign an unique ID to each word in the vocabulary, where the ID index ranges from 0 to the vocabulary size minus 1. Then, for a given word, we can directly obtain the corresponding row vector from the word embedding matrix according to the word ID. Considering Figure 2.12 for an example, instead of using one-hot vector, if we assign an ID of 3 to represent the word, then we can use this indexing to directly obtain the blue vector from the word embedding matrix without the need for matrix multiplication.

To generate the text representation (*i.e.,* text embedding) from the word embeddings, the word embed-

dings in the sentence must be further encoded into one vector. In practice, the word embedding matrix is updated together with the rest of the network (*e.g.,* recurrent neural network, RNN) [1, 5, 52]. In this thesis, we use a method [52] designed for text-and-image applications to pretrain the text encoder by including both the word embedding matrix and RNN, and details are provided in Chapter 3.

There also exist methods for pretraining the word embedding matrix, such as Word2vec [53] and Glove [54], which map similar words into adjacent latent representations. Figure 2.13 visualises a trained word embedding matrix via Word2vec with TensorLayer [1]. By projecting the word embedding vectors into a two-dimensional space using t-SNE [55], the words in the same category (*e.g.,* France, China, and Canada for countries) are seen to have smaller separating distances compare to unrelated words.

2.1.7 Recurrent neural networks

Vanilla recurrent neural network

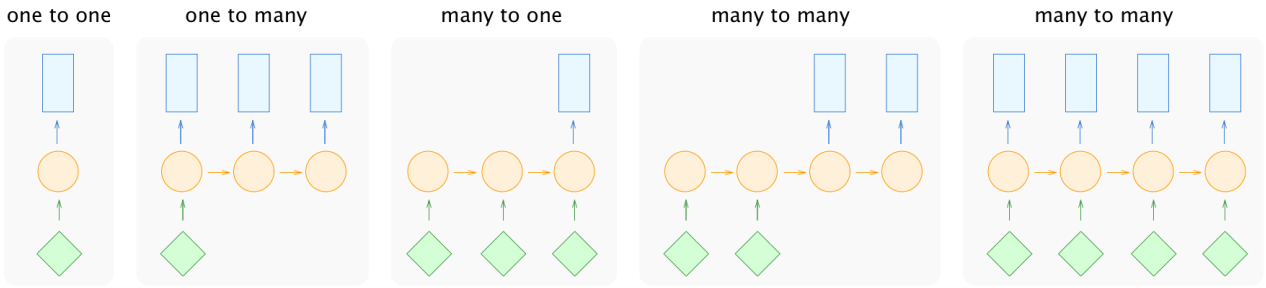

Figure 2.14: Diagrams representing models of feed-forward and recurrent neural networks for different purposes. The green diamonds represent the inputs, the orange circles denote the hidden informations, and blue rectangles represent the outputs.

Both the convolutional neural network (CNN) and fully connected layers are categorised as feed-forward neural networks (FNNs) as they only pass data layer-by-layer and obtain one output for one input (*e.g.,* an image in input with an output of a class label). There are many time-series data sets, such as language, video, and biosignals that could not fit into this framework. The recurrent neural network (RNN) is an expanded deep learning architecture [2] designed for processing time-series data, such as sequentially feeding the words from a sentence into a RNN. Figure 2.14 illustrates the data flow of FNN and RNN for a variety of models, and are described from left to right in the following:

[1] `https://github.com/tensorlayer/tensorlayer/tree/master/examples/text_word_embedding`

- One-to-one. An FNN without using an RNN where one input data results in one output data, *e.g.,* image classification that inputs an image and outputs a class label.

- One-to-many. An RNN decoder decodes one data to sequential outputs, *e.g.,* image captioning [1] that inputs an image and outputs a text description.

- Many-to-one. An RNN encoder encodes a sequential data to one output data, *e.g.,* text sentiment classification that inputs a complete sentence and outputs a classification result. Note that word embedding algorithms, such as Word2vec [53], are not for many-to-one, as they input one single word and output the corresponding word embedding.

- Asynchronous many-to-many. Also known as Seq2Seq [56], this model consists of one RNN encoder and one RNN decoder where the former encodes sequential data into its latent representation, and the latter decodes the latent representation to sequential data, *e.g.,* language translation that encodes a complete sentence into latent space before beginning the translation process [57].

- Synchronous many-to-many. An RNN receives one data and outputs one data for each step, *e.g.,* real-time video anomaly detection that uses the sequential information to predict subsequent frames.

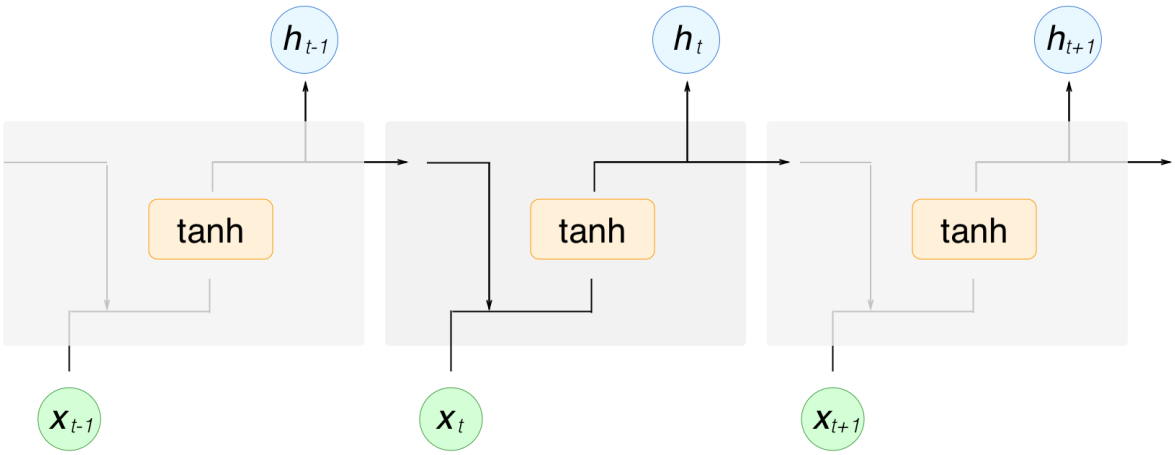

Figure 2.15: Diagram of vanilla recurrent neural network.

Figure 2.15 presents the architectural of a vanilla RNN. Here, only one RNN exists and it is reused for every time step with an output of a hidden state h, which contains the information from the previous

time step. The initial hidden state h_0 is usually set to zeros. Then at each subsequent time step, a new hidden state h_t is computed according to the previously hidden state h_{t-1} and the new input data x_t. The hyperbolic tangent function scales the hidden state to values between -1 and 1. The definition of the hidden state vector h is as follows:

$$h_t = tanh(x_t W_{xh} + h_{t-1} W_{hh} + b_h) \tag{2.15}$$

where W_{xh} and W_{hh} are two separated matrices and b_h is the bias vector. By feeding the hidden state h into the other network, it can be used for a variety of purposes as shown in Figure 2.14. For example, text sentiment classification is a "many-to-one" scenario where the input to the RNN encoder is the word embeddings of every word in a given text, and the final hidden state is fed into a softmax output layer that outputs the probabilities of different sentiments.

Long short-term memory

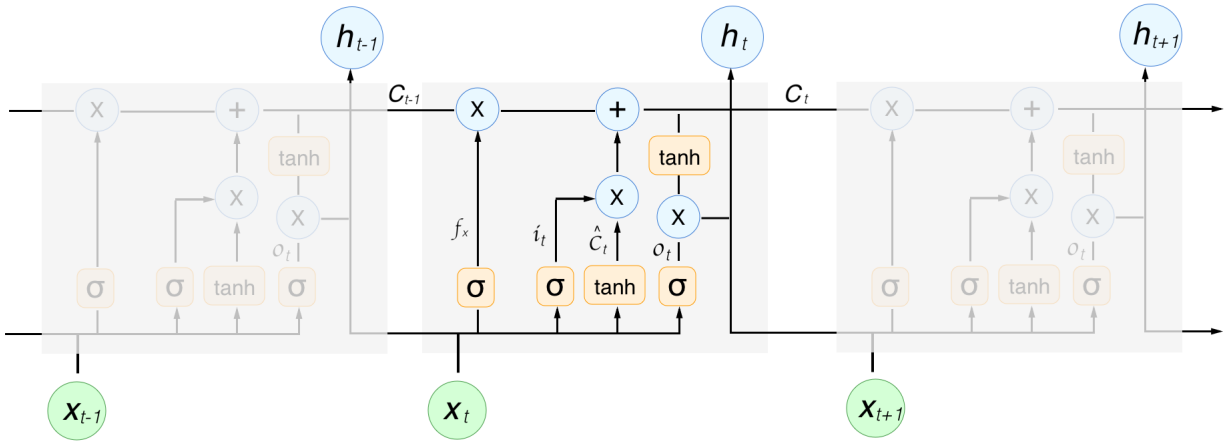

Figure 2.16: Diagram of long short term memory.

The vanilla RNN suffers from a long-term dependency problem where information cannot be maintained for a long time [58]. For example, given the sentences, "I am English. I lived in France for 10 years. I can speak two languages, French and ?", we can guess the last word is "English". However, it is difficult for a vanilla RNN to predict same because the information contained in the first sentence, "I am English", is located at the beginning of the input, far from the last word. So, long short-

term memory (LSTM) [58] is an advanced version of the RNN developed to alleviate this long-term dependency problem of the vanilla RNN.

Figure 2.16 diagrams the LSTM where symbols "+" and "x" represent element-wise summation and multiplication, respectively, and σ represents the sigmoid function. While the vanilla RNN only uses a hidden state vector h to store historical information and the network output, LSTM introduces a cell state vector C designed to store long-term information. The size of the vector C is defined based on the application and data [59]. The inference of LSTM is separated into three steps.

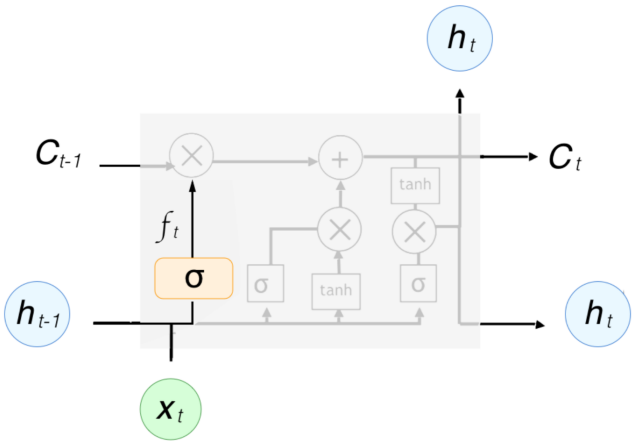

Figure 2.17: The forget gate of long short-term memory.

First, to remove information from the previous time step, as shown in Figure 2.17, LSTM applies a forget gate to remove the information from the previous cell state C_{t-1}. The forget vector f_t is calculated in Equation (2.16), where "$[a, b]$" denotes the concatenation of the two vectors a and b, and W_f and b_f are the matrix and bias, respectively, for the forget gate. Given that the values in f_t are between 0 to 1, multiplying f_t with C_{t-1} element-wisely (*i.e.*, $f_t \odot C_{t-1}$) reduces the specific values in C_{t-1} allowing the network to learn how to remove values from the previous cell state.

$$f_t = \sigma([h_{t-1}, x_t]W_f + b_f) \tag{2.16}$$

After removing information from the previous time step, the useful information of the current time step must be added to the cell state. As seen in Figure 2.18, LSTM uses an input gate to "inject" new information into the cell state by first extracting the useful information $\hat{C}_t$ from the previous hidden state h_{t-1} and input data x_t. Next, the input gate rescales the values into -1 $\sim$ 1 using a

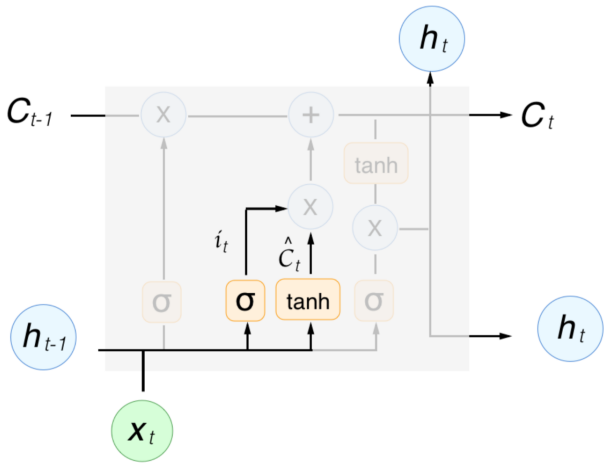

Figure 2.18: The input gate of long short-term memory.

hyperbolic tangent function, similar to the vanilla RNN. Finally, just as in the forget gate, it uses the input vector i_t with values between 0 and 1 to control the the percentage of each values in cell state to add to the next cell state. The input vector i_t and useful information vector $\hat{C}_t$ are calculated in Equation (2.17). Given f_t, i_t and $\hat{C}_t$ from the forget and input gates, as Figure 2.19 shows, the updated cell state vector C_t is obtained by Equation (2.18) where the symbol $\odot$ denotes element-wise multiplication. The updated cell state C_t then passes into the next time step, resulting in a design that better deals with the long-term dependency problem compared to vanilla RNN [58].

$$
\begin{aligned}
i_t &= \sigma([h_{t-1}, x_t]W_i + b_i) \\
\hat{C}_t &= tanh([h_{t-1}, x_t]W_C + b_C)
\end{aligned}
\tag{2.17}
$$

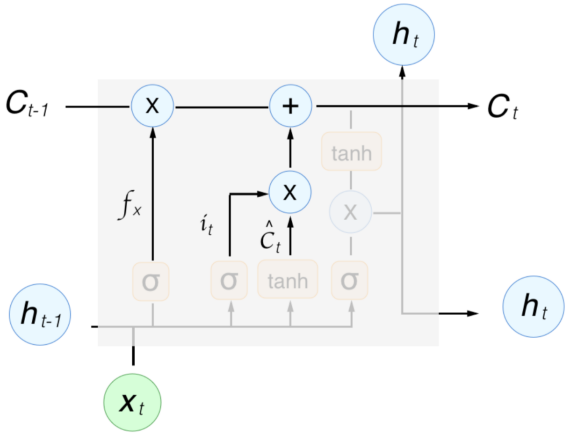

Figure 2.19: Update the cell state of long short-term memory.

$$C_t = f_t \odot C_{t-1} + i_t \odot \hat{C}_t \tag{2.18}$$

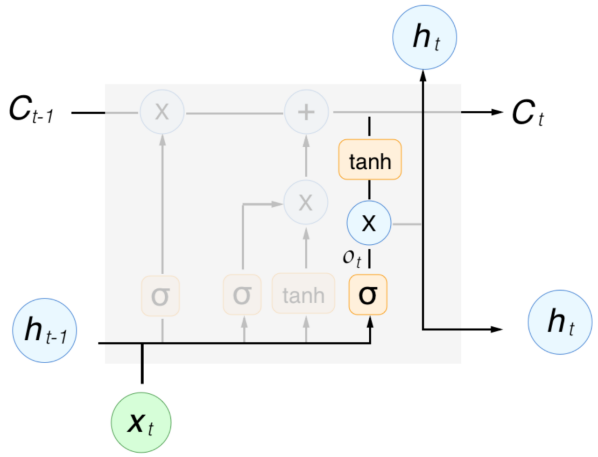

Figure 2.20: The output gate of long short-term memory.

As with vanilla RNN, a hidden state h_t is required for the output of each time step. Given the updated cell state vector C_t, the previous hidden state vector h_{t-1}, and the input data x_t, the LSTM uses an output gate function to computed the current hidden state h_t as in Equation 2.19.

$$\begin{aligned} o_t &= \sigma([h_{t-1}, x_t]W_o + b_o) \\ h_t &= o_t \odot tanh(C_t) \end{aligned} \tag{2.19}$$

Several variants of LSTM exist, including the Gate Recurrent Unit (GRU) [57]. However, Greff *et al.* [59] analysed eight LSTM variants on three representative tasks, including speech recognition, handwriting recognition, and polyphonic music modelling, and summarised the results of 5,400 experimental runs (representing 15 years of CPU time). This review suggests that none of the LSTM variants provides significant improvements to the standard LSTM, so we selected to use the standard LSTM [58] in our experiments.

2.2 Training Deep Neural Networks

In this section, we describe the details of training deep neural networks, including gradient descent, back-propagation, hyper-parameter selection, overfitting problem, and regularisation.

2.2.1 Gradient descent and error back-propagation

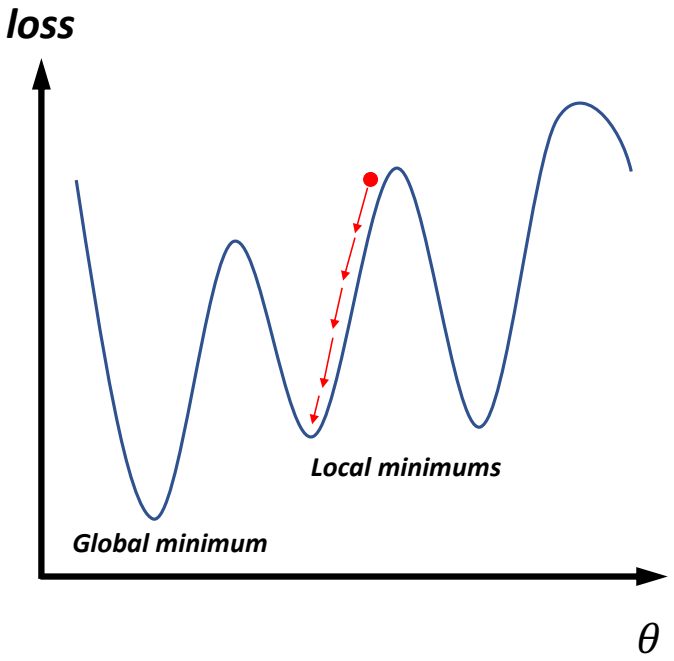

Figure 2.21: An example of gradient descent.

In deep learning, given a network and a loss function, training the network is the process to minimise the loss value $\mathcal{L}$ by updating the network parameters θ, such as the weights, biases, and filters. Gradient descent is one method to update the network parameters [2] for training. Even though there are some other optimisation methods exist, such as limited memory BFGS (L-BFGS) and conjugate gradient (CG), due to the drawbacks related to larger computation requirements, they are not often applied [2]. Therefore, we use gradient descent to optimise our networks, and include here only a review of this approach and how to update the network parameters layer-by-layer through error back-propagation [38].

Figure 2.21 illustrates a simple example of gradient descent. To train a neural network, it is initialised with random network parameters. As denoted by the red arrows, we next iteratively compute the gradient and update the parameters θ to minimise the loss. Specifically, the gradient is the first order

partial derivative of the loss $\mathcal{L}$ with respect to the parameters θ, which is written as $\frac{\partial \mathcal{L}}{\partial \theta}$. Given this gradient, the parameters are updated by $\theta := \theta - \alpha \frac{\partial \mathcal{L}}{\partial \theta}$.

The process of error back-propagation [60] computes the gradients $\frac{\partial \mathcal{L}}{\partial \theta}$ for every parameters in the network. When computing the gradient, an intermediate result $\delta = \frac{\partial \mathcal{L}}{\partial z}$ is introduced, which is the partial derivative of the loss $\mathcal{L}$ with respect to the layer's output z. Based on this intermediate result, the process next computes the partial derivative of the loss $\mathcal{L}$ with respect to every parameters $\frac{\partial \mathcal{L}}{\partial \theta}$ which is then used to update the parameter values.

The layers are indexed as $l = 1, 2, ...L$, where L is the index of the output layer, each layer has an output z^l, an intermediate result $\delta^l = \frac{\partial \mathcal{L}}{\partial z^l}$, and an activation output $a^l = f(z^l)$ (where f is the activation function). We use a MLP network with MSE loss and a sigmoid activation function to illustrate the process of error back-propagation. Given $z^l = a^{l-1}W^l + b^l$, $a^l = f(z^l) = \frac{1}{1+e^{-z^l}}$ and $\mathcal{L} = \frac{1}{2}(y - a^L)^2$, we represent the partial derivative of the activation output with respect to its output as $\frac{\partial a^l}{\partial z^l} = f'(z^l) = f(z^l)(1 - f(z^l)) = a^l(1 - a^l)$ and the partial derivative of the loss with respect to the activation output as $\frac{\delta \mathcal{L}}{\delta a^L} = (a^L - y)$. To compute the partial derivative of the loss with respect to the output layer, we apply the chain rule as follows:

- $\delta^L = \frac{\partial \mathcal{L}}{\partial z^L} = \frac{\partial \mathcal{L}}{\partial a^L} \frac{\partial a^L}{\partial z^L} = (a^L - y)(a^L(1 - a^L))$

Then, the partial derivative of the loss with respect to all other layers' outputs are computed as the following, where $l = 1, 2, ..., L - 1$.

- Given $z^{l+1} = a^l W^{l+1} + b^{l+1}$, we have $\frac{\partial z^{l+1}}{\partial z^l} = W^{l+1} f'(z^l)$
- Then $\delta^l = \frac{\partial \mathcal{L}}{\partial z^l} = \frac{\partial \mathcal{L}}{\partial z^{l+1}} \frac{\partial z^{l+1}}{\partial z^l} = \delta^{l+1} \frac{\partial z^{l+1}}{\partial z^l} = \delta^{l+1}(W^{l+1})^T \frac{\partial a^l}{\partial z^l} = \delta^{l+1}(W^{l+1})^T \odot (a^l(1 - a^l))$

The second step of the error back-propagation is to computes the partial derivative of the loss with respect to the parameters $\frac{\partial \mathcal{L}}{\partial W^l}$ and $\frac{\partial \mathcal{L}}{\partial b^l}$ of each layer based on the intermediate result δ^l.

- Given $z^l = a^{l-1}W^l + b^l$, we can have $\frac{\partial z^l}{\partial W^l} = a^{l-1}$ and $\frac{\partial z^l}{\partial b^l} = 1$
- Then $\frac{\partial \mathcal{L}}{\partial W^l} = \frac{\partial \mathcal{L}}{\partial z^l} \frac{\partial z^l}{\partial W^l} = (a^{l-1})^T \delta^l$, $\frac{\partial \mathcal{L}}{\partial b^l} = \frac{\partial \mathcal{L}}{\partial z^l} \frac{\partial z^l}{\partial b^l} = \delta^l 1 = \delta^l$

Finally, we use the $\frac{\partial \mathcal{L}}{\partial W^l}$ and $\frac{\partial \mathcal{L}}{\partial b^l}$ to update the parameters W^l and b^l as follows:

- $W^l := W^l - \alpha \frac{\partial \mathcal{L}}{\partial W^l}$
- $b^l := b^l - \alpha \frac{\partial \mathcal{L}}{\partial b^l}$

where α, also known as the learning rate, is a value to control the step size of each update. Additional information about this learning rate is provided in the next subsection.

Gradient descent updates the parameter iteratively and converges to a minimum point of the loss function as in Figure 2.21. In practice, the converged point is typically a local minimum rather than the global one. However, as deep neural networks offer a good representation capacity, the local minimums tend to be close to the global minimum [2].

2.2.2 Stochastic gradient descent and adaptive learning rate

As the training data is usually large in practice, the loss value is not computed by using all training data for each update. Otherwise, one update would require significant computational time. Instead, for each update, the modified process of stochastic gradient descent (SGD) [38] randomly selects a small number of the data from the training set. These data are called a "mini-batch", and the quantity of data in the mini-batch is called the "batch size". By updating the parameter multiple times, the mini-batches will cover the entire training set. The process of SGD is outlined in Algorithm 1:

Algorithm 1 The training process of stochastic gradient descent (SGD).

Input: parameters θ, learning rate α, number of training steps/iterations n

1: **for** $t = 0$ **to** n **do**
2: $\frac{\partial \mathcal{L}}{\partial \theta}$; compute the gradient using a random mini-batch
3: $\nabla\theta := -\alpha * \frac{\partial \mathcal{L}}{\partial \theta}$; compute the parameters update
4: $\theta := \theta + \nabla\theta$; update the parameters
5: **end for**
6: **return** θ; return the trained parameters

The learning rate controls the step size of the update and is the most important parameter in SGD. If the learning rate is too large, then the update may fail to find the minimum point. If the learning rate is too small, then the update will be slow. It is difficult to determinate the value of the learning rate, so recent studies proposed adaptive learning rates, such as Adam [61], RMSProp [62] and Adagrad [63], which speed up the training by automatically adjusting the learning rate. The principle of the adaptive learning rate is to shift to a larger step when the parameters receive small gradients and take a smaller step when receiving large gradients. Adam is the most frequently used method. Instead of using the

gradient to update the parameters directly, Adam computes the running average of the gradients and the second moments of the gradients to update the parameters as shown in Algorithm 2. The β_1 and β_2 terms are the forgetting factors, also known as momentum, for the gradients and second moments of the gradients, respectively. By default, β_1 is 0.9 and β_2 is 0.999 [61].

Algorithm 2 The training process of Adam optimisation.

Input: parameters θ, learning rate α, number of training steps/iterations n, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

$m_0 \leftarrow 0$; initialise the first order moment vector
$v_0 \leftarrow 0$; initialise the second order moment vector
for $t = 0$ **to** n **do**
  $\frac{\partial \mathcal{L}}{\partial \theta}$; compute the gradient using a random mini-batch
  $m_t \leftarrow \beta_1 * m_{t-1} + (1 - \beta_1) * \frac{\partial \mathcal{L}}{\partial \theta}$; update the first order moment vector
  $v_t \leftarrow \beta_2 * v_{t-1} + (1 - \beta_2) * (\frac{\partial \mathcal{L}}{\partial \theta})^2$; update the second order moment vector
  $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$; compute the running average of the second moments of the gradient
  $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$; compute the running average of the second moments of the gradient
  $\nabla\theta := -\alpha * \frac{\hat{m}_t}{\sqrt{v_t}+\epsilon}$; compute the parameters update
  $\theta := \theta + \nabla\theta$; update parameters
end for
return θ; return the trained parameters

Many methods exist to determinate when to stop the SGD training, such as setting a fixed number of updates and a threshold for the loss [2]. However, in practice, SGD is usually terminated after being trained for a specified number of epochs or steps [37, 44], where one epoch represents the mini-batch has looped over the entire training set.

2.2.3 Hyper-parameter selection

Validation set

In deep learning, hyper-parameters consist of the settings of the model, such as the number of layers and filters, and the settings of the training, such as the number of epochs, batch size and learning rate. These settings affect the performance of the model, and to obtain a better model, selecting these hyper-parameters appropriately is essential.

To evaluate the performance of different hyper-parameters settings, the data is usually split into training, testing, and validation sets. Then, multiple settings configurations are applied to the training set and evaluated on the validation set. Finally, the settings with the best performance are selected for evaluation on the testing set to obtain the final performance metrics.

Cross-validation

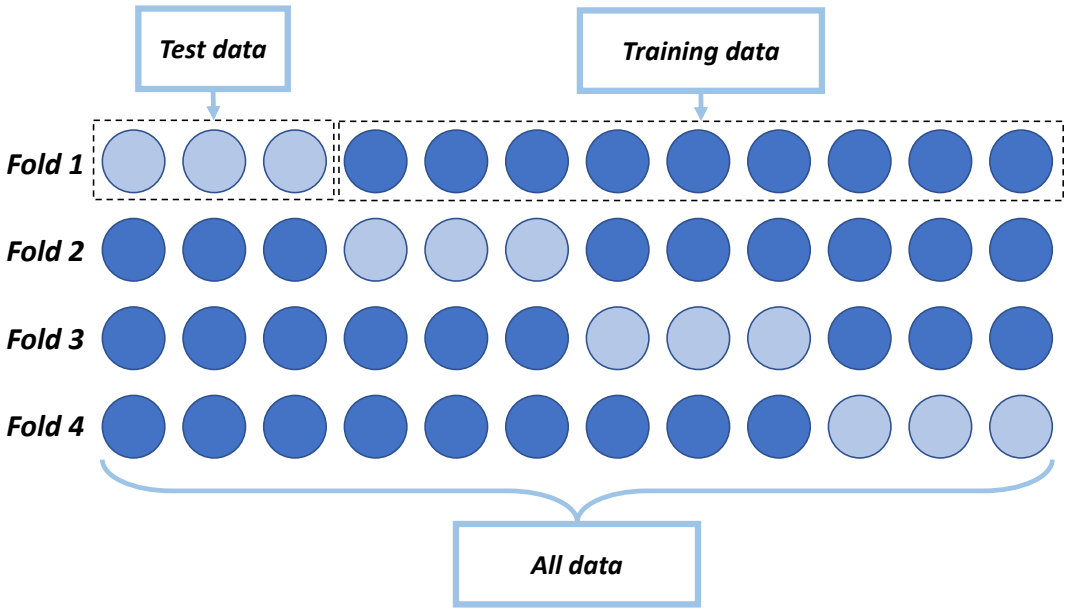

Figure 2.22: An example of cross-validation. The dark blue and light blue circles represent the training and testing data, respectively.

For small datasets, splitting the data into training, validating, and testing sets may be challenging. If the size of the training data is too small, then the performance of the model will be impacted. On the other hand, if the test data is too small, then the evaluation may not adequately reflect the performance. To evaluate small datasets, cross-validation is required in which the data is only required to be split into training and testing sets.

In cross-validation, the dataset is separated into K folds of data all with the same size. One of these folds of data is selected for validation and the remainder for training. The final evaluation result is the average over all folds. Figure 2.22 illustrates 4-fold cross-validation for 12 examples that are split into four folds with three examples providing four results by training the network four times. The mean and standard deviation of these four results represent the performance of the network and its hyper-parameters.

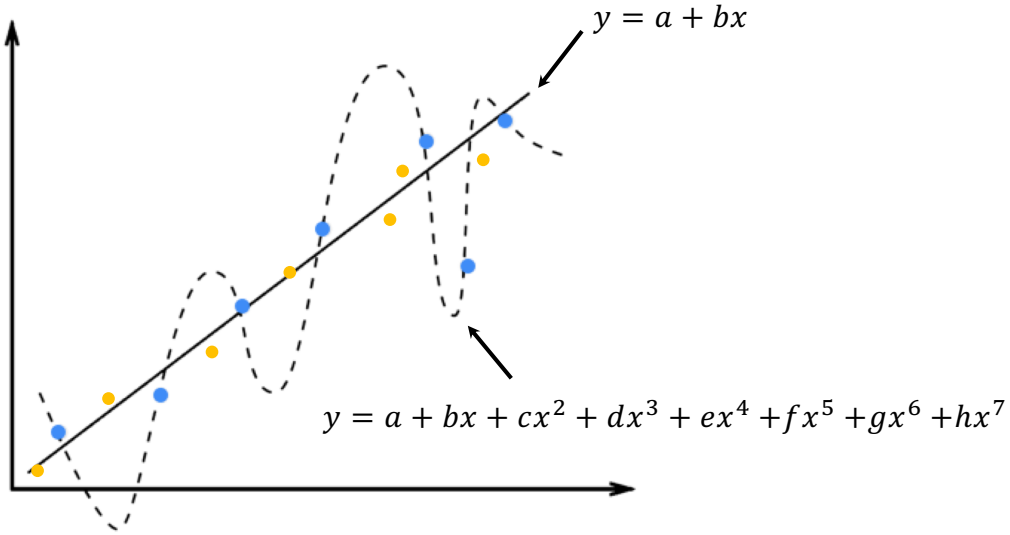

Figure 2.23: An example of overfitting. The blue dots represent the training data points, and the orange dots are the testing data points.

2.2.4 Overfitting and regularisation

Overfitting

An overfitted model from a machine learning algorithm contains excess parameters to fit the training data and results in a low loss on the training set but a high loss on the testing set. In Figure 2.23, for example, the dotted curve is a simple overfitted model that clearly fits all training data points exactly. However, it will fail to fit additional testing data points reliably. In contrast, the straight line modelled by $y = a + bx$ includes fewer parameters than the dotted curve while offering a better fit for the testing data points.

Underfitting is the opposite scenario where the model cannot fit the training data resulting in high losses for both training and testing sets. However, in practice, underfitting can be solved by using a deeper network, but solving overfitting is more challenging. The simplest way to alleviate overfitting is to use more data while training the model. In practice, it can be difficult to collect and label more training data due to data acquisition costs and labelling effort.

Weight decay

Weight decay is a simple and effective regularisation method to solve the overfitting problem, it directly introduces a regularisation term into the loss function for obtaining smaller network parameters θ. For example, as Figure 2.23 shows, if the parameters from c to h of the dotted curve have smaller values, then it will have a lower swing range that can better fit the testing data points. The loss function with the parameter norm penalty is defined as follows.

$$\mathcal{L}_{total} = \mathcal{L}(y, \hat{y}) + \lambda\Omega(\theta) \tag{2.20}$$

where $\mathcal{L}(y, \hat{y})$ is the loss function computed from a given target y and predicted results $\hat{y}$, Ω is the parameter norm penalty function and λ is a small value that controls the strength of the regularisation. Two of the most common parameter norm penalty functions are $\mathcal{L}_2 = ||W||_2^2$ and $\mathcal{L}_1 = ||W||$ norms. The network parameter values are often smaller than 1, so by using $\mathcal{L}_2$, a smaller value can result in a much smaller penalty than $\mathcal{L}_1$ (*e.g.,* $|w| > w^2$) [2]. In contrast, $\mathcal{L}_1$ can have a larger plenty than $\mathcal{L}_2$ for small values, so $\mathcal{L}_1$ has the sparse property of producing very small, even zeros, network parameters enabling the networks to perform feature selection, *i.e.,* discarding some input features by setting the corresponding parameter to zero or a very small value [2].

Dropout

Large neural networks include many parameters making it difficult to deal with the overfitting by combining the predictions from so many parameters. Dropout [64, 65] is a popular technique for addressing this problem, by preventing the large number of parameters from over co-adapting. To prevent the co-adaptation of parameters, during training, the hidden outputs are randomly set to zero, which resembles a random disconnection of the neural units from one layer to the next, as illustrated in Figure 2.24. According to error back-propagation, with a zero-valued output a, the corresponding partial derivative of the loss with respect to the layer output δ will be zero. In other words, only the remaining connected weights will be updated. Therefore, the dropout method can train may different sub-networks while allowing all of them to share the same network parameters [64]. During testing, dropout is disabled, and no output elements are set to zero. In other words, all sub-networks are

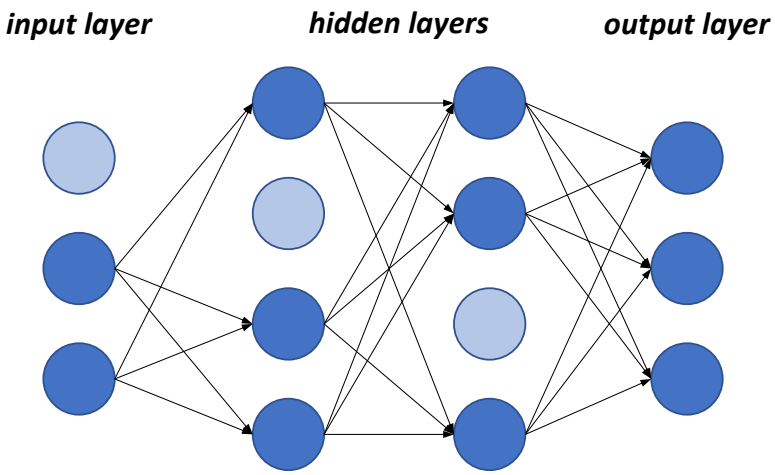

Figure 2.24: Applying dropout to a fully connected network.

used to predict the result represented as using the average from many networks as the final result (*i.e.,* ensemble learning [66]). The theoretical proof for dropout was not presented in the original publication [64], but recent studies proved its effectiveness in ensemble learning [66] and Bayesian approximation [67].

Batch normalisation

Batch normalisation [68] is the introduction of a layer that normalises the inputs to have a mean of 0 and variance of 1 and can improve the performance of a neural network and its training stability. Specifically, during training, the batch normalisation layer estimates the mean and variance of the batch input using a moving average, which updates the moving mean and variance for every iteration to be used for normalising the batch input. During testing, the moving mean and variance are fixed and applied to normalise the input.

Apart from improving performance and stability, batch normalisation provides regularisation. Similar to the dropout process that adds a random factor to the hidden values, the moving mean and variance of batch normalisation introduce randomness as they are updated in each iteration according to the random mini-batch. Therefore, the network must learn to be robust enough to deal with the variation.

Figure 2.25: An example of image data augmentation. The top-left image is the original and the others are obtained by a randomly flip, shift and zoom the original.

Other methods for avoiding overfitting

Other methods exist for preventing overfitting, such as early stopping and data augmentation. Early stopping is a simple approach to avoid overfitting that allows for the discontinuing of the training once it matches an empirical criterion, such as a threshold value of accuracy on the validating set. Data augmentation increases the number of training data by augmenting the existing training data. For example, image data can be augmented by simply flipping, rotating, shifting, and zooming. A data augmentation method that generates arbitrary but reasonable data can reduce overfitting and improve the performance [10,45,46]. Similar to an image, the audio can be augmented by adding noise or perturbation. For example, dropout can be applied into the network during the training phase while improving the performance of speech recognition [64]. Also, a recent study [69] showed that adding speech perturbations into a raw speech audio signal can improve the performance of speech recognition algorithms.

Comparing to image, it is not reasonable to directly apply augmenting transformations of raw textual data because the order of words provides specific meaning. For example, shuffling a text description "a man like dog" does not equal to "a dog like man" in terms of its meaning. Because of this challenge, there remains no common practice for text augmentation [14]. An ideal text data augmentation process would be to rephrase the sentences manually. However, due to the large workload, a common text data augmentation approach is to replace words with pre-defined synonyms, which requires domain knowledge from human [14,15]. Instead of augmenting the text explicitly in the raw data, some studies alleviate the overfitting problem by changing the text embedding [24, 64]. For example, interpolating

the text embeddings of two random texts can help to fill the gaps in the text latent space [24].

2.2.5 Transfer learning

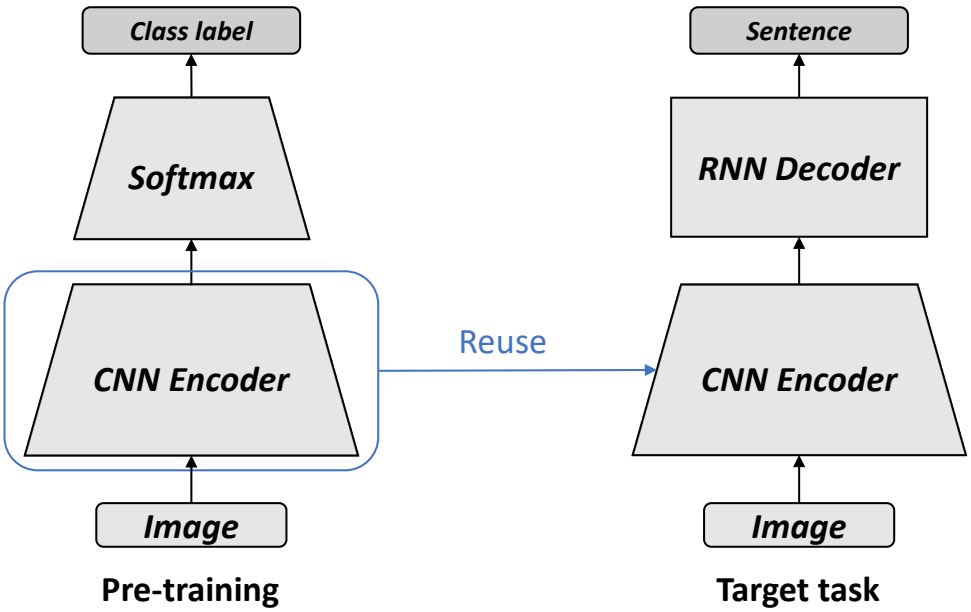

Figure 2.26: An example of transfer learning in deep learning.

Transfer learning, also known as inductive transfer [70, 71], utilises the knowledge of one task and applies it to related tasks. In deep learning, pre-trained models are widely used for improving the performance of many tasks without labelling more data [1, 34]. The idea is to train a network on a large dataset with a general task and reuse the network for another network with the target task. This transfer tends to work if the datasets include common features [70].

For example, as in Figure 2.26, Vinyals *et al.* [1] pre-trained an Inception-V3 network [48] on ImageNet [4] by performing a classification task. Then, the trained Inception-V3 network is used again as the image encoder for an image captioning task to achieve a successful result. In transfer learning, the datasets for pre-training and the target tasks can be different. For example, the objects in ImageNet usually occupy the entire image and are located in the centre. In the image captioning dataset, MSCOCO [9], the objects are relatively small and located across the image. Similarly, Zhe *et al.* [34] used a pre-trained VGG19 network [45] on ImageNet for a human pose estimation task. In pose datasets, such as MSCOCO [9] and MPII [72], the images include people a variety of scales and backgrounds. These examples demonstrate that even though the images used to pre-trained the image encoder are different from the images of the final tasks, the pre-trained encoder can still improve the

performance of target tasks [1, 34]. This success is because by training the encoder with a very large dataset, it can learn the perceptual concept and general features of different objects in the images.

Pre-trained models are also widely used in natural language processing. To encode a sentence, we can pre-train a word embedding vector on a large text dataset using Word2vec [53, 73] or GloVe [54]. The purpose of pre-training is to map words into a latent space. Then, meaningful vectors represent discrete words to improve the performance of the target task. Transfer learning is similar to how humans learn. For example, when we learn to classify text as spam, we do not need to re-learn word meanings from scratch but utilise our existing knowledge of the language. Similarly, when learning a new visual task, humans do not re-learn the concepts of edge, shape, colour, and texture from scratch.

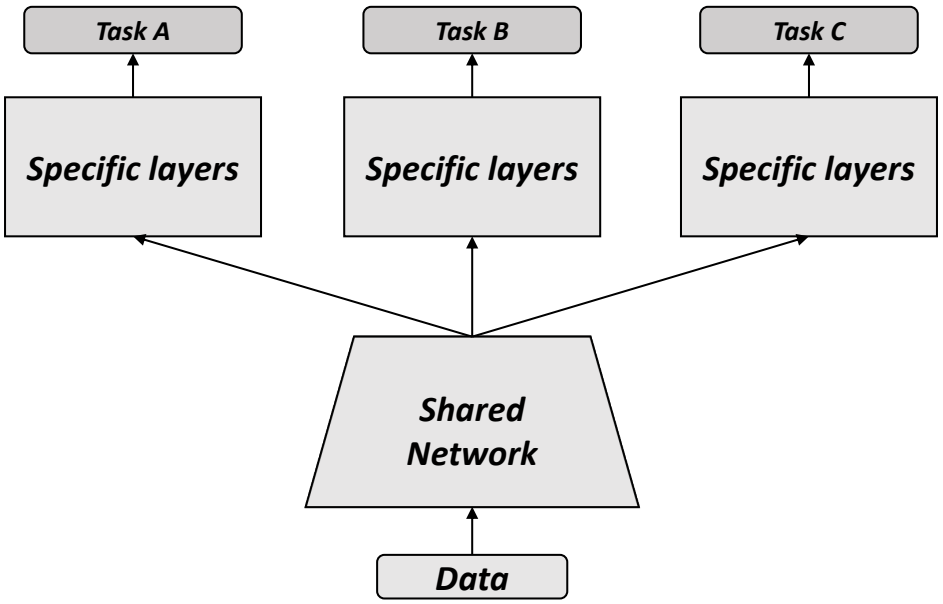

Figure 2.27: An example of multi-task learning in deep learning.

While, transfer learning is powerful and easy to apply, the definition of the knowledge that can be transferred for a learning context is not straightforward, so experimentation is typically required [70]. The transfer learning will fail if the data distribution for the pre-training and target tasks are unrelated, and only work if the pre-trained dataset is "suitable" or "appropriate" to the target context [2].

Multi-task learning is another form of transfer learning [2, 44]. Instead of pre-training a network and then reusing it for another task, the multi-task learning approach jointly trains a network with several relative tasks by sharing some part of the network, as Figure 2.27 illustrates. For example, a recent study of Mask R-CNN [44] combined object segmentation and detection together, so the features used by both tasks were shared and improved jointly to improve the score of the object detection.

2.3 Generative Adversarial Networks

By using deep neural networks, many models have been developed recently for generating data such as image and signal [19, 74]. Architectures of these models include deterministic networks [75–77], Variational Autoencoders (VAE) [18,78,79], autoregressive models [80,81] and Generative Adversarial Network (GAN) [26,82]. Among these, GAN-based methods [13,19,20,24,26] have demonstrated great success in generating high-visual quality and high-resolution images since 2015 and can be applied to solve computer vision problems [7, 83]. All of these studies are based on the vanilla GAN framework introduced in Goodfellow *et al.* [26]. In this section, we describe the vanilla GAN followed by the multi-modal problem and how to apply auxiliary information into the GAN framework.

2.3.1 Vanilla GAN

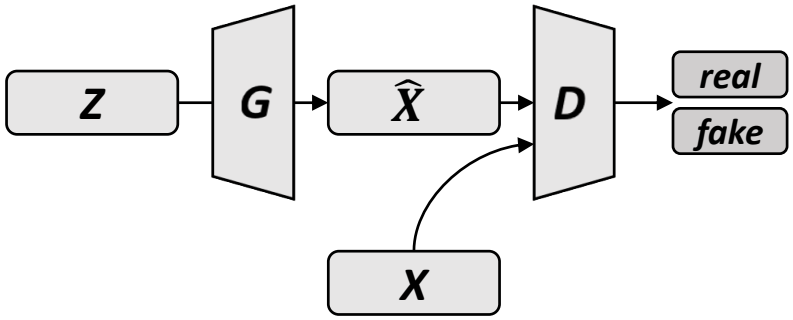

Figure 2.28: The vanilla GAN.

The vanilla GAN consists of a generator network G and a discriminator network D. The generator learns to map the latent variables (a noise vector) z to the visual data x to synthesise fake samples $\hat{x} = G(z)$. The discriminator incorporates real samples x and fake samples $\hat{x} = G(z)$ as inputs to outputs the probability ($D(x)$ or $D(G(z))$) for distinguishing real and fake images. Figure 2.28 illustrates the training process of the vanilla GAN. Specifically, the generator and discriminator are trained in a competing way, such that the generator attempts to fool the discriminator, while the discriminator attempts to distinguish real and fake samples. Through this competition, the two networks enhance one another. A minimax function models this training process over the value function $V(D, G)$ as follow:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}}[logD(x)] + \mathbb{E}_{z\sim p_z}[log(1-D(G(z)))] \tag{2.21}$$

where the latent variable z is sampled from a prior distribution p_z, and the real samples x are sampled from the real data distribution p_{data}. The p_z is usually a uniform distribution or standard normal distribution with a mean of zero and variance of one [19]. The corresponding loss functions is written as follow:

$$\begin{aligned} \mathcal{L}_D = &\quad \mathbb{E}_{x\sim p_{data}}[logD(x)] + \mathbb{E}_{z\sim p_z}[log(1-D(G(z)))] \\ \mathcal{L}_G = &\quad \mathbb{E}_{z\sim p_z}[logD(G(z))] \end{aligned} \tag{2.22}$$

From the theoretical perspective, GAN training learns a function that can map a distribution to another (*e.g.,* from p_z to p_{data}). The syntheses process is like a sampling process where given a latent vector z, it finds the corresponding data from p_{data} [26]. The evaluation of generative models remains an open problem, especially for those tasks that do not have specific outputs [29, 84, 85]. For this reason, most of GAN studies incorporate a human evaluation method [25, 86, 87].

2.3.2 Deep convolutional generative adversarial networks

Based on the vanilla GAN, Radford *et al.* [19] proposed a learning method to synthesise 64×64 images from latent variables by utilising the power of a Deep Convolutional GAN (DCGAN). The experiment demonstrated that the latent variables reflect certain features in the images. By interpolating the input noise vector, the features of the images can be linearly changed, *e.g.,* by controlling the window size of a bathroom or the smiles on faces. The loss functions of the discriminator D and generator G are the same as the vanilla GAN [26].

Training a GAN is difficult and can lead to a failed training. Some methods [19, 88, 89] were developed for improving the stability of the adversarial training process, and DCGAN proposes a set of empirical tricks on the network architecture and training method to enable the GANs to be trained stably

Figure 2.29: Example results of randomly generated bedroom images using DCGAN. Image is from [19].

for synthesising images in most settings. The generator is a deconvolutional decoder [43] and the discriminator is a convolutional encoder. The following tricks help stabilise adversarial training:

- For the generator, use deconvolutions with a stride of 2 for the up-sampling operation.
- For the discriminator, use convolutions with a stride of 2 for the down-sampling operation.
- Use batch normalisation [68] in both generator and discriminator (as detailed in Section 2.2.4).
- Use rectifier (ReLU) [33] in the generator and leaky ReLU [36] with a slope of 0.2 in the discriminator.
- Use an initial learning rate of 0.0002 and Adam optimisation [61] with a momentum β_1 of 0.5 for changing the learning rate adaptively.

Figure 2.29 shows example results from randomly generating bedroom images using DCGAN. The image visual quality and resolution have significant improvement compared with the vanilla GAN [26]. To stabilise GAN training, the above tricks are incorporated for all GANs in this thesis.

2.3.3 Conditional GAN

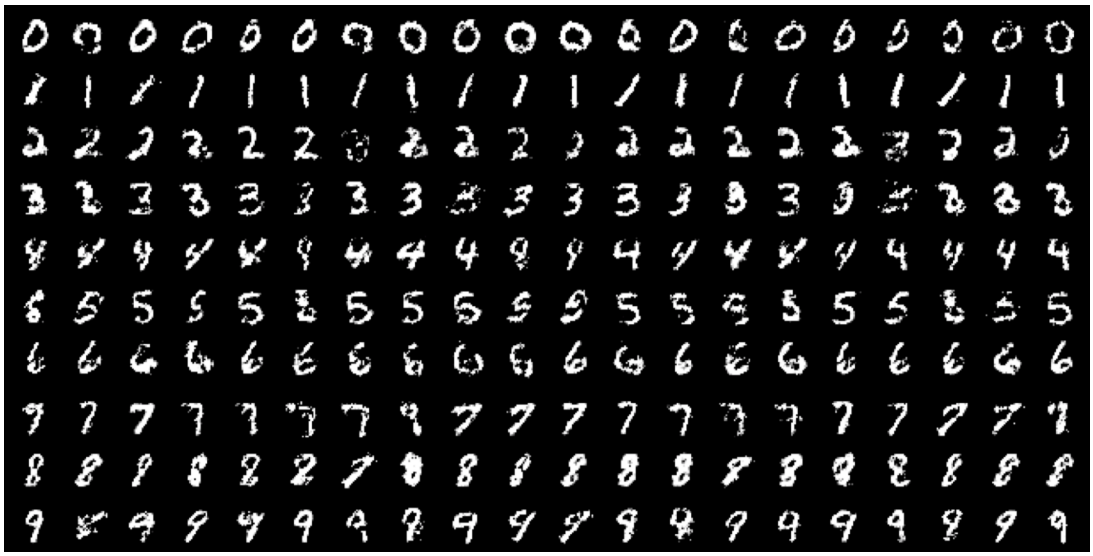

Figure 2.30: Example results of generating digit images using a vanilla conditional GAN where each row is conditioned on one label. Image is taken from [17].

A vanilla GAN can only synthesise data randomly, which limits its practical applications. There exist many studies built upon the vanilla GAN with additional enhancing conditions that allow for synthesised images to be not only plausible but also match the constraints imposed by the conditions. For example, a GAN can be conditioned on discrete class labels [16, 17, 90–92]. Also, many other works synthesised images by conditioning a GAN on images for tasks such as domain transfer [13, 93–95], image editing under constraints imposed by users [96, 97], image super-resolution [28, 29], image synthesis from surface normal maps [98], image inpainting [30], and style transfer [28, 99]. These conditions can also be of a text format in which images are generated to match given text descriptions. Reed *et al.* [24] proposed an end-to-end deep neural architecture based on the conditional GAN framework, which successfully generated realistic images (64×64) from natural language descriptions. Apart from image synthesis, some recent studies use GAN to synthesise different types of data for a variety of applications, such as synthesising signals for speech enhancement [100] and high-level features for domain adaptation [83].

These types of studies can be categorised as conditional GAN [17] and are considered as mapping a mixture distribution to another. A vanilla conditional GAN [17] is extended from the vanilla GAN by feeding auxiliary information y into both the discriminator and generator as additional input. Then, for the generator, the latent variables z and y are combined into a joint latent representation, and the data samples x and y are input to the discriminator. The vanilla conditional GAN can be formulated

into the following two-players minmax game.

$$\min_G \max_D V(D,G) = \mathbb{E}_{x,y\sim p_{data}}[logD(x,y)] + \mathbb{E}_{z\sim p_z}[log(1-D(G(z,y),y))] \tag{2.23}$$

In this thesis, all GAN models are conditional GAN that conditioned on different auxiliary information, such as image, text description or both. However, our conditional GANs are more complex than the vanilla conditional GAN described above.

Chapter 3

Efficient Text-to-Image Synthesis

Efficient use of data is essential for deep learning, especially for the generative tasks because training data are difficult to obtain and label. Synthesising an image conditioned on the text description (*i.e.,* text-to-image synthesis) is a typical generative task that suffers from limited training data. Ideally, each image should have an infinite number of annotated text descriptions. In this chapter, we explore how to improve the visual quality of the image in text-to-image synthesis without manually labelling more training data, providing controlled generated results using object attribute information.

3.0.1 Introduction

Since 2015, throught the improvement of the vanilla generative adversarial network (GAN) [26], GAN-based methods [19, 20, 101] have demonstrated great success in generating high-quality visual images. For example, DCGAN [19] successfully synthesises 64×64 RGB images by extending the vanilla GAN with a deep convolutional network. Following this, the Progressive GAN [20] successfully synthesise 1024×1024 face images. However, the vanilla GAN framework [19, 26] only randomly synthesise images making it difficult to apply to many real-world applications because these usually require specific outputs. To control the synthesis, the auxiliary classifier GAN (ACGAN) [16] synthesises images for specific classes, such as generating numerical characters conditioned on a given digit or a different object in the ImageNet dataset conditioning on class labels to achieve class-based image synthesis. However, the class label only provides the information for the object categories, such as bird, flower or car, and it cannot provide other object attribute information, such as the colour of the object.

Compared with the class label, a text description can contain more information, including both the object category and the object attribute information, such as the background scene and human action. Therefore, learning image synthesis conditioned on a text description is more controllable and practical compared to a random or class-based image synthesis approach like DCGAN or ACGAN, respectively. To achieve this, Reed *et al.* [24] introduced a GAN-based text-to-image synthesis method called GAN-CLS [24] that successfully synthesised 64×64 images conditioned on a provided text description. The experiment demonstrated promising results on the Caltech-200 birds [102] and Oxford-102 flowers [103] datasets, which only contain one kind of object, either a bird or flower are centrally aligned in the images. However, GAN-CLS has difficulty in synthesising a visually clear image on more complex datasets, such as the MSCOCO [9]. Unlike the bird and flower datasets, MSCOCO contains multiple object categories, such as a person and a car or aeroplane. Also, the size, location, and background of the objects are random in MSCOCO making it more challenging compared to the bird and flower datasets. A dataset containing a single category is called a single-category dataset (*i.e.,* the Caltech-200 birds and Oxford-102 flowers) and one that contains multiple categories is a multi-category dataset, MSCOCO is a typically multi-category dataset.

An essential challenge of text-to-image synthesis is that labelled text descriptions are limited in the training set. For example, MSCOCO [9] includes only five labelled text descriptions for each image, which makes it more likely that training will not cover sufficient variances to synthesise an image. A problem with several solutions instead of one unique solution is known as a multi-modal problem [26]. Image synthesis conditioned on text descriptions is highly multi-modal as many possible text descriptions that can correctly match an existing image. Specifically, given an image, we can consider infinite ways to describe it. For example, given an image with some fruits on a table, we might describe it as "some fruits on a table", "this is a table with some fruits", "a table", "bananas and apples on a table" or "here are some fruits". Each of these phrases can describe the given image correctly. Similarly, provided a text description, we could consider infinite ways to create an image. For example, for the text, "people are playing tennis", we can draw people on an image with different backgrounds, lighting, clothes, poses, and sizes. Therefore, this generative task is different from typical machine learning applications, such as image classification and object detection, that has only one input and one ground truth output. The conversion between text and image should could have infinite results.

An ideal dataset should include infinite labelled text descriptions per image to ensure the model is sufficiently general to produce the best results. When using more text descriptions to train the

text-to-image model, the training data can include more information and cover more possible cases of the data, which will increase the robustness and generalisation of the model [14]. Therefore, the visual quality of synthesised images can be improved when inputting unseen text descriptions to the model [24]. However, it is expensive to label many text descriptions manually for each image in the dataset. Fortunately, in recent years, image captioning algorithms that synthesise text descriptions conditioned on images have achieved success using deep learning [1,104,105]. The state-of-the-art deep learning-based image captioning methods [1,104,105] outperform the previous methods [106,107] with comparable performance to humans on the MSCOCO dataset [1] [1].

To generate more text descriptions, we propose the use of a pre-trained image captioning module to synthesise a large number of text descriptions for each image in the dataset and use these text descriptions to train the text-to-image synthesis. This approach includes more information and covers more variance of the data, which increase the robustness and generalisation of the model. This method does not require additional domain knowledge from human, such as manually labelled more training data. Moreover, by using the pre-trained image captioning module, we successfully train the text-to-image generator on image datasets that do not have labelled text descriptions. Specifically, we pre-train the image captioning module on MSCOCO, then use the image captioning module to synthesise text descriptions for the images in MPII [72]. We use these synthesised text descriptions to train the text-to-image generator. This training process utilises the knowledge of MSCOCO and applies it to MPII, which is a process of transfer learning [71]. To the best of our knowledge, the proposed method is the first work that can perform transfer learning on text-to-image synthesis for unlabelled dataset. Because the synthesised texts are generated from images, we named the proposed method as image-to-text-to-image (I2T2I).

3.1 Related Works

3.1.1 Text-to-image synthesis

In 2015, the study in DRAW [108] proposed a model based on the recurrent neural network (RNN) for image synthesis. This method uses multiple steps to synthesise images and approaches the synthesis as a sequential problem that allows the iterative construction of the images. It successfully synthesises

[1] `http://cocodataset.org`

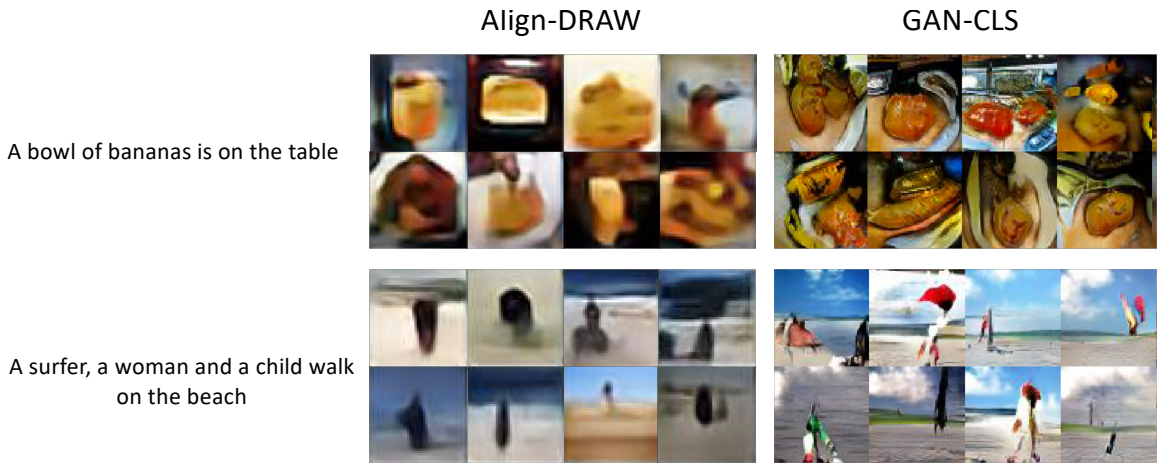

Figure 3.1: Example results of align-DRAW and GAN-CLS on the MSCOCO dataset. The image are from [24].

images for simple numeric character datasets, such as MNIST, which contains back-and-white handwritten numeric characters and the Street View House Numbers dataset [109] that contains coloured street view house numbers. Extending from DRAW, Mansimov *et al.* [110] proposed an RNN model called align-DRAW for text-to-image synthesis. Even though, align-DRAW successfully synthesises images conditioned on text description, it tends to output blurry images [110]. Adding an additional sharpening post-processing step can generate clearer edges, but it is not an ideal method to improve the image visual quality [110].

To utilise the power of GAN to synthesise an image, to solve the multi-modal problem between the text and an image, and to synthesise images conditioned on a given text description. Reed *et al.* proposed a GAN-based text-to-image synthesis method called GAN-CLS [24]. This method outperforms the previous RNN-based text-to-image synthesis methods [108,110]. The authors demonstrated the result on two single-category datasets: Caltech-200 birds [102] and Oxford-102 flowers [103], and on a multi-category dataset, MSCOCO [9]. The example results of align-DRAW and GAN-CLS can be found in Figure 3.1. The results show that the GAN-CLS method can synthesise more object details. Therefore, we used GAN-CLS as the baseline in our research. The details on the GAN-CLS method are as follows.

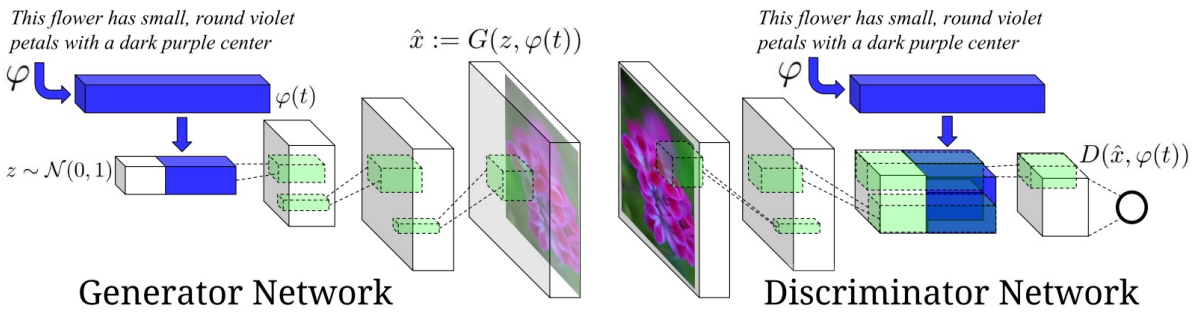

Figure 3.2: Network architecture for text-to-image synthesis using GAN. The image is from [24].

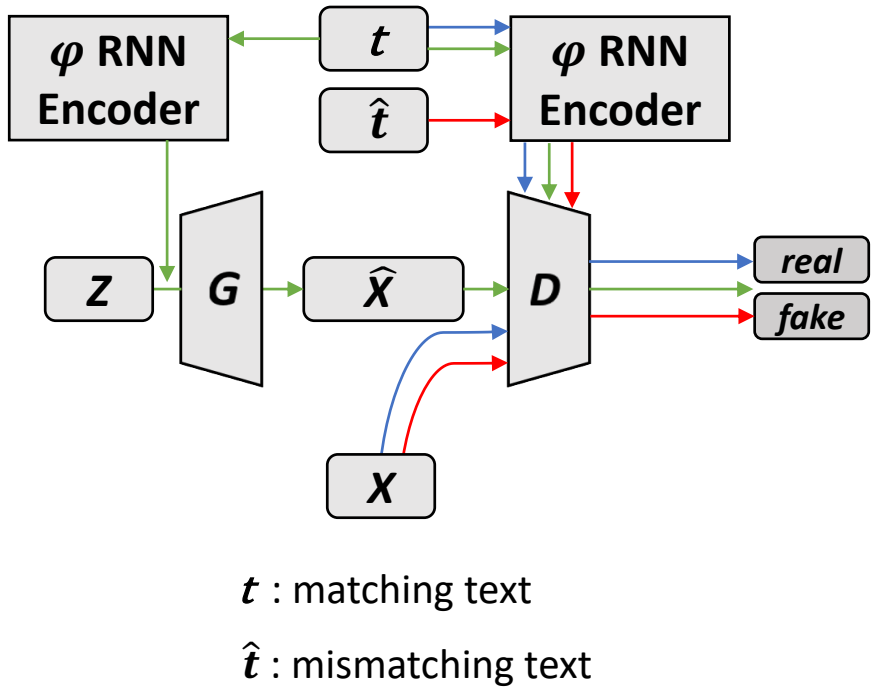

Figure 3.3: Diagram of GAN-based text-to-image synthesis training. Blue lines: Feed the real image with its matching text description to the discriminator. The discriminator is updated by considering it a positive sample. Red lines: Feed the real image with a mismatched text description to the discriminator. The discriminator is updated by considering it a negative sample. Green lines: Feed a text description to the generator and then feed the corresponding synthesised image to the discriminator. The discriminator is updated by considering it a negative sample, while the generator is updated by impelling the discriminator to consider the synthesised image a positive sample.

3.1.2 GAN-based text-to-image synthesis

Figure 3.2 shows the network architecture of GAN-CLS. It consists of one generator G, one discriminator D, and one text encoder φ. First of all, the authors pre-trained the RNN text encoder φ that

can map the text description into an embedding vector. Then, the generator outputs images based on two inputs: 1) the text embedding $\varphi(t)$ and 2) the normal distribution z. The discriminator outputs the probability of its input image to be real based on two inputs: 1) the input image and 2) the text embedding $\varphi(t)$.

Based on this architecture, Figure 3.3 shows the training process of GAN-CLS. As the green lines indicate, the generator learns to synthesise fake image $\hat{x}$ conditioned on a given text description t, and then feed the fake image into the discriminator to fool the discriminator to consider the fake image a positive sample. On the other hand, as the red and blue lines indicate, the discriminator not only needs to classify the real image x with the matching text description t as a positive sample and to synthesise image $\hat{x}$ with a matching text description t as the negative sample, but also needs to classify the real image with a mismatched text description $\hat{t}$ as a negative sample. By learning to classify mismatched text, the discriminator can give a stronger signal to the generator about the text information, which helps the generator synthesise a better visual quality image that can match the text description [24]. In summary, the loss functions of the generator and discriminator are as follow:

$$\begin{aligned}\mathcal{L}_D &= \mathbb{E}_{(x,t)\sim p_{data}}[logD(x,\varphi(t))] \\ &\quad + \mathbb{E}_{(x,\hat{t})\sim p_{data}}[log(1-D(x,\varphi(\hat{t})))] \\ &\quad + \mathbb{E}_{(z,t)\sim p_{data},p_z}[log(1-D(G(z,\varphi(t)),\varphi(t)))] \\ \mathcal{L}_G &= \mathbb{E}_{(z,t)\sim p_{data},p_z}[log(D(G(z,\varphi(t)),\varphi(t)))]\end{aligned} \tag{3.1}$$

where the first term of $\mathcal{L}_D$ is to optimise discriminator D for classifying the real images and matching text descriptions (*i.e.,* x and t) as the positive samples. The other two terms are for classifying the real images with mismatched text descriptions (*i.e.,* x and $\hat{t}$) and the synthesised images with matching text descriptions (*i.e.,* $\hat{x} = G(z,\varphi(t))$ and t) as the negative samples, respectively. The $\mathcal{L}_G$ is to optimise generator G for competing with the final term of $\mathcal{L}_D$. The $\mathcal{L}_G$ lets generator G attempt to fool the discriminator to classify the synthesised images as positive samples.

3.1.3 Filling the gaps in the text space for text-to-image synthesis

Extended from GAN-CLS, to alleviate the problem of the limited number of labelled text descriptions for single-category datasets (*e.g.,* the bird and flower datasets), Reed *et al.* [24] proposed a method

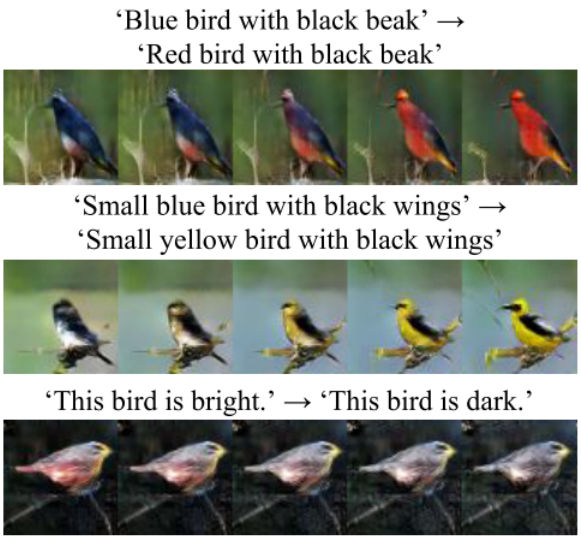

Figure 3.4: Generating bird images by interpolating two texts on text space. Image is taken from [24].

called GAN-INT-CLS to generate text embeddings by interpolating two random text embeddings from the output of the RNN text encoder φ, and used these interpolated text embeddings to train the generator. More specifically, the authors trained the generator using the interpolated text embeddings, which were obtained by averaging two randomly selected text descriptions from the training set. In other words, they use the text embedding from Equation (3.2) to replace the the text embedding $\varphi(t)$ of G in Equation (3.1).

$$\varphi(t) = \frac{\varphi(t_1) + \varphi(t_2)}{2} \tag{3.2}$$

where t_1 and t_2 are two text descriptions randomly selected from the training set.

Given N number of text descriptions, we can have N data points in the text space, where the text space is the universal set of text descriptions that contains all possible text embeddings. Using the random averaged text embeddings, we can have N^2 number of data points in the text space. This method allows the generator to learns to fill in the gaps in the text space in between data points [24]. For example, Figure 3.4 shows the results of generating images by interpolating two text descriptions, where each interpolated image represents one data point on the text space. On the top of Figure 3.4, we can see that given two text descriptions "blue bird with black beak" and "red bird with black beak", the averaged embeddings between them could represented "blue and red bird with black beak". The experimental results show that using averaged text embeddings can generate more diverse results on

bird and flower datasets compared with GAN-CLS. The contribution of GAN-INT-CLS is to propose an implicit way to fill in the gaps on the text space, increasing the density of the text space without requiring someone to explicitly label more text descriptions for the images.

However, GAN-INT-CLS assumes that two randomly selected text descriptions can be interpolated. This may work well for the bird and flower datasets where all the text descriptions describe the same kind of object. The text descriptions of the bird and flower datasets mainly describe the colours and textures of different parts of the objects [24]. For example, the interpolation of the text embeddings for "a white bird" and "a red bird" could be "a light red bird". In practice, for multi-category datasets, different text descriptions may describe totally different objects, so the complicated information, such as different object categories, quantities, actions and backgrounds cannot be interpolated. For example, it is unreasonable to interpolate the text embeddings of "two peoples are walking on the grass" and "a table with many plates of food and drinks" because we cannot find a reasonable meaning between them. Thus, it is unsuitable to train a multi-category dataset using the averaged text embeddings [24].

In addition, as the averaged text embeddings do not have ground true images, the averaged text embeddings can only be used for training the generator but not the discriminator. Thus, the discriminator cannot directly learn to distinguish the averaged text embeddings [24]. To address these problems, instead of using the averaged text embeddings of two randomly selected text descriptions, our proposed method synthesises the actual text descriptions for all training images and uses them to train both the generator and discriminator simultaneously. By doing this, all text embeddings are explainable, and our human evaluation shows that the visual quality of the synthesised images is improved.

3.1.4 Image captioning

Image captioning generates a text description for a given image. It can be considered the inverse process of text-to-image synthesis. Many methods [106,107] have considered the image captioning task to be several sub-problems and have solved each of them separately. For example, one researcher [106] first detected the objects in an image and classified the attributes of each object, and then used the object and attribute information to generate a sentence to describe the image.

In recent years, image captioning has had great success using deep learning [1, 104, 105]. Instead of splitting the image captioning into several sub-problems, these techniques are built on a single model,

which consists of a CNN encoder and an RNN decoder. The model encodes the image into an image embedding using CNN and decodes the image embedding into a text description using RNN.

By optimising the CNN and RNN jointly, the deep learning methods outperform the previous methods [106,107] and even have comparable performance with humans on the MSCOCO dataset [1]. More information about the ranking of image captioning methods can be found in the MSCOCO captioning leaderboard [2].

3.2 Methods

The proposed method consists of one image captioning module, one RNN text encoder, one generator, and one discriminator. The image captioning module is used to generate text descriptions for the image in the training set. These text descriptions are used in the training of the other modules. The RNN text encoder is used to encode the text description into a text embedding vector, which is fed into both the generator and discriminator. The generator is used to the synthesise image conditioned on the input text embedding vector. The discriminator is used to classify whether the input image appear plausible and whether it matches with the input text description. In this section, we describe how to train each of these modules and why we use an image captioning module to generate text descriptions to train the generator.

3.2.1 Image captioning module

To have high-quality synthesised text descriptions, we use the state-of-the-art deep learning-based image captioning model [1] as our image captioning module. To encode images into image embeddings, following the verified model architecture described in [1], the state-of-the-art Inception V3 network [48] is adopted as the image encoder E, as the left-hand side of Figure 3.5 shows. The convolutional output is further encoded into a fixed-length vector of 512 values using a fully connected layer. The image embedding is also known as image feature representation, which contains the visual information of the input image [111]. The image encoder is pre-trained on the ImageNet [4] classification task. Given the image embedding, to generate the text description, in the first step, the image feature representation is fed into an long short-term memory (LSTM) decoder, which has two layers with 512

[2] `http://cocodataset.org`

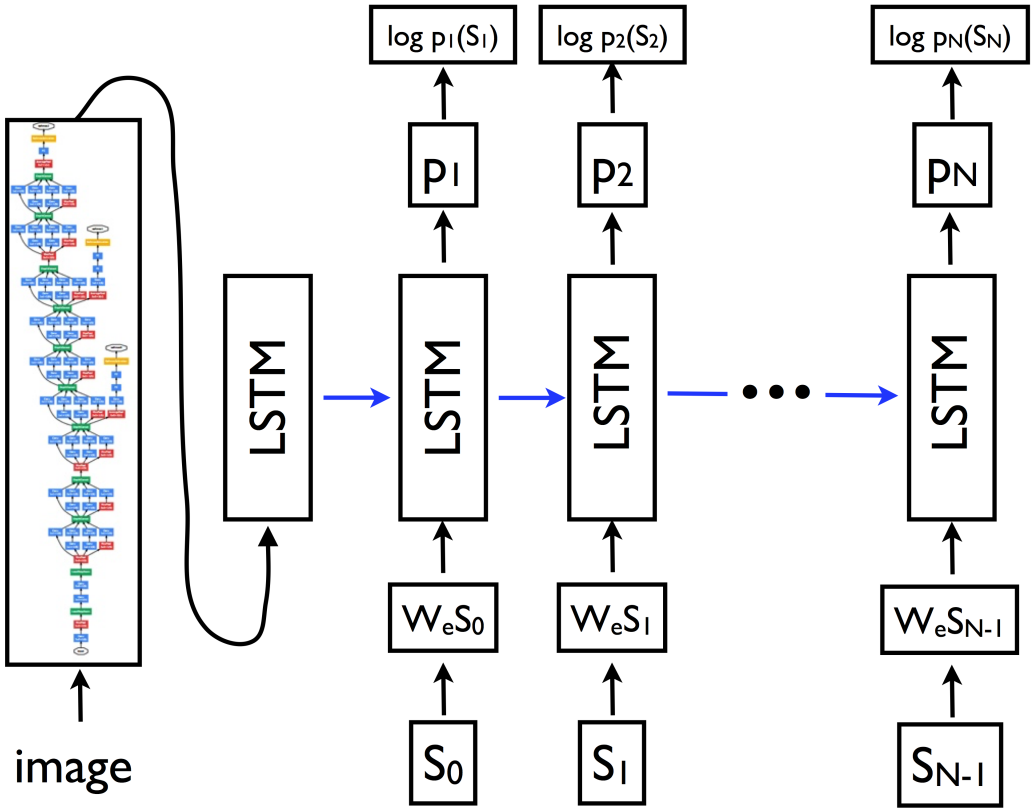

Figure 3.5: Architecture of image captioning module. The Image is from [1].

hidden units [1]. Next, the hidden and cell states will be passed into the next step as the left blue arrow shows in Figure 3.5. In the second step, given the initialised hidden and cell states for the LSTM decoder, a start token (*i.e.,* a unique ID to represent the start of a sentence) is fed into a word embedding layer with an embedding size of 512, and the embedding output will be further fed into the LSTM decoder to obtain the hidden state. Finally a softmax output layer outputs the probabilities of each word in the vocabulary.

To train the image captioning module, we follow the verified training process in [1], which achieved the state-of-the-art performance. In the first step, as the image encoder is pre-trained on ImageNet, we fix the parameters of the image encoder and optimise the LSTM decoder only. For regularisation, dropout in keeping with a probability of 70% is adopted between the two LSTM layers. Following Vinyals *et al.* [1], we use a batch size of 64 and train the model for 1,000,000 million iterations. The initial learning rate is 2 and decays by 0.5 at 500,000 iterations. The standard stochastic gradient descent (SGD) is used as the optimiser. In the second step, the entire model, including both the image encoder and LSTM decoder, are optimised together, which can slightly further improve the image captioning module [1]. Following [1], a learning rate of 0.0005 is used, and update the model

is updated with another 1,000,000 million iterations with same batch size and optimisation method. In addition, during training, following [1], images are first resized to 346 × 346, and then randomly cropped into 299 × 299. Note that, in MSCOCO, the object and the pertinent information would not be located at the edge of the images, so random cropping would not loss the text-related information in the image [1, 9]. In addition, we randomly change the brightness, contrast, saturation, and hue of images for data augmentation [1]. To stabilise the training, images pixel values are rescaled to -1 ∼ 1 [1, 19].

For feed-forward propagation (*i.e.,* inferencing), similar to the training phase, to obtain the initialised hidden h_0 and cell states c_0 of the LSTM decoder, the image encoder E encodes the image onto a vector with the length of 512 and then feeds the vector into the LSTM decoder, as Equation (3.3) shows. After that, to obtain the first word, s_1, of the sentence, a special start token s_0 is feed into the model, as Equation (3.4) shows. Moreover, the probabilities p_1 of all words in the vocabulary are computed, as Equation (3.5) shows. Given the probabilities p_1 of all words, to obtain the first word s_1, we can select the word with the highest probability as the predicted word. For later steps, we feed the output word to the next step and obtain the next output word in the same way. Note that, we stop to feed the word to the next step when the output word is the end token of the sentence.

However, using the word with the highest probability will lead to only one text description that can be generated from one image. In contrast, we can use top k sampling ($k > 1$), in which, we first select the k words with the highest probabilities, and then sample one word from these words according to their probabilities. The word with higher probability would more easily to be chosen. Selecting the word with the highest probability can be considered to be setting k to 1 in top k sampling. Overall, the text description, $t = (s_1, ..., si)$ for i starts from 1, can be generated by iterating the recurrence relation defined from Equations (3.3) to (3.6):

$$h_0, c_0 = LSTM(\theta, E(x)) \tag{3.3}$$

$$h_i, c_i = LSTM(h_{i-1}, c_{i-1}, We(s_{i-1})) \tag{3.4}$$

$$p_i = softmax(h_i W_{ho} + b_o) \tag{3.5}$$

$$s_i = sample(p_i, k) \tag{3.6}$$

where θ is the initial hidden and cell states of LSTM, which should be all zero values. In addition, E is the image encoder, W_e is the word embedding matrix, and W_{ho} and b_o are the weight matrix and bias vector for the output fully connected layer. Given that MSCOCO has 11,519 words (including the start and end tokens), the probability vector p_i has 11,519 values.

3.2.2 Textual data augmentation via image captioning

Figure 3.6: Examples of synthesised text descriptions from the image captioning module on the MSCOCO dataset.

A multi-modal problem exists between the image and text description, while in practice, a limited

number of labelled text descriptions exist for each image in the training set [9]. Due to the multi-modal problem between the image and text, it is important to fill in the gaps in the text space as described in Section 3.1.3. One text description could have many synonymous text descriptions. For example, "a man riding a surf board on a wave" is similar to "a person on a surf board riding a wave" and "a surfer riding a large wave in the ocean". In addition, "five bananas and a bottle of wine" is similar to "a brunch of bananas and a bottle of wine" and "wine and bananas". However, it is expensive to label more text descriptions manually. Fortunately, the softmax output of RNN-based image captioning can exploit the uncertainty to generate synonymous texts. By setting k to be greater than 1 in Equation (3.6), a large number of text descriptions can be generated from a single image. In this study, we set $k = 3$. The probability of a text description t given an image x can be defined as shown in Equation (3.7):

$$p(t|x) = \prod_{i=1}^{n} p(s_i|x, s_0, ..., s_{i-1}) \tag{3.7}$$

In practice, as we use the same images that the image captioning is pre-trained on, to synthesise text descriptions for these images. The synthesised text descriptions will contain the original text annotations of the images.

Training a model with a large number of text descriptions can have more data points in the text space. In order to visualise it, t-SNE [55] is a technique to visualise high-dimensional data (*e.g.,* text embedding) by giving data points a location in a two or three-dimensional space, where the similar data points will have shorter distances. Figure 3.7 visualises the text embeddings of the text descriptions from both the dataset and image captioning module. Each marker represents one data point (*i.e.,* one text description). Markers with different colours are associated with different images on the left-hand side. The circle markers indicate the data points of 20 synthesised text descriptions from the image captioning module. The cross markers indicate the data points of 5 text annotations of each images. We can see that the synthesised text descriptions are located near the labelled text descriptions. With more synthesised text descriptions being used, an image can have more data points in the text space and increase its coverage density.

By using more text descriptions to train the model will include more information and cover more variance of the data, increasing the robustness and generalisation of the model [2, 14]. In addition,

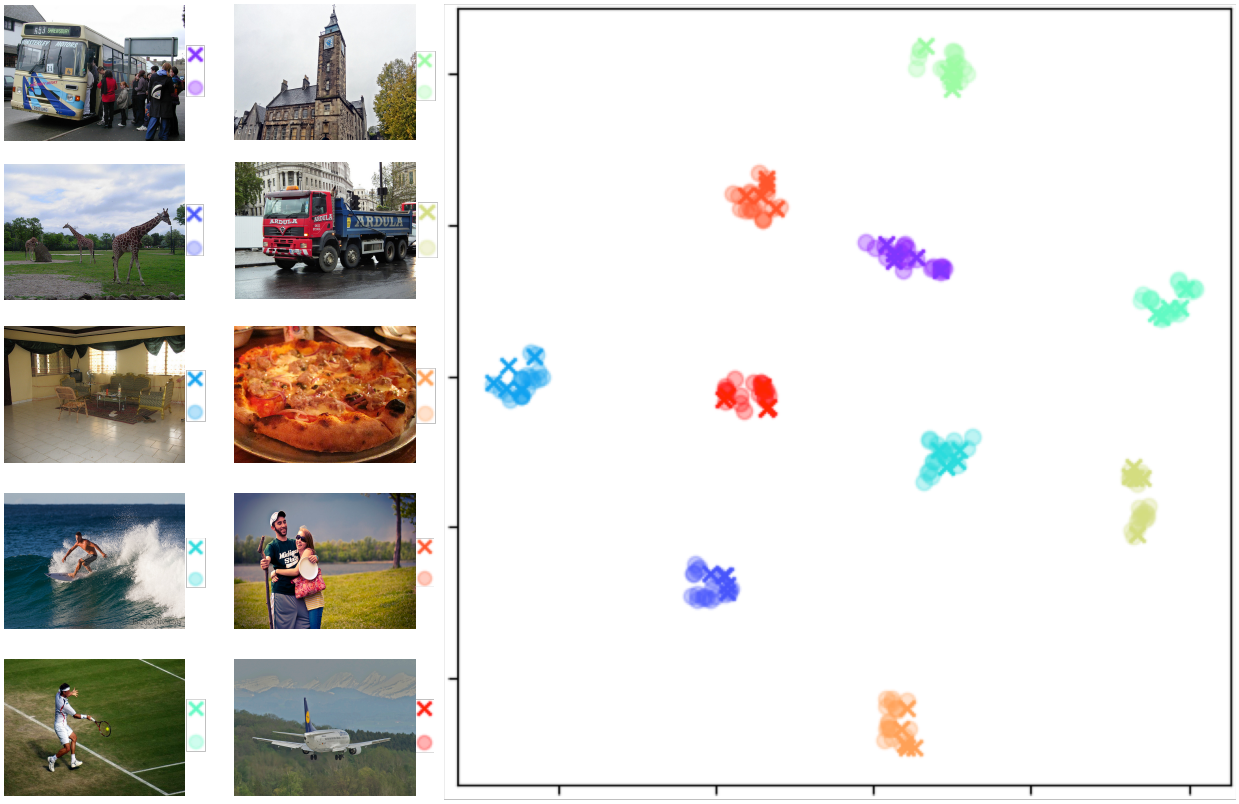

Figure 3.7: Visualisation of text space using t-SNE. Cross markers: the text descriptions from dataset; Circle markers: the synthesised text descriptions.

by using the state-of-the-art image captioning method [1], our method does not require extra domain knowledge from humans, no extra information, such as the pre-defined synonyms, grammar, or manually labelled data, need to be provided. This process can be considered the textual data augmentation to explicitly fill in the gaps in the text space.

Text encoder

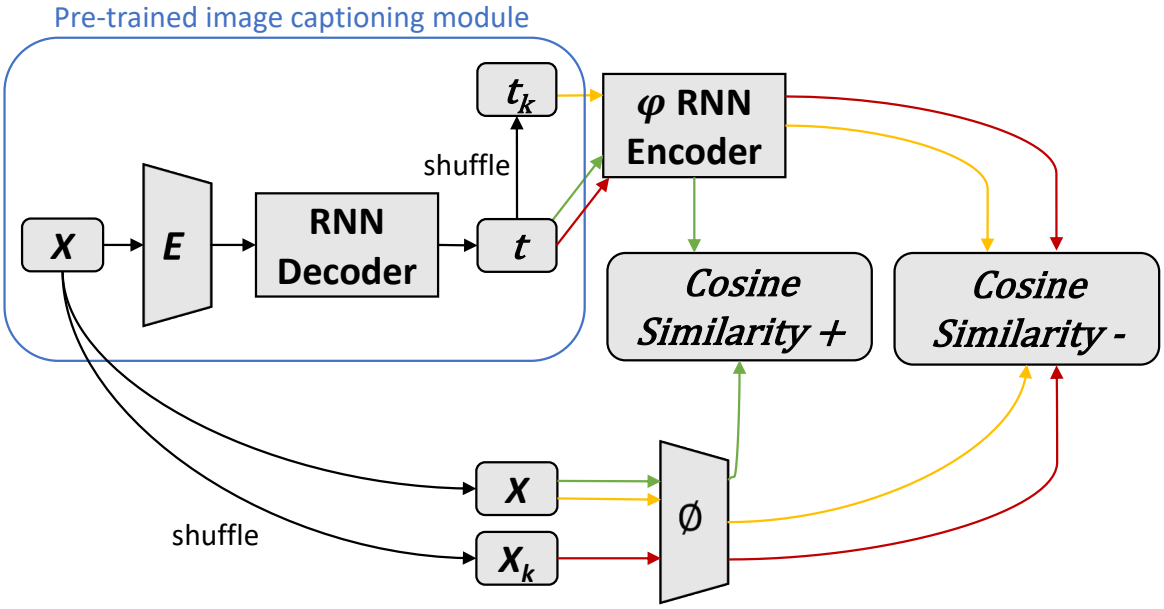

Figure 3.8: Training process of the text encoder. Given a batch of images x, the image captioning module synthesises a batch of text descriptions t associated with the images x. Then, we shuffle the batch of texts t to obtain a batch of mismatched texts t_k, so that each text cannot be associated with its matching image. Similarly, shuffle the images x to obtain a batch of mismatched images x_k, so that each image cannot be associated with its matching text. The RNN encoder φ and the image encoder ϕ are used to encode the text descriptions and images into two vectors with the same length. To train the RNN encoder φ, we follow the verified process of [52]. Green lines: Maximise the cosine similarity between the image x and its matching text description t. Orange lines: Minimise the cosine similarity between the image x and its mismatched text description t_k. Red lines: Minimise the cosine similarity between the mismatched image x_k and text description t.

To learn the text-to-image synthesis, the first step is to learn an RNN text encoder that can encode the text description into an embedding. Considerable work has been done to pre-train the word embedding and RNN encoder [51,53,54]. As an application for images and texts, we adopt the approach of Kiros *et al.* [52], which was specifically designed for text-and-image applications, where images and texts can be mapped into the same latent space by using a CNN image encoder and RNN text encoder [52]. Given a batch of images x (the number of images equals to the batch size), the image captioning module can generate a batch of text descriptions t that match with the input images one-to-one. After that,

we can obtain a batch of mismatched text descriptions t_k by shuffling the order of the batch of text descriptions t, so that each image cannot be associated with its matching text description t. Note that, like with the other methods for training the text encoder [24, 52], we are not modifying any text descriptions. The mismatched texts are obtained by shuffling the order of the given text descriptions. We can also shuffle the order of a batch of images x to obtain a batch of mismatched images x_k [3], so that each text description cannot be associated with its matching image.

As Figure 3.8 shows, to train the RNN encoder and obtain the embeddings of image and text, we first use an Inception V3 network ϕ and an LSTM network φ as the image and text encoders, respectively. Following Kiros *et al.* [52], we use the pair-wise ranking loss [52] between image and text embeddings, specifically designed for text-and-image applications, to train our RNN encoder. The loss function is defined in Equation 3.8. The training is to find a joint latent space for images and text embeddings, in which the embeddings between the matching image-and-text pairs have higher cosine similarity compared with the mismatched pairs.

$$\begin{aligned}\mathcal{L}_{\varphi} = & \sum_{x}\sum_{k} max\{0, \alpha - s(\phi(x), \varphi(t)) + s(\phi(x), \varphi(t_k))\} \\ & + \sum_{t}\sum_{k} max\{0, \alpha - s(\phi(x), \varphi(t)) + s(\phi(x_k), \varphi(t))\}\end{aligned} \tag{3.8}$$

where s denotes the cosine similarity function of two embedded vectors (*e.g.,* $\phi(x)$ and $\varphi(t)$), and x_k and t_k are the mismatched images and texts. The α is a small margin value, which is set to 0.2 [52]. More specifically, the cosine similarity is a way to measure the similarity between two vectors, varying from -1 to 1 (*e.g.,* two identical vectors have a similarity of 1, while two completely independent vectors have a similarity of 0, and two diametrically opposed vectors have a similarity of -1). Minimising the term of $-s(\phi(x), \varphi(t))$ can increase the cosine similarity of the matching image-and-text pairs. Moreover, minimising $s(\phi(x), \varphi(t_k))$ and $s(\phi(x_k), \varphi(t))$ can reduce the cosine similarity of the mismatched image-and-text pairs [52]. The max operation in the equation means that, if the value on the right-hand side of max is less than zero, we stop to update the parameters of the image and text encoders. The positive margin value α is the threshold value that controls when to stop the updating (*e.g.,* a small α means it is easier to stop the update, and vice verse). In our experiment, we

[3] This is because $\hat{x}$ represents the synthesised images in this thesis. We use x_k instead of $\hat{x}$ here. For consistency, we use t_k instead of $\hat{t}$ to represent the mismatched texts here.

follow the verified setting in [52] to set this value to 0.2.

In terms of network architecture, the LSTM encoder φ has a hidden size of 256 and an output vector with a length of 256. To compute the similarity, the image encoder ϕ needs to encode the image to a vector with the same length as the LSTM encoder. Therefore, after encoding the image using Inception V3, we apply a fully connected layer to make the convolutional output a dimension of 256. In addition, similar to the image captioning module, we use pre-trained parameters to initialise the Inception V3 network.

3.2.3 Text-to-image module

For image generation, we combine image captioning, the text encoder, and the state-of-the-art GAN-CLS, as Figure 3.9 shows. The embedding output of the RNN encoder φ is concatenated into the input noise of the generator and the hidden layer of the discriminator [24]. Most importantly, during training, both matching and mismatched texts are synthesised at the beginning of every iteration. Then, the generator and discriminator use these new texts to learn image synthesis (shown in Figure 3.9). As the texts are generated in every iteration, the RNN encoder is trained synchronously with the generator and discriminator. In addition, following the verified training process of image captioning [1] and GAN-CLS [24], our image data augmentation includes random left and right (*i.e.,* horizontal) flipping with a probability of 50% and random cropping to 299 $\times$ 299 from 346 $\times$ 346 before resizing them to 64 $\times$ 64.

To have a qualitative comparison with the original GAN-CLS described in Reed *et al.* [24], we use the same network architecture, except the RNN encoder. More specifically, the architectures of the generator and discriminator follow the DCGAN design principle to stabilise the GAN training [19]. The text embedding $\varphi(t)$ is first encoded to a vector with 128 values using a fully connected layer followed by the leaky ReLU with a slope of 0.2. Then, to generate the images, before feeding the text embedding into the generator, the text embedding is first concatenated with a noise vector z with 100 values, making it a vector of 228 values As the 2D convolutional operation can only be applied to 3D volumes, to use the 2D CNN for image generation, the concatenated vector is first mapped to a higher dimensional vector of 25,088 values using a fully connected layer, and then the vector is reshaped into a volume (3D tensor) with a size of 4 $\times$ 4 $\times$ 1568 followed by a batch normalisation layer [19, 24].

Three convolutional layers are used to further encode the volume in order to mix the information of

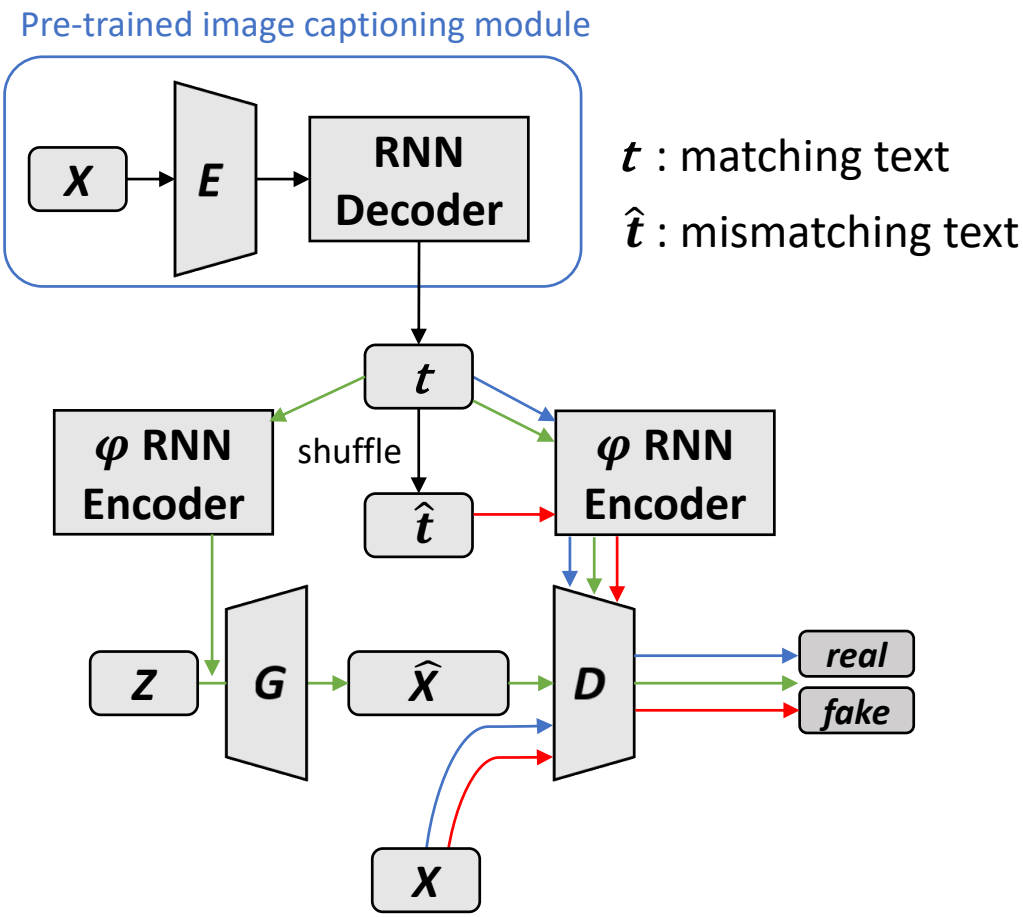

Figure 3.9: Training process of the text-to-image generator. Black lines: Given the pre-trained image captioning module, generating a batch of matching texts t associated with the input images x one-by-one, and obtain a batch of mismatched texts $\hat{t}$ so that they cannot be associated with the input images one-by-one. Blue lines: The discriminator learns to classify real images x and matching texts t as the positive samples. Red lines: The discriminator learns to classify real images x and mismatched texts $\hat{t}$ as the negative samples. Green lines: The generator learns to synthesise plausible images $\hat{x}$ by learning to fool the discriminator to consider the synthesised images and input texts to be positive samples.

the latent noise vector and text embedding. The first convolutional layer uses a filter size of 1×1, and the others use a filter size of 3×3. All layers use a stride of 1×1. Batch normalisation with ReLU activation is applied to every layers. After that, four deconvolutional layers decode the volume to the image with a size of $64 \times 64 \times 3$. All layers use a filter size of 4×4 and a stride of 2×2 (*i.e.,* all deconvolutional layers can double the height and width of the volume by a factor of 2). In addition, the batch normalisation with ReLU activation is applied to the first three deconvolutional layers. Meanwhile, as the final deconvolutional layer outputs the images, the hyperbolic tangent (tanh) is applied without using batch normalisation. To further improve the performance, three extra

convolutional layers that have the same architecture as the previous convolutional layers, are added between the first and second deconvolutional layers [24].

For the discriminator, following the DCGAN design principle [19], the input image is first encoded to a volume of $4 \times 4 \times 1{,}568$ by using four convolutional layers. All layers have a filter size of 4×4 and a stride of 2×2 (*i.e.,* all convolutional layers can halve the height and width of the volume by a factor of 2). Batch normalisation and leaky ReLU activation with slope of 0.2 are applied to every convolutional layer. After that, to further encode the information, a residual block with three convolutional layers is applied. The first convolutional layer uses a filter size of 1×1, and the others use a filter size of 3×3. All layers use a stride of 1×1. In addition, the batch normalisation and leaky ReLU activation with a slope of 0.2 are applied to every layer in the residual block. To put the text information into the discriminator, similar to the generator, the text embedding is first encoded into a vector of 128 using a fully connected layer followed by the leaky ReLU with a slope of 0.2. After that, to concatenate the text embedding with the convolutional output of the residual block, the vector of 128 values is duplicated into a volume of $4 \times 4 \times 128$ by expanding it to the height and width directions (*i.e.,* by copying it 16 times). The concatenated volume of the text embedding and the convolutional output contain the information on both the text and image. Finally, a convolutional layer with a filter size of 3, stride of 1×1 and sigmoid activation, encodes the concatenated volume into a single value that can represent the probability that an image is real.

For training the RNN text encoder, a learning rate of 0.0001 and the Adam optimisation [61] with a momentum of 0.5 are adapted [52]. For training the text-to-image generator, following DCGAN and GAN-CLS [19, 24] to stabilise the GAN training, we use an initial learning rate of 0.0002 and the Adam optimisation with a momentum of 0.5. Both learning rates of the RNN encoder and generator are halved every 100 epochs. We use a batch size of 64 and train the model for 600 epochs. The first training step of image captioning takes two weeks on an a Nvidia GTX 980 GPU. The second training step takes an extra three weeks. The training of text-to-image synthesis takes 16 days on the same GPU. Our code is implemented using TensorLayer 1.3.11 [112] and TensorFlow 1.0.0 [113]. Code can be found in the Appendix.

Algorithm 3 illustrates the training process step by step. Given a pre-trained image captioning module $im2txt$ and a batch of images x, we can first obtain the matching and mismatched text descriptions for the image x, and then encode the text descriptions into the text embedding. We denote h as the embeddings of matching texts and $\hat{h}$ as the embeddings of mismatched texts. Then, the text

Algorithm 3 Training algorithm for the proposed Text-to-Image-to-Text (I2T2I) method.

Input: Pre-trained image captioning module $im2txt$, training images x, normal distribution z, number of iterations n

for $i = 1$ **to** n **do**
  $t, \hat{t} \leftarrow im2txt(x)$ obtain a batch of matching and mismatched texts
  $h \leftarrow \varphi(t)$ encode matching text
  $\hat{h} \leftarrow \varphi(\hat{t})$ encode mismatched text
  $\mathcal{L}_\varphi / \delta\varphi$ compute the gradient of the text encoder according to Equation (3.8)
  $\varphi \leftarrow \varphi - \alpha\delta\mathcal{L}_\varphi / \delta\varphi$ update the text encoder
  $\hat{x} \leftarrow G(z, h)$ synthesise the image
  $s_r \leftarrow D(x, h)$ real image, matching text
  $s_w \leftarrow D(x, \hat{h})$ real image, mismatched text
  $s_f \leftarrow D(\hat{x}, h)$ synthesised image, matching text
  $\mathcal{L}_D \leftarrow log(s_r) + (log(1 - s_w) + log(1 - s_f))/2$
  $D \leftarrow D - \alpha\delta\mathcal{L}_D / \delta D$ update the discriminator
  $\mathcal{L}_G \leftarrow log(s_f)$
  $G \leftarrow G - \alpha\delta\mathcal{L}_G / \delta G$ update the generator
end for
return G, φ

encoder φ can be updated by following Equation (3.8). For text-to-image synthesis, the generator G synthesises image $\hat{x}$ conditioned on a random normal distribution z and the text embedding h. Given the synthesised image $\hat{x}$, we can obtain the outputs of discriminator D, where s_r is the output when inputting a real image x with a matching text description t. In addition, s_w is the output when inputting a real image x with a mismatched text description $\hat{t}$. Moreover, s_f is the output when inputting the synthesised image $\hat{x}$ with the text description t. The discriminator D learns to discriminate the real/fake image and text matching by learning to output s_w and s_f to be 0 and s_r to be 1. Therefore, the generator G can learn to synthesise image $\hat{x}$ that matches the text description t by making discriminator D outputs s_f as 1. At the end of one iteration, generator G and distributor D update according to their gradients.

3.3 Evaluation

3.3.1 Datasets

Two datasets were used in our experiments. We evaluate our proposed method, I2T2I, by comparing it with GAN-CLS [24]. We first train the image captioning module on MSCOCO [9], and then train two text-to-image generators on MSCOCO and MPII [72], respectively:

- MSCOCO [9] is a multi-purpose dataset that has labels for object detection, image segmentation, image captioning, and human pose estimation. For image captioning, it contains 82,783 training and 40,504 validation images, each of which is annotated with five text descriptions from different annotators.
- The MPII Human Pose dataset [72] is a state-of-the-art benchmark dataset for human pose estimation. It has 25,925 images covering 410 different human activities, but no labelled text descriptions are provided.

3.3.2 Results on MSCOCO

To evaluate our method, we compared I2T2I and GAN-CLS on the MSCOCO dataset. To have a fair comparison, in both methods, the generator and discriminator used the same architectures, I2T2I can be considered to extend GAN-CLS by using more text descriptions from the image captioning module. The results on the validation set can be compared in Figures 3.10 and 3.11. The results of GAN-CLS are taken from the supplementary material and paper by Reed *et al.* [24].

The quantitative evaluation for machine learning tasks that do not have ground truth results is still an open question in machine learning [2, 101]. To address this, human evaluation is commonly used in GAN research [24, 29, 101]. We used a human ranking method to quantitatively compare GAN-CLS and I2T2I. To directly compare the MSCOCO results, we selected 10 texts from the supplementary material and paper by Reed *et al.* [24]. As 54% of the images in MSCOCO contain a "person", we randomly selected five texts with person and another five texts related to other categories.

We spread volunteer recruiting advertisements in Imperial College London. To prevent bias, we select 25 volunteers who do not know our work. Moreover, the images of different methods were shuffled by Fisher-Yates method [114], so that the participants would not know which image belonged to which method. Every method synthesised six images for every text. Then we asked the volunteers to rank the images, as 0 for the best and 1 for the worst, based on the following three criteria.

- Which method appears like real image?
- Which method more matches the text description better?
- Which method has more diverse results?

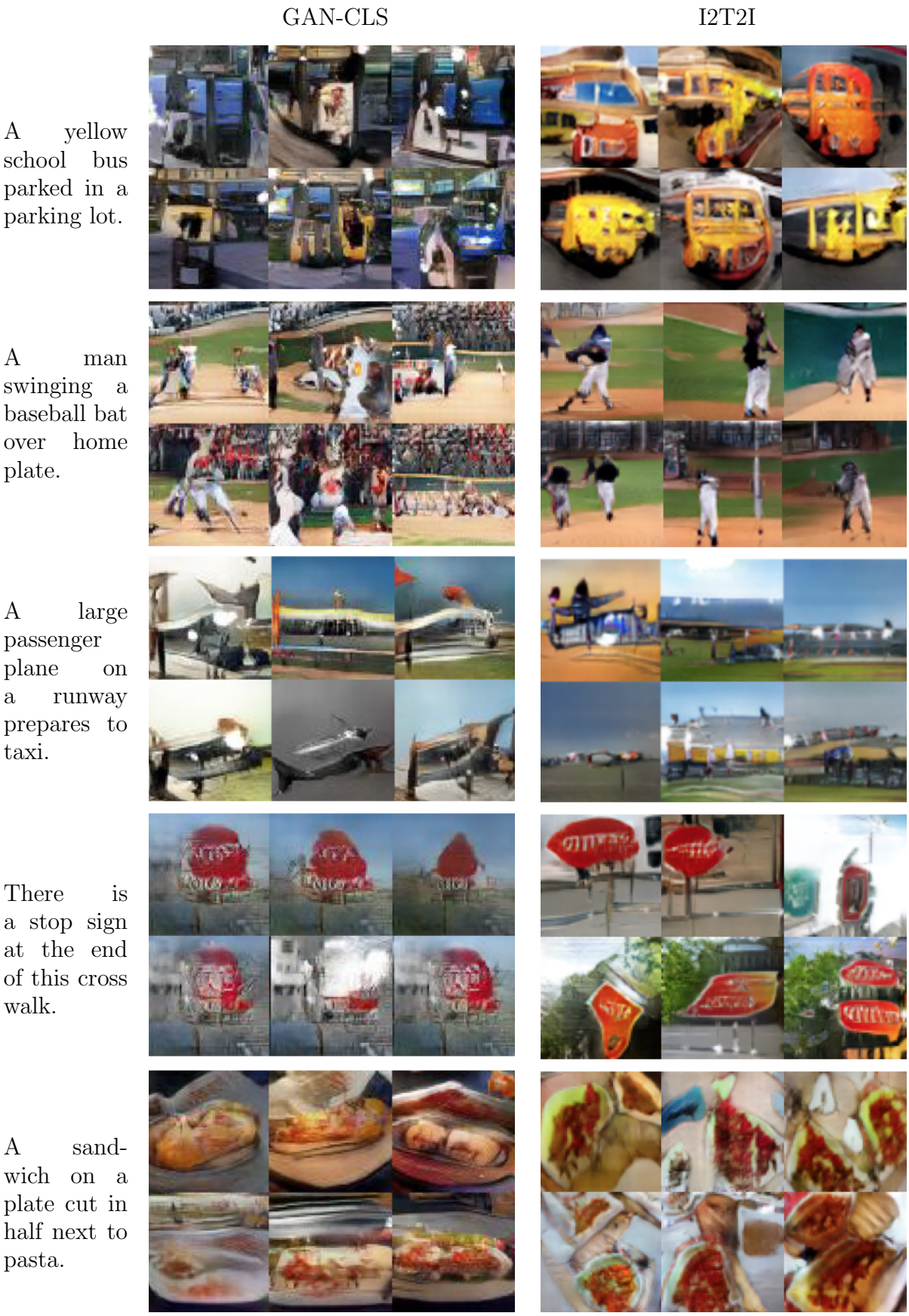

Figure 3.10: Comparing text-to-image synthesis with and without textual data augmentation. The GAN-CLS results are from [24].

In the first criterion, we only consider the image quality without taking the text description into account. The second criterion considers the text description, not only the object to be described but also the action and background if available. Similar with the first criterion, the third criterion also considers the image only, but assesses the diversity of the six images. Greater diversity indicates more difference in details of the objects, shapes, scenes, and backgrounds.

Criteria	**GAN-CLS**	**I2T2I**
Reality	0.644	0.356
Text Matching	0.572	0.428
Diversity	0.476	0.524

Table 3.1: Human evaluation of I2T2I and GAN-CLS on MSCOCO. Volunteers rank the methods as 0 for the best and 1 for the worst. A smaller score represents better performance.

Table 3.1 shows the scores of the three criteria, where smaller scores represent better performance. The results show that the proposed method can synthesise a more realistic image compared to GAN-CLS. In addition, our method can slightly improve the text matching. This is partly due to the improvement in the image quality, as more details on the image could make it more closer to the text description (*e.g.,* in the third row of Figure 3.11, our method has a clear "wave", which makes it more closer to the description of "riding a surfboard on a wave"). However, our method slightly sacrifices image diversity, as GAN-CLS has a better score (*e.g.,* in the second row of Figure 3.10, our method cannot generate various backgrounds and poses compared to GAN-CLS). The reason may be that we are using the same network architecture as GAN-CLS, but using more text descriptions to train the model requires a generator with more capacity to disentangle the text embedding.

3.3.3 Results on MPII for transfer learning

Criteria	**GAN-CLS**	**I2T2I**	**I2T2I-Transfer**
Reality	1.284	0.912	0.804
Text Matching	1.076	0.976	0.948
Diversity	1.024	1.136	0.840

Table 3.2: Human evaluation of I2T2I and GAN-CLS on MSCOCO, and I2T2I pre-trains image captioning on MSCOCO and trains the rest on MPII.

As 54% of the images in MSCOCO and 100% of the images in the MPII dataset are related to human activities, it is possible to pre-train the image captioning module on MSCOCO and then train the text-to-image generator on MPII. In other words, we utilise the knowledge of the image captioning task on MSCOCO and apply it to text-to-image synthesis on MPII. This process can be considered

Figure 3.11: Comparison between synthesised images using GAN-CLS and our I2T2I on the MSCOCO validation set. The GAN-CLS results are from [24].

transfer learning, as described in Section 2.2.5.

For this transfer learning task, we compared the results of three methods: 1) GAN-CLS trained on MSCOCO, 2) I2T2I trained on MSCOCO, and 3) I2T2I trained on MSCOCO and MPII for transfer learning, namely "I2T2I-Transfer". As MPII only contains images of humans, we only compared the results of human activities. We used the same quantitative evaluation method as the MSCOCO experiment. Because we need to compared three methods, the ranking starts from 0 to 2 (*i.e.,* 0 for the best, 1 for fair, 2 for the worst).

Table 3.2 shows the comparison of these three methods. It is clear that both I2T2I and I2T2I-Transfer have a better score than GAN-CLS in terms of image reality and text matching. For image reality

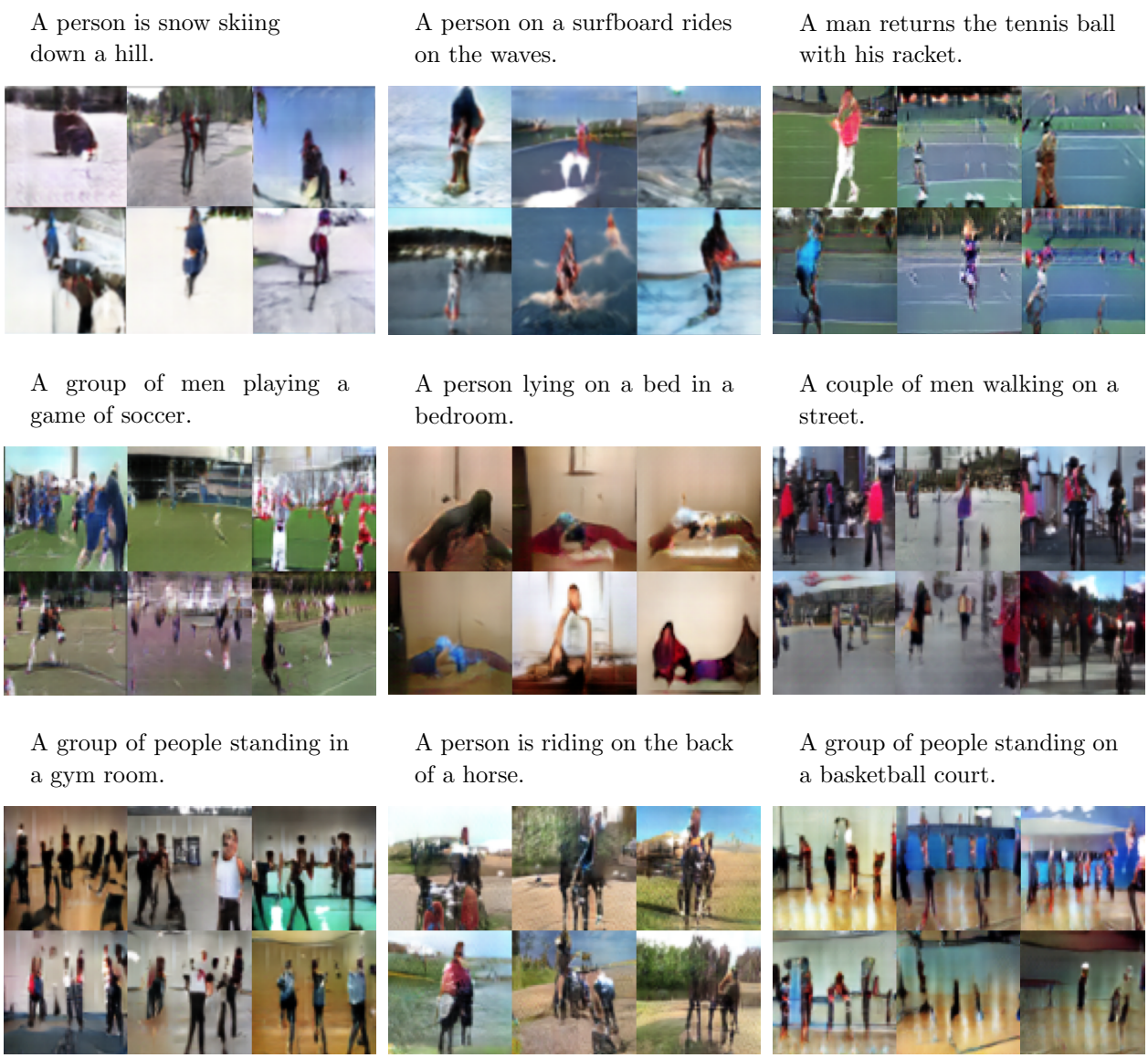

Figure 3.12: Examples of synthesised images on MPII using transfer learning.

and text matching, I2T2I-Transfer has slightly better scores than I2T2I. This may be because I2T2I-Transfer only needs to synthesise human activities, which reduces the requirement of network capacity. Therefore, under the same model architecture, I2T2I-Transfer can better synthesise human activities. For diversity, I2T2I-Transfer has a better score than both GAN-CLS and I2T2I. This may because MPII is a dataset built for human pose estimation [72], which contains many more human activities and human poses compared to MSCOCO. Thus, learning to synthesise human activities using MPII is better than using MSCOCO.

However, as described in Section 2.2.5, transfer learning tends to work if the datasets have common and general features. In other words, if two datasets are irrelevant, the transfer learning will fail, or if

two datasets are not fully relevant, the performance of transfer learning will be reduced. For example, in Figure 3.12, the results of "a person lying on a bed in a bedroom" do not correctly match with the text description. The person appears to be doing yoga instead of lying on a bed. The reason is that, MPII does not contain images with people sleeping. An extremely case is that, if the categories in the dataset for training the text-to-image generator are unseen by the image captioning module in its training phase, the image captioning module will fail to synthesise text descriptions. For example, it is impossible to use a carton image dataset to train the text-to-image generator. Another extreme case is that, if the dataset for training the text-to-image generator does not contain the categories we want to synthesise in the testing phase, the text-to-image generator will fail to synthesise the images. For example, as MPII only contains human activities, the text-to-image generator cannot synthesise images of different categories such as a car, bus, or animal which exists in MSCOCO. Nevertheless, without using a pre-trained image captioning module, GAN-CLS is not able to learn the text-to-image synthesis on the MPII dataset. Therefore, transfer learning is an advantage of our method over GAN-CLS, which has the potential to improve the image synthesis for a desired category.

3.4 Conclusions and Discussions

This chapter investigated a novel approach to improve the image's visual quality for text-to-image synthesis. To the best of our knowledge, this is the first work that can performs transfer learning for the text-to-image synthesis task. An image captioning module is adapted to synthesise text descriptions for each image in the training set to fill gaps in the text space. To compare with the work by Reed *et al.* [24], we use the same generator and discriminator architectures. Human evaluation is incorporated to compare our method with Reed *et al.* [24] based on image reality, text matching, and image diversity. The result from the MSCOCO dataset shows that our method has better scores than the Reed *et al.* [24] method in terms of image reality and text matching. However, Reed *et al.* [24] provided a better score on image diversity. The transfer learning result on the MPII human pose dataset shows that, by learning to synthesise human activities only, our method result in better scores on all three criteria than Reed *et al.* [24] method. However, a limitation of transfer learning is that we need to ensure that the image captioning module can synthesise text descriptions for the image dataset that does not have labelled text descriptions.

Learning from synthesised data is gaining more attention in the computer vision field. For example,

some studies use computer graphic techniques to synthesise plausible images for learning pose estimations [115] and object segmentations [116]. Other recent studies [117, 118] improve on liver lesion classification by using synthesised data from a GAN. In the future, it will be interesting to investigate how to use synthesised data to improve natural language processing applications. To improve the image visual reality for text-to-image synthesis, many subsequent studies exists that focuses on increasing the image resolution [20, 86, 119]. This approach is different from our method in that we fill the gaps in the text space. A subsequent study [86] successfully increases the image resolution from 64×64 to 256×256, and synthesises more detailed objects. In the future, it will be interesting to increase the image resolution of our approach.

Chapter 4

Efficient Image-to-Image Translation

The previous chapter presents a novel method for text-to-image synthesis, which can improve the image visual quality without requiring label more data manually. In this chapter, image synthesis is conditioned on images instead of text, also known as image-to-image translation. Compared to text descriptions, images can provide accurate semantic visual information, such as the precise location and shape of an object. More importantly, image-to-image translation is additionally challenging compared to text-to-image synthesise because the data is difficult and even impossible to obtained from the real world.

The unsupervised image-to-image translation is reviewed in the following that syntheses images conditioned on input images without using paired images to supervise the training. This proposed method is verified on the task of face swapping, portrait gender transformation, and image inpainting. The limitations of the proposed method are also analysed followed by describing how the subsequent study addresses these limitations, which is essential for the next chapter.

4.1 Introduction

Images can provide semantic visual information that is difficult to provide with text, such as in the visual details of a human face or the precise shape, colour, texture, and location of a bird. Many advanced deep learning tasks are considered as translating images from one domain to another domain by creating synthesised images for various purposes. The "edges to photo" example in Figure 4.1 can assist designers with fast prototyping. Another example is the manipulation of features in a

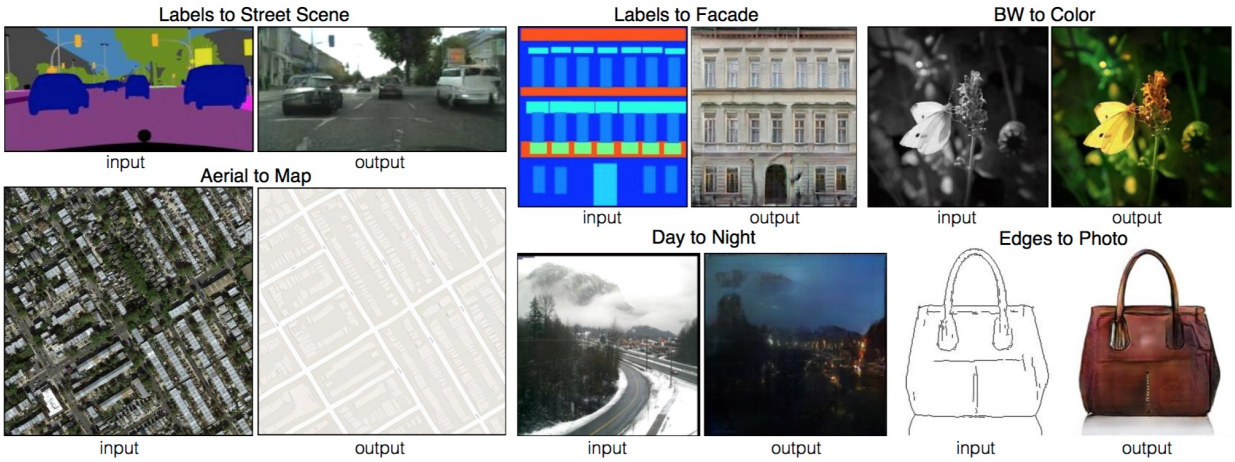

Figure 4.1: Examples of supervised image-to-image translation. Image is from Pix2Pix [13].

photo, such as the weather or background. The machine learning community has focused heavily on image-to-image translation tasks (*e.g.,* face swapping [120], changing the time of day in an outdoor image [121], changing the weather in an image [122], and composing a photo from sketch [123]), but each is approached via task-specific algorithms. Therefore, it would be preferred to have a "universal" algorithm to achieve all these tasks.

Recently, the work by Pix2Pix [13] proposed a method for learning image-to-image translation with supervision that was successful in solving various tasks by only changing the training datasets. The concern with this approach is the challenge of collecting large amounts of paired images for supervised training. For example, collecting landscape images of different seasons requires photographs of the same location with the same camera settings across extensive periods. While this instance is a difficult but doable task, many other scenarios exist that are impossible. For example, changing the gender represented in face images is not realistic as it is not possible to capture a single person's face with different genders.

Therefore, learning image-to-image translation from unpaired datasets offers more practicability. In the following, we study unsupervised image-to-image translation in which no paired images are required for training. Specifically, a generator is trained to synthesise images conditioned on the joint distribution of a class or domain label (*e.g.,* female and male faces) and a noise vector (*i.e.,* latent representation). As the generator synthesises images conditioned on the joint distribution of the class label and noise vector, by eliminating the noise vector and changing the class label only, the process will synthesise the images with the common features except the information related to the class label that changed. Next, an encoder is trained to map images back to the latent representation. Therefore,

when given an image, we can first map the image into the latent representation, then feed this and the changed class label into the generator to synthesised the images for another domain.

To evaluate the proposed method, the method is first demonstrated through face swapping for two people and two backgrounds without using paired images. Next, portrait gender transformation is demonstrated that changes the gender of arbitrary face images. This task is more challenging compared to face swapping because the model must create a new human face and be trained on many different faces with various backgrounds (*i.e.,* 202,599 face images contained in CelebA [124]) instead of only two faces with two backgrounds. The model also must understand diverse backgrounds, clothing and facial details, such as a facial expression, and hairstyle and colour all in an unsupervised way. Finally, an image inpainting task is demonstrated as an image-to-image translation problem, that translates incomplete images to completed images. In this task, the method not only learn the latent representations of the input images but also learn to synthesise the missing information. The experimental results show that the proposed method successfully achieves the unsupervised image-to-image translation. We next analyse the limitations of our approach and how the subsequent study addresses these limitations. The analysis of these limitations is essential for our study in the next chapter in which we learn to control the generated results using both the semantic visual information of images and the object attribute information of text description.

4.2 Related Works

4.2.1 Auxiliary classifier generative adversarial networks

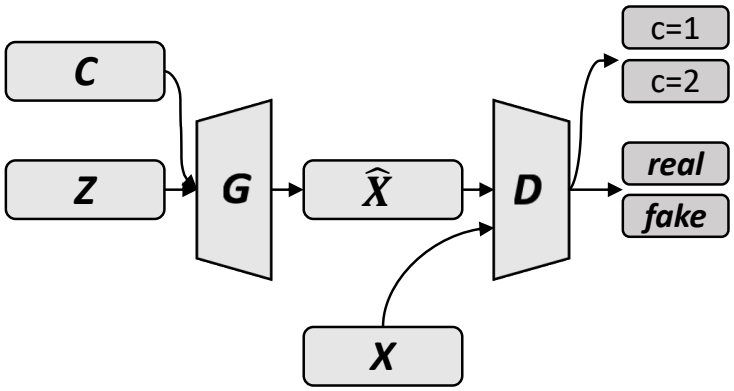

Figure 4.2: Auxiliary classifier generative adversarial network.

Auxiliary classifier GAN (ACGAN) [16] is a conditional GAN extended from DCGAN for synthesising images conditioned on a discrete class label c. In this chapter, we will use ACGAN for the generation part. The class label c and the noise vector z are concatenated as the input of the generator G, as Figure 4.2 shows. Following the verified model architecture of DCGAN [19], ACGAN uses deconvolutional layers to decode the joint latent space of z and c to perform image generations. More specifically, image synthesis conditioned on a class label is highly multi-modal because there are numerous possible images that can correctly match a class label. To address the multi-modality problem between image and class, ACGAN utilised the latent distribution z to synthesise various images for each class label c by finding the conditional probability $P(x|z,c)$ rather than $P(x|c)$. For the discriminator D, it is a convolutional encoder network that outputs the class probability D_c of the input image, and the probability D_x of the input image to be a real image.

To generate plausible and correct images, the generator is not only trained to fool the discriminator in terms of realistic images but is also trained to maximise the probability of the synthesised image to fit the correct class label. Extended from the vanilla GAN described in Equation 2.22, the loss function of the generator is defined as follows:

$$\mathcal{L}_G = \mathbb{E}_{x\sim p_{data}}[logD_x(G(z,c))] + \mathbb{E}_{z\sim p_z}[logD_c(G(z,c))] \tag{4.1}$$

The discriminator learns to classify the real image x to the correct class and maximise the probability of the real image to be the positive sample. At the same time, it learns to classify the synthesised images from the generator as a negative sample and to minimise the probability of the synthesised images to fit the correct class. The loss function of the discriminator is defined as follows:

$$\begin{aligned}\mathcal{L}_D = \; & \mathbb{E}_{x\sim p_{data}}[logD_x(x)] + \mathbb{E}_{z\sim p_z}[log(1 - D_x(G(z,c)))] \\ & \mathbb{E}_{x\sim p_{data}}[logD_c(x)] + \mathbb{E}_{z\sim p_z}[log(1 - D_c(G(z,c)))]\end{aligned} \tag{4.2}$$

By competing with each other, the generator and discriminator can enhance each other. At the end, the generator is able to synthesise images that can match the given class label.

4.2.2 Supervised image-to-image translation

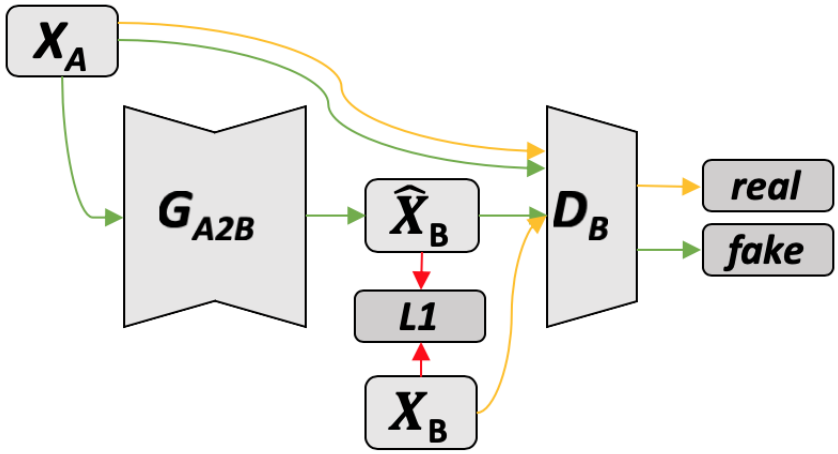

Figure 4.3: Pix2Pix: image-to-image translation network.

Pix2Pix [13] proposed a supervised image-to-image translation to achieve different image processing tasks, as shown in Figure 4.1. Figure 4.3 illustrates the training pipeline of Pix2Pix. Given paired images X_A and X_B, the authors use a convolutional network as the generator, in which the input and output have the same size. First, as the red lines indicated, Pix2Pix computes the $\mathcal{L}_1$ loss between the target and output images and updates the generator in a supervised way. In addition, as the orange and green lines indicate, to improve the visual quality of the output image, the discriminator classifies real paired images (*i.e.,* X_A and X_B) as positive samples and inputs images with synthesised images (*i.e.,* X_A and $\hat{X}_B$) as negative samples. By learning to fool the discriminator, the generator learns to synthesise plausible images conditioned on the input image. The loss functions for Pix2Pix can be summarised as follows:

$$\begin{aligned} \mathcal{L}_D = &\quad \mathbb{E}_{x\sim p_{data}}[logD(x_A, x_B)] + \mathbb{E}_{x\sim p_{data}}[log(1 - D(x_A, G(x_A)))] \\ \mathcal{L}_G = &\quad \mathbb{E}_{x\sim p_{data}}[logD(x_A, G(x_A))] + \mathbb{E}_{x\sim p_{data}}\lambda||x_B - G(x_A)|| \end{aligned} \tag{4.3}$$

where x_A and x_B are images from domains A and B, respectively, $\hat{x_B} = G(x_A)$ is the synthesised image for domain B, and the λ is the scale factor of the $\mathcal{L}_1$ loss that can help to control the weight between the $\mathcal{L}_1$ loss and the adversarial loss.

Following this study, there are many studies that focus on improving the visual quality of images,

such as Pix2PixHD [125] and Cascaded Refinement Networks [126]. However, all these studies need paired images for training, which limits the practicality of image-to-image translation and can only translate images from one domain to another domain within one generator. In contrast, our study is to investigate an unsupervised method to achieve the image-to-image translation and the way to translate images from one domain to another domain or vice verse using the same generator. At the end of this section, we will discuss the limitations of our proposed method, and then discuss how the subsequent studies, such as the study on CycleGAN [25] can address the limitations.

There are also many computer vision and image processing studies that can be considered image synthesis conditioned on an image [25]. For example, image super resolution [29] that inputs a low-resolution image, then outputs a high-resolution image. Moreover, image inpainting [127] inputs an incomplete image, then outputs a complete image by filling in the missing parts. Depth estimation [128], which inputs an RGB image, then outputs the estimated depth map to represent the distances between every pixel and the camera. Another study is style transfer [28], which inputs a photograph, then outputs the modified image in Monet style. All these studies can achieve state-of-the-art results using deep learning in a supervised way. However, compared with these studies, our study focuses on achieving image-to-image translation in unsupervised way rather than improving the performance of a specific task, such as the image resolution and visual quality of the image.

4.3 Methods

Our image translation model constitutes a convolutional generator G and encoder E. The translation process takes inputs of images from the source domain and outputs images of the target domain, which is controlled by a given discrete class/domain label. Before being able to use the model, a two-step training process is necessary. The first step trains the image generator G for all classes, and the second step trains the image encoder E to map the given images to the latent space. During the translation process, the given image x is first mapped to the latent variables via the encoder E. The encoder then inputs both the estimated latent variables $\hat{z}$ and the desired class label c to the generator G. The details of the method are further discussed in the following.

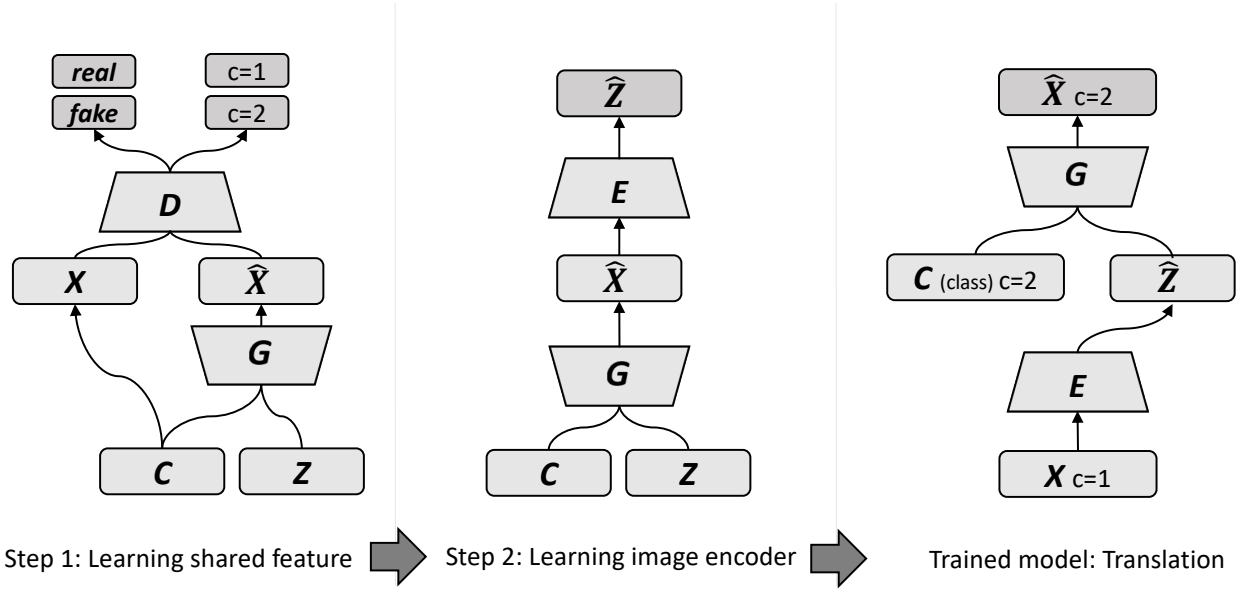

Figure 4.4: Network architectures of two-step learning for unsupervised image-to-image translation.

4.3.1 Learning shared features

To utilise both the GAN's capability of unsupervised feature learning and the convolutional encoder's representation capabilities, we split the learning process into two steps. In the first step, we use an ACGAN [16] to learn the global shared latent representation of images from different classes, training the generator G to synthesise images for each class c, and use the normal distribution as the latent variable z (*i.e.,* the noise vector). As the generator synthesises images from class labels and latent variables, by keeping the latent variables fixed and changing the class label only, the synthesised images are able to share a global structure and change the features corresponding to the class label. For example, the synthesised images will share the same global features in the form of synthesising the same person's face while changing facial expressions or generating the same bird but with different colours [16].

This idea is based on the fact that class-independent information contains a global structure regarding the synthesised images [16]. More specifically, ACGAN [16] finds the conditional probability $P(x|z,c)$, which means the image is synthesised conditioned on latent variable z and class label c. The class label c can be considered as a specific feature of the joint latent variable $z+c$, and the latent variable z is shared to the images of different class labels c. By learning to synthesise images conditioned on the join latent variable $z+c$, the shared latent variable z represents all features except the class label c. Therefore, we can expect the common features across the domains to be captured by the shared

latent variable z [16,19]. We leverage this feature for translating the global structure from one domain to another.

Algorithm 4 Training the generator for our unsupervised image-to-image translation.

Input: image x, normal distribution z, number of iterations n

for $i = 1$ **to** n **do**
  $\hat{x} \leftarrow G(z, c)$ synthesise the image conditioned on the normal distribution and label
  $s_r, c_r \leftarrow D(x)$ real image
  $s_f, c_f \leftarrow D(x)$ synthesised image
  $\mathcal{L}_D \leftarrow log(s_r) + log(c_r) + log(1 - s_f) + log(1 - c_f)$
  $D \leftarrow D - \alpha\delta\mathcal{L}_D/\delta D$ update the discriminator
  $\mathcal{L}_G \leftarrow log(s_f) + log(c_f)$
  $G \leftarrow G - \alpha\delta\mathcal{L}_G/\delta G$ update the generator
end for
return G

Algorithm 4 illustrates the training process of the image generator step by step. At every iteration, a batch of images $\hat{x}$ are randomly synthesised using the generator G conditioned on the normal distribution and random labels. Then, given the synthesised images $\hat{x}$ from the generator G, the discriminator D outputs s_f, the probability of the synthesised image to be real, and, c_f, the probability of the synthesised image to be the correct label. On the other hand, given the real images x with the corresponding labels c from the dataset, the discriminator D can output s_r, the probability of the real image to be real, and c_r, the probability of the real image to be the correct label. At the end of one iteration, following Equations (4.1) and (4.2), the generator G learns to fool the discriminator D by making it D output s_f and c_f as 1. Meanwhile, the discriminator D learns to discriminate the real/fake image and correct label by learning to output s_f and c_f as 0, and s_r and c_r as 1. The return of this algorithm is the trained generator G, which will be used in the next step, while the discriminator D will no longer be used in the next step.

4.3.2 Learning image encoder

In the second step, as the middle of Figure 4.4 illustrates, we learn the mapping from the image to the shared latent representation z by introducing a new network. We add an encoder E and put it on top of the previously trained generator G. Then, we fix the pre-trained generator G and expect the encoder E to reconstruct the input latent noise vector z. The mean square error (MSE) between the input latent variables and output variables is used as the loss. The training of encoder is supervised, which is different from the previous GAN training that requires the generator and discriminator compete with each other. The loss function is as follows:

$$\mathcal{L}_E = \mathbb{E}_{c\sim p_{data}, z\sim p_z} \left\| z - E(G(z,c)) \right\|_2^2 \tag{4.4}$$

There are two key points about this training. First, instead of feeding just one class of images to the encoder E, we feed the images of all classes to it. This not only enables the encoder E to work with the images of different classes but also forces the encoder E to learn the shared features of different classes. Second, as we use the normal distribution as the noise vector, the outputs of the encoder are linear (*i.e.,* no activation function).

Algorithm 5 Training the encoder for our unsupervised image-to-image translation.

Input: generator G, normal distribution z, label c, number of iterations n

1: **for** $i = 1$ **to** n **do**
2: $\hat{x} \leftarrow G(z, c)$ synthesise the image conditioned on the normal distribution and random label
3: $\hat{z} \leftarrow E(\hat{x})$ reconstruct the input normal distribution
4: $\mathcal{L}_E \leftarrow \|z - \hat{z}\|_2^2$
5: $E \leftarrow E - \alpha\delta\mathcal{L}_E/\delta E$ update the encoder
6: **end for**
7: **return** E

Algorithm 5 illustrates the training process step by step. Given the pre-trained generator G from Algorithm 4, at every iteration, a batch of normal distribution vectors z and class labels c are randomly input into the pre-trained generator G. We then have a batch of synthesised images $\hat{x}$. Given these synthesised images $\hat{x}$, the encoder E encodes them into latent representation $\hat{z}$. At the end of one iteration, the encoder E learns to reconstruct the latent representation by minimising the MSE between the input normal distribution vector z and the reconstructed output $\hat{z}$. The return of this algorithm is the trained encoder E.

4.3.3 Translation

After training the image generator G and image encoder E, as the right-hand side of Figure 4.4 illustrates, our encoder can now map images to the shared latent representation z and then synthesise images of the target domain with the generator. More specifically, given an image $x_{c=1}$ from the domain $c = 1$, we first use the image encoder E to synthesise its shared latent variable $\hat{z} = E(x_{c=1})$, and then use the synthesised latent variable as the input to the generator G to synthesise the translated image $\hat{x}_{c=2} = G(\hat{z}, c = 2)$ for the domain $c = 2$.

As the generator G can synthesise images for both classes and the image encoder E is trained using images of both classes as the input, our method can achieve the translation from the domain $c = 1$ to $c = 2$ or vice verse without an extra generator. In contrast, the Pix2Pix and subsequent study — CycleGAN both require training another generator to perform the inverse translation [13, 25]. In other words, our method can achieve bidirectional translation using the same model, the translation processes of our method can be summarised as follows:

$$\hat{x}_{c=2} = G(E(x_{c=1}), c = 2) \tag{4.5}$$

$$\hat{x}_{c=1} = G(E(x_{c=2}), c = 1) \tag{4.6}$$

4.3.4 Network architecture

To stabilise the GAN training, the architectures of both the generator G and discriminator D are extended from DCGAN [19]. More specifically, for the generator G, we set the number of latent variables z to 100 and embedded the class label to a vector of five values, then concatenate them to a vector of 105. As a 2D convolutional operation can only be applied into a 3D volume, to utilise the 2D CNN for image generation, we then apply a fully connected layer to decode the vector of 105 values to a vector of 8,192 values and reshape it into a volume with the size of $4 \times 4 \times 512$. Following the verified architecture of DCGAN, four deconvolutional layers with a filter size of 5 and a stride of 2 are used to decode the volume to a 64×64 RBG image. Batch normalisation is applied to the output of the fully connected layer and every convolutional layer except the output layer. The ReLU is applied to the output of every batch normalisation layer. To stabilise the training, images pixel values are rescaled to -1 $\sim$ 1 [1, 19]; thus, we use the hyperbolic tangent function on the output layer.

For the discriminator D, like the DCGAN, four convolutional layers with a filter size of 5 and a stride of 2, are used to encode the 64×64 image into a volume of size $4 \times 4 \times 512$. Batch normalisation is applied to all convolutional layers except for the first one. Moreover, to stabilise the GAN training, leaky-ReLU with a slope of 0.2 is applied to the end of all batch normalisation layers [19]. Following this, to output the class probabilities and the probability of the image to be real, we then flatten the volume into a vector of 8,192 values. After that, we apply two separate fully connected layers to output the class probabilities and the probability of the image to be real, respectively. The architecture of

the encoder E follows the discriminator D except that the final layer outputs a vector with 100 values to reconstruct the latent variables z.

4.4 Evaluation

4.4.1 Datasets and training details

Datasets	Tasks
Presidential debate videos	face swapping
CelebA	portrait gender transformation
Street View House Number	image inpainting

Table 4.1: Datasets and tasks for unsupervised image-to-image translation.

Table 4.1 summarises the datasets and tasks. We first present our results on the presidential debate video for face swapping. We detected and extracted face images of Obama and Clinton from two short presidential debate videos. This resulted in 8,452 images for Obama and 5,065 images for Clinton, with a variety of facial expressions. We also split the images into 80% for training and 20% for testing. For the second task, we use the CelebA face dataset [124] for portrait gender transformation. The CelebA face dataset contains 84,434 male images and 118,165 female images. The dataset has a variety of backgrounds and faces. We also split this dataset into 80% for training and 20% for testing. Lastly, we present our results of image inpainting by using the Street View House Number (SVHN) dataset [109], which contains 73,257 digits for training and 26,032 digits for testing.

In all three cases, we used the same network architecture and training method. To stabilise the training, we used the learning rate of 0.0002, and Adam optimiser [61] with a momentum of 0.5 [19]. These settings have been empirically found to be able to stabilise the GAN training in both DCGAN and ACGAN studies [16, 19]. During training, we used a batch size of 64, trained 100 epochs for Step 1, and 20,000 iterations for Step 2. The number of epochs of Step 1 is based on the visual quality of the synthesised image, and the number of iterations of Step 2 is based on whether the loss would not be reduced. For the data augmentation in Step 1, we randomly flip the image horizontally with a probability of 50%, zoom in on the image with a range from 0 to 5%, and rotate the image with a degree of $\pm$10%. We experiment with different data augmentation settings and find that this setting generates the best visual quality of the synthesised images. The code was implemented by using

TensorLayer 1.3.11 [112] and TensorFlow 1.0.0 [113], and can be found in the Appendix. By using an Nvidia Titan XP GPU, training a generator takes about two days and an encoder takes 20 minutes.

4.4.2 Face swapping

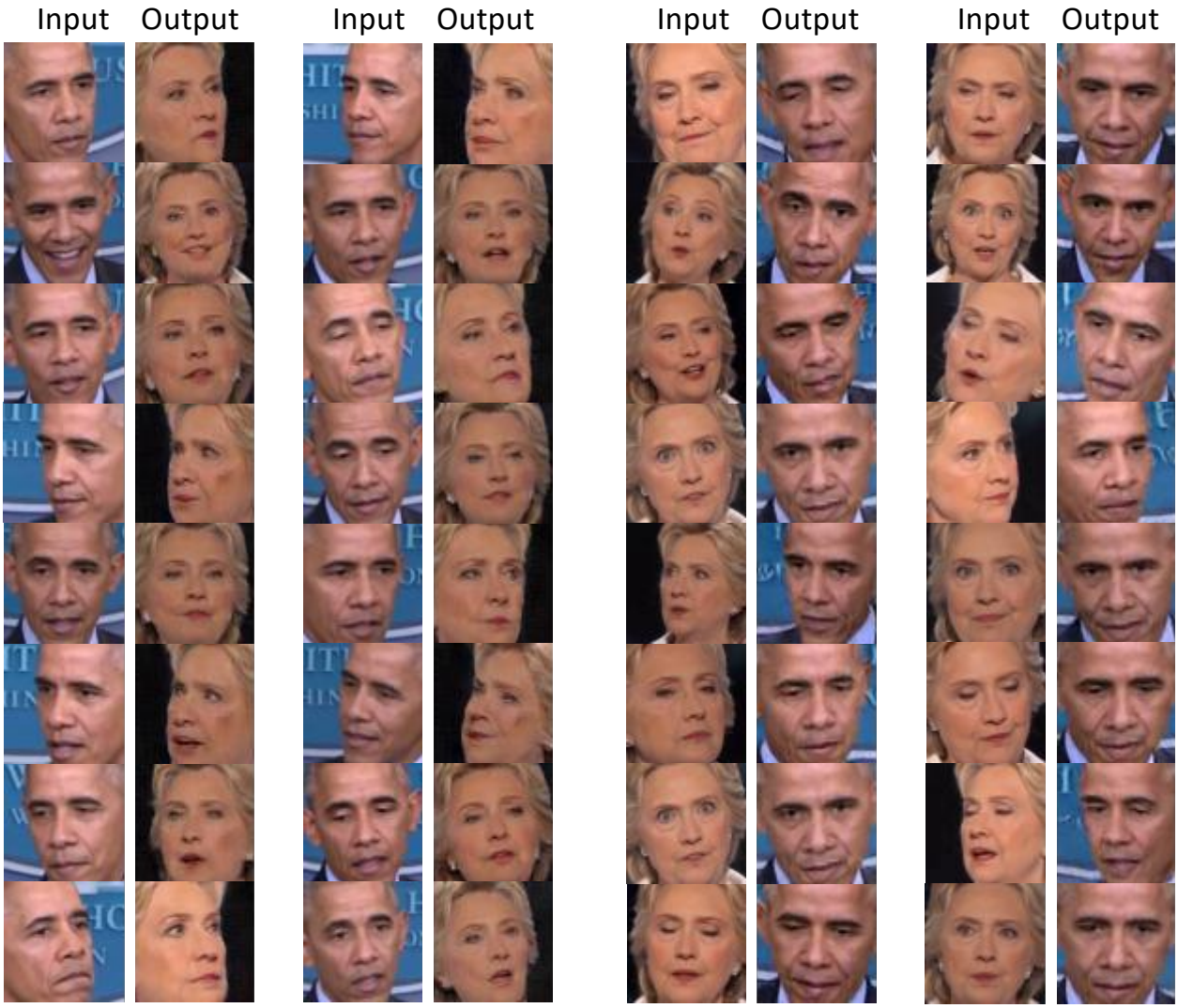

Figure 4.5: Example results of our method on face swapping. The left columns are input images, and the right columns are the corresponding synthesised output images.

The random results of our first task, face swapping, are shown in Figure 4.5. Our method not only learned to change the face and background but also learned to keep the face orientation to some degree. It means the latent variable z contains the shared features of Obama and Clintons faces, such as the face orientation. Taking the top-right image as an example, the synthesised Clinton image is facing right like the input Obama image. Another example can be seen in the bottom-right image,

where both the input Clinton image and the synthesised Obama image are facing forwards. As the image synthesis is conditioned on the class c and the latent variable z, the results can show that the latent variable z contains information about the face orientation.

Moreover, this experiment is based on two short videos with two different backgrounds. Therefore, the background is also translated rather than maintained (*e.g.,* Obama's images always have a blue background, while Clinton's images always have a black background). In addition, there is still a problem that the face key-point locations do not precisely match. This was noted in the bottom-right image, where the synthesised image of Obama's face is larger than the input image, and his mouth and eyes were not in the correct locations.

4.4.3 Portrait gender transformation

Face swapping is an image-to-image transformation task that focuses on two persons with two backgrounds, and transforms the face of one person into the face of another one, without creating new human face; while portrait gender transformation deals with faces of multiple people and various backgrounds, and must create new human faces to transform the gender of faces, which is more challenging than the face swapping. In this experiment, the portrait gender transformation is trained on 202,599 different faces with various backgrounds from the CelebA dataset. Figure 4.6 shows some random results of portrait gender transformation. The method not only learned the characteristics and expressions of human faces but also learned to reconstruct the background to some degree. Thus, the latent variable z contains the shared features, such as the facial characteristics and expressions and the background. Taking the top-left image for example, the synthesised male image not only kept the face orientation of the input image but also maintained the white background. Another example can be seen in the top-right image where the synthesised female image not only kept the smiling expression but also reconstructed the black clothes.

There are some failure cases in Figure 4.6. For example, in the bottom-left image, the background details cannot be reconstructed and all background pixels become the same colour. Another example is the bottom-right image where the characters in the background disappear in the output image. Nevertheless, the supervised learning methods [13, 125] cannot achieve portrait gender transformation because it is impossible to collect images of a person in different genders. However, the proposed unsupervised image-to-image translation method successfully achieve this task.

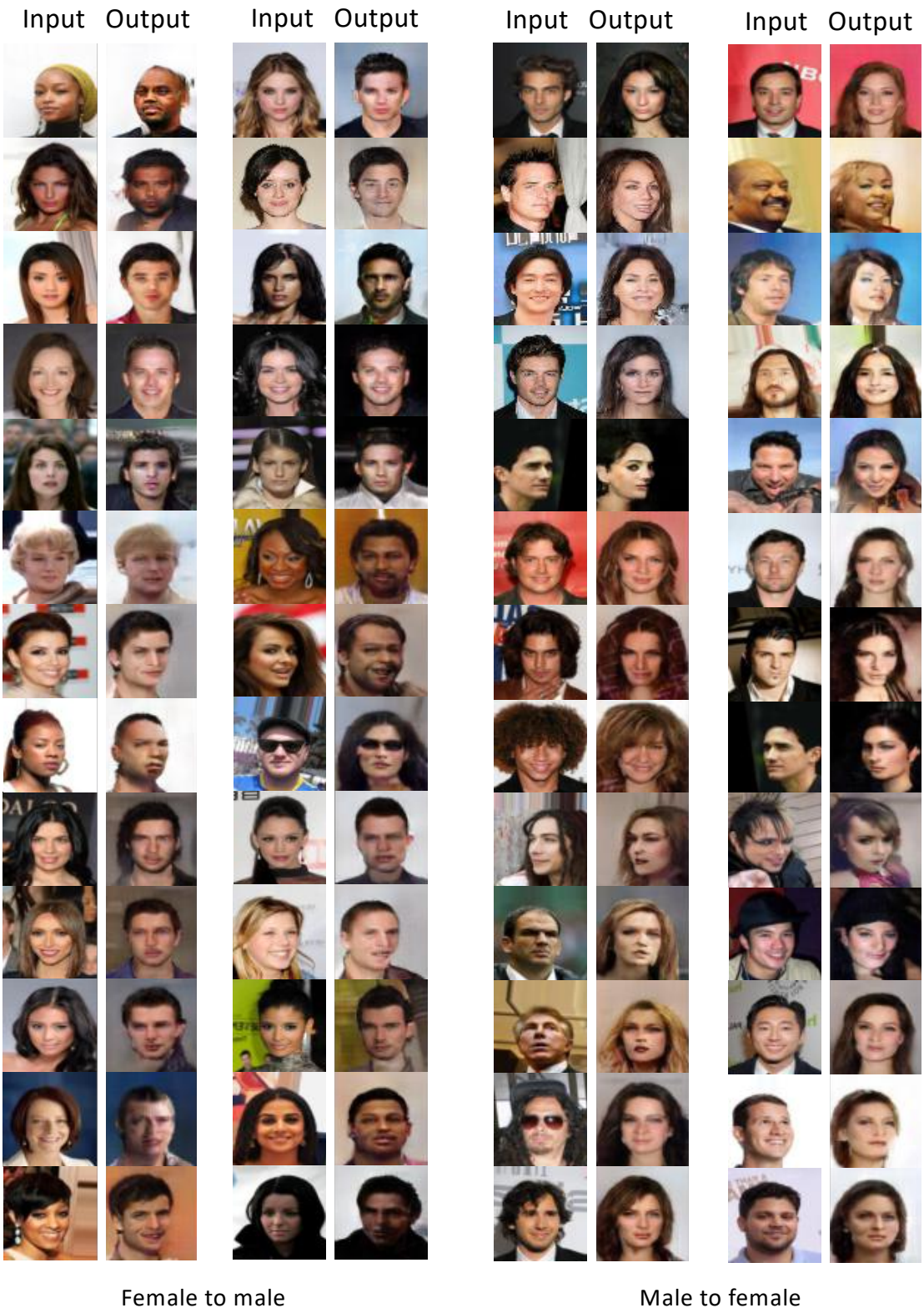

Figure 4.6: Example results of our method on portrait gender transformation. The left columns are input images, and the right columns are the corresponding synthesised output images.

4.4.4 Image inpainting

We can also consider the completed and uncompleted images to be two different classes and then translate the incomplete image to a completed image to achieve image inpainting. This task is more challenging than the previous two tasks, as the method needs to learn the latent representations of input images and to synthesise the missing information.

Figure 4.7 shows the results of transferring incomplete SVHN images to completed SVHN images. In every block, the first column contains the original ground truth images for comparison. The second column contains the ground truth images after removing the centre part using a square mask. Finally, the third column contains the translated images that recover the missing pixels. We can see that our model can synthesise the missing parts of the images that match the content and background of the ground truth images. For example, the top-left image shows that the model synthesises the missing part of the digit "2". To match the ground truth image, our model uses white colour to fill the digit and blue to fill the background. Thus, the latent variable z contains the shared features of both the completed and incomplete images, such as the digit and background colours.

To evaluate our method quantitatively, we compare our method with three representative methods. PatchMatch (PM) [129] is an unsupervised method that searches for similar patches in the image to inpaint the image. It has quickly become one of the most successful inpainting methods. The content-encoder (CE) [30] is a state-of-the-art GAN-based method specifically designed for image inpainting. However, CE is a supervised method that uses the $\mathcal{L}_2$ norm between the missing part and the target ground truth as a part of the loss function.

Figure 4.8 directly compares the results of our method and those of PM and CE. The results show that all methods can fill the missing part of the images to inpaint the digits. However, for PM and our method, a large artefact can be found on the edge of the inpainted area, which makes the images unrealistic. In contrast, CE has relatively fewer artefacts on the edge of the inpainted area.

To quantitatively compare the methods, we use structural similarity (SSIM) and the peak signal-to-noise ratio (PSNR) as the metrics [130]. These two metrics is to evaluate the similarity of two images and commonly used in evaluating image inpainting and super resolution methods [29,30] by comparing the output image with the ground truth image. The value of SSIM varies from 0 to 1, with a higher value indicating a better quality of inpainting. Specifically, the SSIM of two images x and y is defined in Equation (4.7).

Figure 4.7: Example results of our method on image inpainting on the Street View House Number (SVHN) dataset. In each subplot, the first column contains the original images. The second column contains the images to be inpainted, and the third column contains the inpainted images.

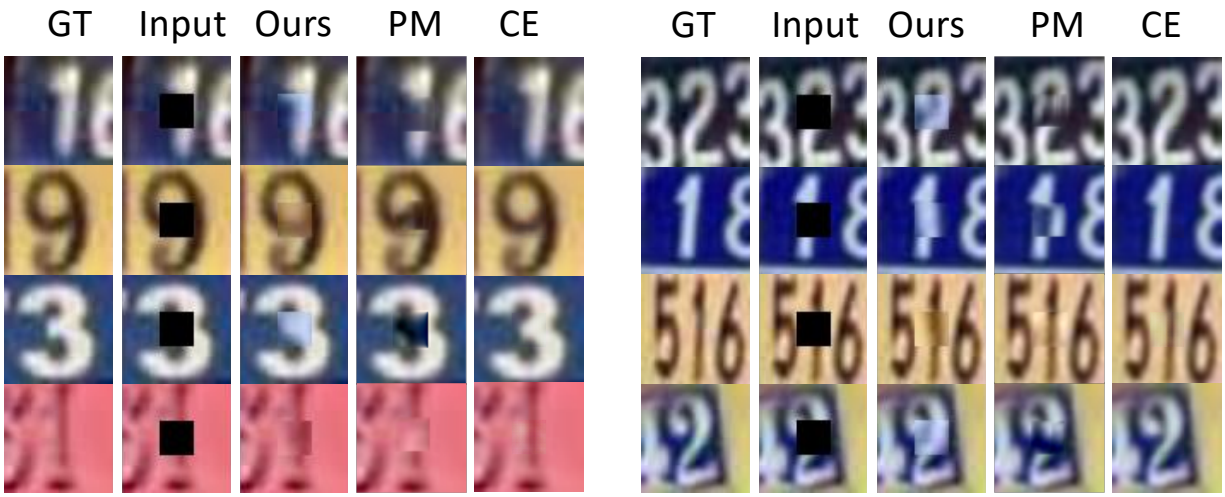

Figure 4.8: Comparison between our method and others. Note, GT: ground truth; PM: PatchMatch [129]; CE: content-encoder [30].

$$\mathcal{SSIM}(x,y) = \frac{(2\mu_x\mu_y+c_1)(2\sigma_{xy}+c_2)}{(\mu_x^2+\mu_y^2+c_1)(\sigma_x^2+\sigma_y^2+c_2)} \tag{4.7}$$

where μ_x and μ_y are the averages of x and y, respectively; σ_x and σ_y are the variances of x and y, respectively; σ_{xy} is the covariance of x and y; $c_1 = (k_1 L)^2$ and $c_2 = (k_2 L)^2$ are two values to stabilise the division with the small denominator, L is the dynamic range of image pixel values (*e.g.,* an image pixel value varying from 0 to 1 has a L of 1), and k_1 and k_2 are 0.01 and 0.03, respectively by default [130].

On the other hand, PSNR measures the ratio between the maximum possible power of data (*i.e.,* the maximum possible pixel value of an image) and the power of noise (*i.e.,* the MSE between the ground truth and prediction). A higher PSNR has a better quality of image inpainting. Specifically, PSNR in decibels (db) is defined in Equation (4.8).

$$\mathcal{PSNR}(x,y) = 10log_{10}(\frac{R^2}{MSE(x,y)}) \tag{4.8}$$

where R is the maximum possible pixel value in the image (*e.g.,* an image pixel value varying from 0 to 1 means R equals to 1).

	Ours	**PM**	**CE**
SSIM	0.9199	0.9034	0.9902
PSNR	24.58	22.81	35.69

Table 4.2: Quantitative comparison between our method and others. Notes, PM: PatchMatch [129]; CE: content-encoder [30].

Table 4.2 shows the quantitative results of our method and those of PM and CE on the test set of SVHN using SSIM and PSNR. Our method has better performance compared with PM, which is also an unsupervised method that does not use a ground truth image to learn a model in a supervised way. However, our method does not outperform CE because CE is a supervised learning method that uses the $\mathcal{L}_2$ norm between the missing part of the image and the ground truth image as a part of its loss function. However, our method learns the inpainting in an unsupervised way. On the other hand, compared with the methods specifically designed for image inpainting, a limitation of our method is that we cannot inpaint an image with random masks. For example, Figure 4.9 shows some inpainting results for recovering incomplete images with a random mask by removing 80% of the pixels..

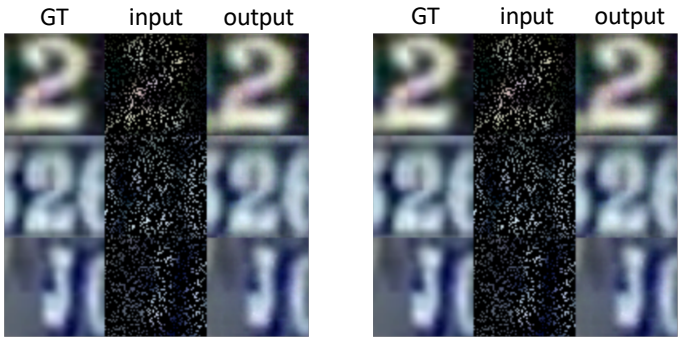

Figure 4.9: Example results of the content-encoder (CE) with random masks. The Image is from [127].

4.4.5 Failure cases

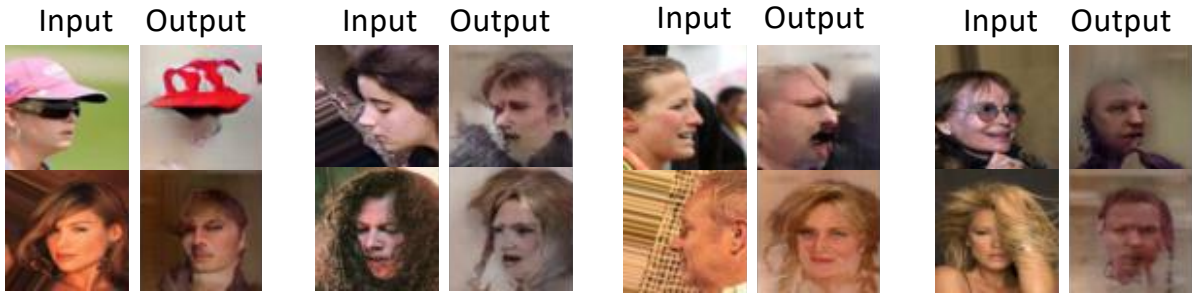

Figure 4.10: Failure cases of our method on portrait gender transformation. In each subplot, the left column contains input images, and the right column contains the corresponding synthesised output images.

Figure 4.10 shows the typical failure cases of unsupervised image-to-image translation on the portrait gender transformation task. We found the following three situations will easily lead to failed translations.

- **Uncommon facial orientation:** In the CelebA face dataset [124], most of the faces are facing forwards instead of towards the side. We observe that, when encountering some faces facing the side, the translation performance is reduced.
- **Uncommon wearing and obstruction:** For example, the top-left of Figure 4.10 shows that the network fails to understand a pink hat facing the side because it is uncommon in the training dataset. Moreover, for the same reason, the bottom-right images show that putting a hand over the face will result in a failed translation.
- **Skin colour background:** When the background colour is similar to the facial skin colour, we observe that the neural network will be confused. An example is the bottom images of the third subplot from the left.

Figure 4.11: Failure cases of our method on image inpainting on the Street View House Number (SVHN) dataset. In each subplot, the first column contains the original images, the second column contains the images to be inpainted, and the third column contains the inpainted images.

Figure 4.11 shows the failure cases of unsupervised image-to-image translation on image inpainting. We found the following two situations will easily lead to failed translations.

- **Large missing area:** If the digit is largely missing, we observe that the neural network may fail to synthesise the missing part.
- **Uncommon colour:** If the digit or background colours are uncommon, we observe that the neural network may fail to synthesise the correct colour to fill the missing pixels.

4.5 Limitations and Solutions

4.5.1 Limitations

Though our experiment shows promising results, there are still many limitations. For example, Figure 4.7 shows that the synthesised images have large artefacts where the shadow of the square mask appears on the results. In addition, our method was unable to recover incomplete images with random masks. More importantly, the proposed method failed to learn the complex translations shown in Figure 4.1. We failed to translate higher resolution images (*e.g.,* 256×256), which have defects in image quality and generality.

The limitations of our method are due to the hypothesis that the distribution of synthesised images $\hat{x}$ is the same as the distribution of the real images x. In other words, it assumes the generator G is able to synthesise not only images $\hat{x}$ that have the same quality as the real images x but must also all the images in the dataset. However, in practice, the synthesised distribution was not able to match the real distribution exactly, leading to two problems:

First, the training of the encoder E is based on the synthesised images $\hat{x}$ from the generator G; thus, the encoder E never observes the real images x, whereas, in the translation step, we input real images x to the encoder E. As the synthesised images $\hat{x}$ do not precisely match the real images x (*i.e.,* the synthesised image do not have the same quality as the real images), the training of the encoder E contains biases. Therefore, in the translation step, the encoder E will reconstruct inaccurate latent variables $\hat{z}$ for the real images x, resulting in poor translation. In theory, improving the quality of the synthesised images will result in less bias, and many studies focus on improving the visual quality and resolution for synthesising images [20, 86]. However, in practice, the quality of the synthesised images cannot be improved infinitely, so the bias caused using the synthesised images $\hat{x}$ to train the encoder E cannot be eliminated.

Figure 4.12: Example results of the GAN collapse. Each block contains the images from different collapsed generators trained on MNIST. All images are synthesised from different noise vectors. Image is from [131].

Second, in the vanilla GAN framework, the generator G is only required to synthesise images to fool the discriminator D. There are no constraints for the generator G to have the ability to synthesise all images in the training set [131]. Therefore, the generator is able to fool the discriminator, even if it is only able to synthesise a small portion of the images in the training set [131]. This phenomenon is called GAN collapse [131]. Figure 4.12 shows some results from collapsed generators, where the generators always output similar images given different noise vectors. In extreme cases, the generator G could fool the discriminator by ignoring the noise vector z and always outputting a unique plausible image [131].

Therefore, training the encoder with the synthesised images from the generator has the potential to miss the information of the training set. Then, in the translation step, when the input images contain

features that the encoder has never seen in its training phase, the synthesised latent vector $\hat{z}$ will have biases, resulting in poor or failed translation.

4.5.2 Solutions

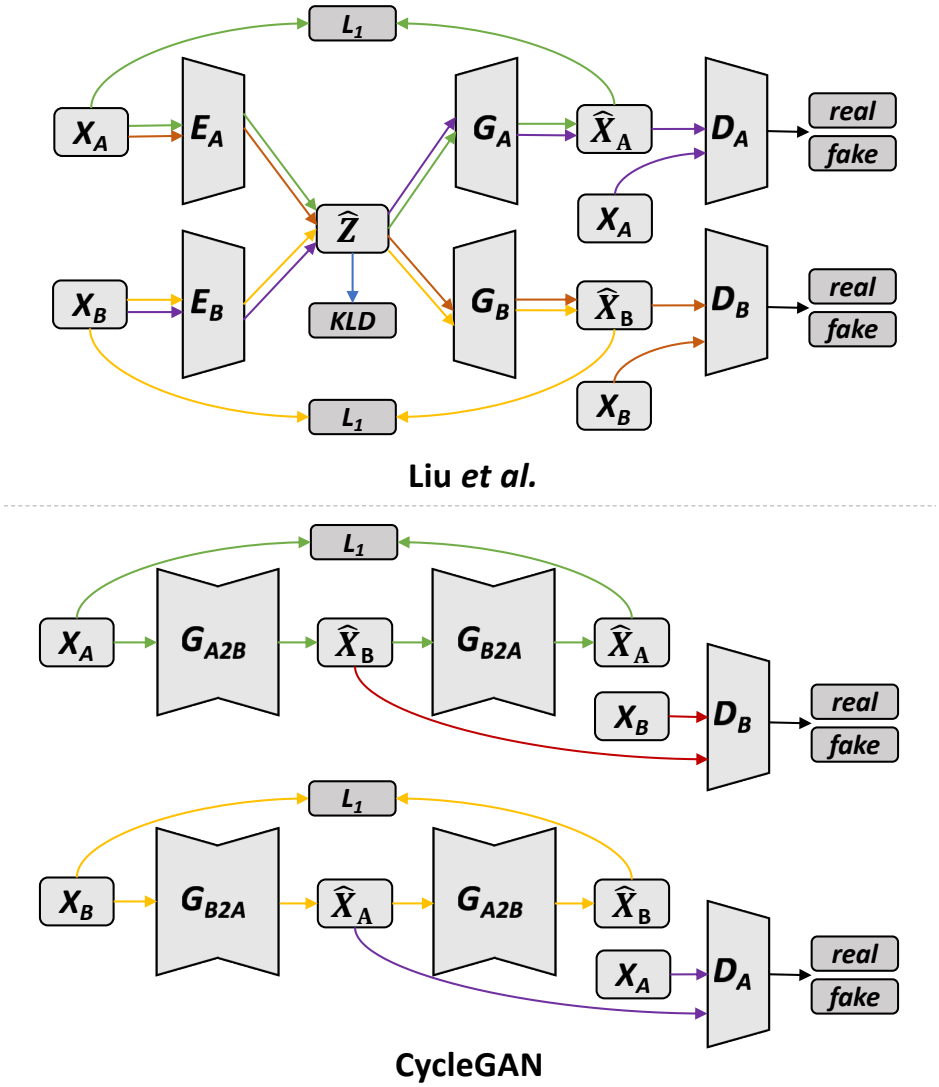

Figure 4.13: Methods that train the encoders using real images.

Without solving these problems, even with an ideal encoder (*e.g.*, a super deep encoder), the proposed two-step method cannot extend to more advanced tasks. Fortunately, there are some subsequent studies [25, 95] that were published after our experiments that have addressed these problems. All these methods train the encoder by inputting real images. In other words, the encoder is able to observe real images and encode them to latent representations.

Liu *et al.* [95] achieved unsupervised image-to-image translation by training two image encoders with real images as the inputs. As the top of Figure 4.13 shows, the images of domains "A" and "B" are encoded into the same latent space using two separate encoders, and then the method uses two generators and two discriminators to synthesise and discriminate the images, respectively. More specifically, the two generators not only need to reconstruct the images in an autoencoder way, as the green and yellow lines show but also need to synthesise plausible images for translations, as the red and purple lines show. The two discriminators are used to discriminate the synthesised images from the two generators, respectively. By competing with the discriminators, the two generators can synthesise images for domains "A" and "B", respectively. However, one limitation of this method is that the training is unstable due to the saddle point searching problem [95].

After that, three similar methods, CycleGAN [25], DualGAN [132] and DiscoGAN [133], are proposed to consider the encoders to be part of the generators, without encoding images into a normal distribution. As the bottom of Figure 4.13 shows, the methods consist of two generators for translating the images from domain "A" to "B" or from "B" to "A", respectively. Similar to the work by Liu *et al.* [95], two discriminators are adopted to discriminate the images from different domains. Meanwhile, the key of these methods is the reconstruction loss indicated by the green and yellow lines in the bottom of Figure 4.13. It recovers the synthesised images back to its original domain. By doing this, it makes the synthesised images contain more information of the original input images, rather than only satisfying the discriminators. The reason is that, if the synthesised images do not have sufficient information of the original input images, we cannot recover the synthesised images back to the original images [25, 132, 133]. Their experiment shows that a larger weight of the reconstruction loss can help the synthesised images contain more features of the original domain, but setting the weight as too large will result in failed translation (*i.e.,* the synthesised images will look the same as the input images). The weight of this loss is selected based on the experiment. This loss is called "cycle consistency loss" in CycleGAN [25] and is also known as cycle loss.

Compared to the work by Liu *et al.* [95], which requires the encoder to output a prior normal distribution, CycleGAN considers the encoder to be a part of the generator, it implicitly learns the latent distribution without requiring an encoder to output a prior normal distribution. The training of CycleGAN is more stable and successfully applies to more applications [25], this implicit way to learn the latent distribution has two advantages over using the prior distribution. First, simple image datasets may match the simple prior distribution, while diverse and complicated datasets, such as

multi-category natural scene images, require a more complex distribution, which is more appropriate [134]. Otherwise, this requires the generator to disentangle the latent factors from simple noise to complicated image distribution, which require a very deep network and a large amount of data [134]. Second, the implementation is simple, many existing fully convolutional network architectures can be used for reference.

4.6 Conclusions and Discussions

Unsupervised image-to-image translation to generate images conditioned on the input images was proposed with a method that does not require paired images to supervise the training. First, the approach learned to synthesise images conditioned on both latent variables and class labels and then trained an image encoder to map images to the latent variables. By doing so, given an image from one domain, it is first mapped to the latent variables, and then synthesised the translated image by using its latent variables and the changed class label. As the latent variables capture the common features across different domains [16], the synthesised images will contain the common features of the input images to achieve unsupervised image-to-image translation. We evaluate our method on the tasks of face swapping, portrait gender transformation, and image inpainting.

The limitations of the proposed method were analysed and introduced how subsequent studies [25, 132, 133] can address these limitations. This analysis is important for the next study that provides controllable generated results using both semantic visual information and object attribute information from the image and text description, respectively. Moreover, the proposed method can translate images from two domains to each other using the same generator, while the subsequent studies require an extra generator.

Chapter 5

Efficient Semantic Image Synthesis

The previous two chapters study image synthesis conditioned on either text description or image. This chapter proposes a method to synthesise images leveraging the advantages from both text description and image to achieve more controllable generation compared to the previous methods. Specifically, the synthesised images contain the the semantic visual information of the input images, such as the shape and location of a bird or flower, while also matching the object attribute information from the text description, such as the colour and texture of the bird or flower. Comparing with Chapter 4 where the model only changes the input images with a single condition (*e.g.,* change the gender), this method uses a text description to control changing the input image. In addition, compared with Chapter 3 where the model synthesises images conditioned on text descriptions only, this method uses an image to provide conditions for controlling the corresponding features of the object.

5.1 Introduction

Text descriptions provide object attribute information, such as the colour of an object (*e.g.,* bird or flower) and images provide semantic visual information, such as the location and shape of an object. The previous two chapters study controllable and data efficient image generation methods that use either text description or image as the controlling condition. Further improvements to the generative method should be attainable through the use of more than one type of information to control the results. Recently, extending the from text-to-image synthesis, Reed *et al.* [135] proposed a GAN-based method that synthesises the object to match both the text description and bounding box coordinates.

For example, given a bounding box on the left-hand side of an image and the text description of "this bird is completely black.", the output image is expected to be a blackbird on the left-hand side of the image. This method offers more controllable results compared to the text-to-image synthesis because it can use the location information as the additional information to control the synthesis. However, this method cannot control the synthesis using other semantic visual information, such as the background and object shape information. In this chapter, to offer more controllable results, we combine the advantages of the text description and image to synthesise images containing both the semantic visual information from the input images and the attribute information from the text description, which we call semantic image synthesis.

The challenge of this task is that after combining the input text description and image, the ground truth images are unknown. It is impractical to create such labelled images manually to supervise the training because, for example, 1,000 images and 1,000 text descriptions would provide one million combinations. In other words, if we want to supervise the training, we need to create one million ground truth images manually. Instead of doing this arduous process, an adversarial method is proposed along with a novel model that only requires images and their matching text descriptions (*i.e.,* matching image-text pairs) for the training.

Specifically, the proposed method has a discriminator, a generator, and a text encoder. First, the text encoder encodes the text descriptions into the embedded vectors and feeds them into the generator and discriminator. To address the unknown ground truth problem, the discriminator learns to classify by matching image-text pairs as real samples and learns to classify the mismatched image-text pairs as fake samples. In doing so, the discriminator offers a strong signal to the generator about which matching image and text pairs should appear, impelling the generator to synthesise images that match the text description. Meanwhile, the generator synthesises images using the input images and text descriptions, and the discriminator learns to classify the synthesised image and text description pairs as fake samples. The generator and discriminator can enhance each other by competing. At the end of the training, the generator achieves semantic image synthesis without requiring the ground truth output images.

To evaluate the proposed method, the results are demonstrated using Caltech-200 bird [102] and Oxford-102 flower datasets [103]. The results show that the synthesised images can maintain the background and shape information of the input image while matching the text descriptions. The interpolating results of the image and text spaces are visualised, respectively. By interpolating two

input images on the output of the encoder portion of the generator and fix the text description, the synthesised images can have a smooth change in the object shape while keeping the same colour and texture information to match the text description. On the other hand, by interpolating two text descriptions on the output of the text encoder and fix the input image, the synthesised images can have a smooth change in colour and texture while keeping the shape and background to match the input image. In this chapter, the information from the images and text descriptions are combined to synthesise new images, and this method can be extended to different data types in the future.

5.2 Related Works

5.2.1 Text-to-image synthesis

To solve the multi-modal problem of image and text, Reed *et al.* [24] proposed a deep neural architecture based on the conditional GAN framework, which successfully generated plausible 64×64 images from text descriptions. To increase the image resolution, Zhang *et al.* [86] proposed StackGAN, which successfully generated 256×256 images. Compared with the previous text-to-image synthesis methods built upon the GAN framework, our proposed method is a variant of the conditional GAN framework. The synthesised image is conditioned not only the input text description but also the input image. Utilising the semantic visual information of the input image and the object attribute information of the input text description, our method manipulates the input image and requires the output image to match with the input text description.

5.2.2 Other text-based image synthesis

Synthesising images conditioned on the class label or text description can have more applications than the vanilla GAN, but it does not provide control over object location or pose [135]. It is difficult and even impossible to contain all the precise information such as the background, location, colour, and texture into a text description. To make the image synthesis more controllable, Reed *et al.* [135] proposed a GAN-based network that can synthesise the object that matches with both the text description and bounding box (or key-points) coordinates. The example results are shown in Figure 5.1.

Compared with text-to-image synthesis, this method can control the location of the object. However, as the key-point and bounding box only contain the location information, the background and object

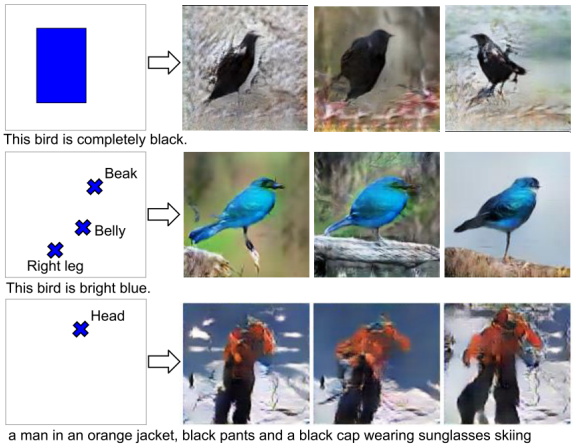

Figure 5.1: Example results of image synthesis conditioned on both the text description and key-points (or bounding box); image from [135].

shape information are not provided. Moreover, this method requires manually labelled object location information (key-point or bounding box) for supervised learning. Compared with this study, the purpose of our proposed method is to manipulate the input images conditioned on the input text descriptions, where the output image is able to maintain the location, shape, pose and background of the object of the input images. In addition, our method does not require labelled object location information for training.

5.2.3 Baseline method

Reed *et al.* [24] proposed a method for semantic image synthesis, denoted as "style transfer" in their paper. This is the only generative model able to tackle the same image synthesis task described in this chapter. However, the main purpose of their work is text-to-image synthesis, rather than using the text description to manipulate the image semantically, as we do in this chapter.

We use the method proposed by Reed *et al.* [24] as a baseline approach for comparison. This method can be considered a special case of the two-step method that we introduced in this thesis. More specifically, the first step is to pre-train a generator network G that can generate plausible samples of image $\hat{x}$ from latent variables z, and the second step was to further train an encoder network E that inverts the synthesised image $\hat{x}$ back to the latent variable $\hat{z}$. An MSE $\mathcal{L}_E$ loss between the input and output latent variables was employed for training encoder E:

$$\mathcal{L}_E = \mathbb{E}_{t\sim p_{data}, z\sim p_z} \left\| z - E(G(z, \varphi(t))) \right\|_2^2 \tag{5.1}$$

where G and E represent the generator and encoder, respectively, z is the normal distribution, φ is the RNN text encoder, and t is the text description from the training set. Therefore, $G(z, \varphi(t))$ is the synthesised image conditioned on the text. Optimising encoder E with the MSE loss can learn to reconstruct the latent variable z from the input image x. During inferencing, the trained encoder E first encoders the image x into latent variables $\hat{z}$, then the trained generator G synthesises a new image based on $\hat{z}$ and the embeddings of the target text description $\varphi(t)$.

However, as discussed in Section 4.5, the main drawback to these kinds of approaches is that the encoder E has been trained only on the synthesised/fake image $\hat{x}$ from the generator, rather than on the real image x. As it is almost impossible for the generator to generate a distribution of complex real data, the distribution of the synthesised image cannot fully match the real image distribution. Therefore, this training method creates biases in the encoder when encoding real images. Our method, in contrast, uses the implicit encoding approach discussed in Section 4.5. We consider the encoder to be a part of the generator, allowing the encoder to directly observe real images, while not mapping images into a normal distribution.

5.3 Methods

Our proposed method consists of a generator network G, a discriminator network D and a text encoder network φ. The inputs to the generator are an image and a text embedded from the text encoder. We consider the image encoder to be a part of the generator, and encode the input image into the feature space, then concatenate the output of encoder with the text embedding. The rest of the generator then decodes the combined features to a synthesised image. Compared with the baseline method from [24], we employ the proposed architecture along with a specific adversarial loss function to optimise the learning for image synthesis. The proposed architecture enables us to utilise real images for training the image encoder, avoiding the bias caused by using the synthesised image to train the encoder, as discussed in Section 4.5. In addition, the latent representation of images is directly from the convolutional layer, which contains more spatial information (*i.e.,* the latent representation is a

three-dimensional volume (tensor) rather than a vector). These are the critical differences in network architecture that allowed our model to synthesise much better images than the baseline method.

For adversarial learning, the inputs of the discriminator are an image and a text embedding. The discriminator learns to classify the matching image and text pairs as positive samples and to classify the mismatched pairs and synthesised images as negative samples. In doing so, the discriminator can pass a signal to the generator including how the real image should appear and how the matching image and text pair should appear. To fool the discriminator, the generator not only needs to synthesise a plausible image but also needs to ensure the synthesised image can match the given text description. More details are described as follows.

5.3.1 Network architecture

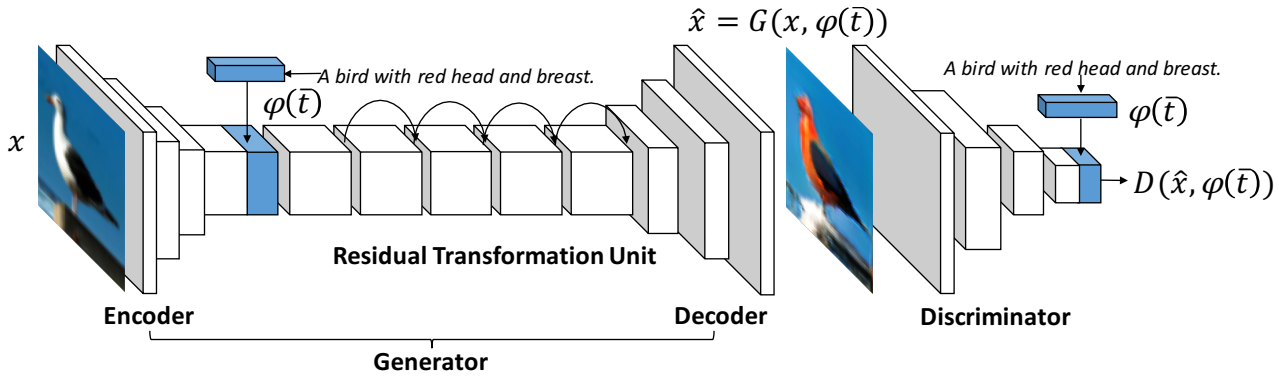

Figure 5.2: Network architecture of the semantic image synthesis model, consisting of a generator G, discriminator D, and text encoder φ. The text encoder encodes the text description into a vector and feeds the vector into both the generator and discriminator. The generator consists of an encoder, residual transformation unit, and decoder. The discriminator is an image encoder that outputs the probability of whether the input image and text pair are real.

The proposed method is built upon a conditional GAN framework, conditioned on both images and text descriptions. Our generator adopts an encoder-decoder architecture. The encoder part of the generator is employed to encode input images and text descriptions. The decoder then synthesises images based on latent representations of the image and text. The discriminator D performs the distinguishing task conditioned on image and text semantic features. More specifically, Figure 5.2 illustrates the details of our network architecture. Inspired by [28], the generator is designed to include an image encoder, a decoder, and a residual transformation unit.

The image encoder is a convolutional neural network (CNN) that encodes input source images of size $64 \times 64 \times 3$ into spatial latent representations with the dimensions of $16 \times 16 \times 512$. The encoder

consists of three convolutional layers. The first layer uses a filter size of 3×3 and stride of 1×1. The other two convolutional layers use a filter size of 4×4 and stride of 2. The 3D spatial latent representations are critical to the performance of our model, as they retain the convolutional features of the source images. To stabilise the training, following DCGAN [19], we use ReLU activation in all convolutional layers. Moreover, batch normalisation [68] is performed in all layers except the first convolutional layers.

To encode the text descriptions, we pre-train the text encoder φ using the same method described in Section 3.2. The only differences are that the texts are not from the image captioning module but the training set and that we use LSTM with a hidden size of 128. We further apply a text embedding augmentation method proposed by Zhang *et al.* [86] on $\varphi(t)$, making it to a dimension of 128. Similar to GAN-INT [24] and our proposed method in Section 3.2.2, this augmentation method can generate a mass of additional text embeddings during training. In addition, it can introduce noise distribution to the generator, which allows the model to synthesise many images when given a text description. More importantly, this method is easily implemented and added to any architecture. After encoding the text description to a new vector of 128, we duplicate the text embeddings spatially to be $16 \times 16 \times 128$ and, finally, concatenate it with the encoded image latent representations.

In contrast, for image input, we did not add noise because it would be difficult for the model to maintain image features irrelevant to the text input (*e.g.,* backgrounds), thus violating the purpose of our task. However, the synthesised images can still be sampled differently due to the added noise of text input, which we thought would be a better way to maintain diversity.

After concatenating the image and text-latent representations, the joint latent representation needs to pass through a residual transformation unit, which is made up of four residual blocks [10]. Following the verified design in [86], each residual block consists of two convolutional layers with a filter size of 3×3 and stride of 1×1, the batch normalisation and ReLU activation are applied into every layer. This residual transformation unit can further encode the concatenated image and text latent representations jointly. There are two main reasons to include this unit. First, the residual architecture makes the generator network easy to learn the identity function, so that the similar structure of the input source image can be retained to the output image [28]. This is an important property for our task, as we expect our model to keep the features of input images. Second, the deeper network can have a "deeper" encoding process, which can help to better encode the joint image and text-latent representations [10, 86].

The output of the residual transformation unit is fed into a decoder, which consists of several layers that decode the latent feature representations into synthesised 64 × 64 RGB images. To stabilise the training, following DCGAN architecture [19], both ReLU activation and batch normalisation are adopted in all layers except the last layer. More specifically, the decoder consists of two up-sampling layers, where each layer is double the size of its input feature map using bilinear interpolation. Moreover, convolutional layers with a filter size of 3×3 and stride of 1×1 are applied on each up-sampling layer. In the end, a convolutional layer with a filter size of 3×3 and a stride of 1×1 makes the output $64 \times 64 \times 3$ (*i.e.,* RGB three-channel image). To scale the pixel values of synthesised images from -1 to 1, hyperbolic tangent activation is applied to the output of the generator.

For the discriminator network, as the right-hand side of Figure 5.2 shows, we first apply the convolutional layers to down-sample the images into latent representations of $4 \times 4 \times 512$. Following the verified design in [86], this down-sampling process consists of four convolutional layers with a filter size of 4×4 and a stride of 2, and the batch normalisation and leaky-ReLU with a slope of 0.2 are applied to each convolutional layer. After that, one residual block is applied to further encode the image representation. It consists of three convolutional layers with a stride of 1×1. Moreover, the first convolutional layer uses a filter size of 1×1 and the others use 3×3. Given the encoded image representation, similar to the generator network, we concatenate the image latent representation with the text embeddings. Finally, we employ two convolutional layers to produce the final probability. The first convolutional layer uses a filter size of 1×1 and a stride of 1×1, and the final convolutional layer uses a filter size of 4×4 and a stride of 1×1, which make the output a single value.

5.3.2 Adversarial loss

In this task, the ground truth images are unknown, and we do not know what the outputs of the generator should be. It is impossible to draw all possible images for each image and text description manually; otherwise, given n images and m texts, we need to manually draw $n \times m$ images, which is impractical. In other words, it is impractical to create a dataset with ground truth images to supervise the training. By utilising adversarial learning, we can provide an automatic method to learn the implicit loss function, rather than creating outputs to be specific known targets.

Figure 5.3 shows the training process of the proposed method. We denote the matching text description as t, the mismatched text description as $\hat{t}$, and the semantically relevant text description as $\bar{t}$. More

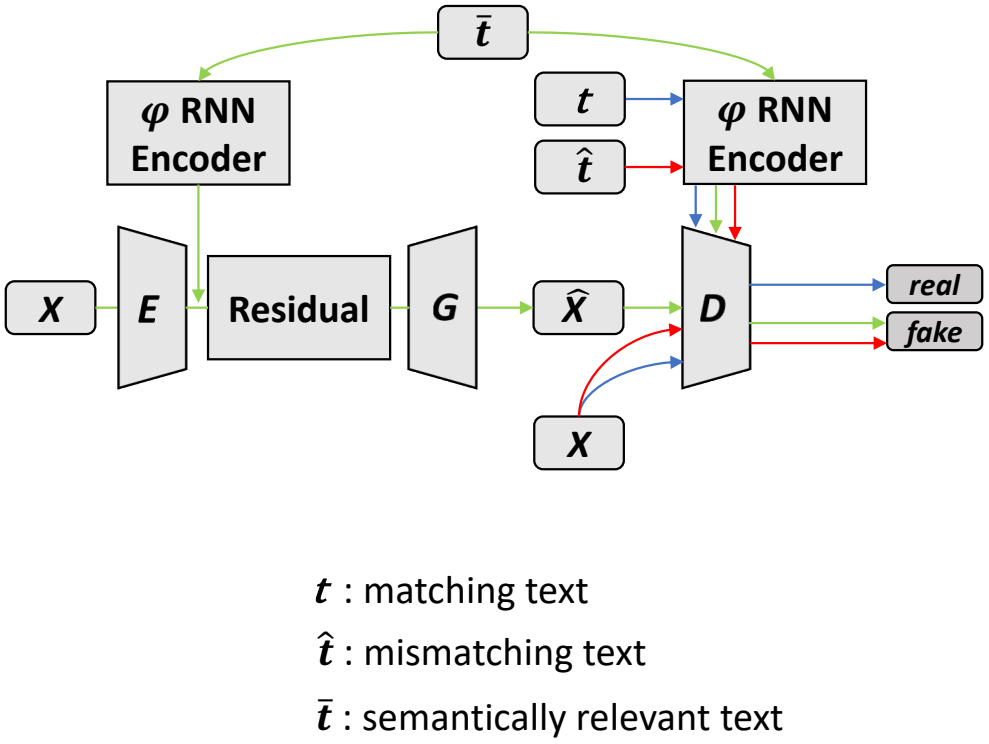

Figure 5.3: Adversarial learning for semantic image synthesis.

specifically, the semantically relevant text description $\bar{t}$ includes the matching text description t and other related but not precisely matched texts (*e.g.,* given an image of a specific kind of bird, $\bar{t}$ can be text describing other kinds of birds but cannot be text for other objects, such as flowers and buildings). We denote the probability of a text description match for an image x as s, and the synthesised/fake image from generator $G(x, \varphi(\bar{t}))$ as $\hat{x}$. We indicate different types of input pairs of the discriminator D in different colours in Figure 5.3. The discriminator D learns to classify the following three types of data, where the symbols $+$ and $-$ denote positive and negative examples for the discriminator.

- $s_r^+ \leftarrow D(x, \varphi(t))$ classifies the real image with the matching text description as a positive example, indicated by blue lines.
- $s_w^- \leftarrow D(x, \varphi(\hat{t}))$ classifies the real image with the mismatched text description as a negative example, indicated by red lines.
- $s_s^- \leftarrow D(\hat{x}, \varphi(\bar{t}))$ classifies the synthesised image with a semantically relevant text description as a negative example, indicated by green lines.

The first term, s_r^+, is the basic loss that teaches the discriminator to learn the relation of the image and the correct/matching text description. The second term s_w^- is proposed by Reed *et al.*[24] to jointly train with the first term (s_r^+) to enable the discriminator to learn stronger image and text matching signals. In other words, the discriminator knows what the matching image and text pairs should be. Therefore, this term can make the generator synthesise a plausible image to match the text description better [24]. The loss of these two terms for updating the discriminator D is defined as follows:

$$\begin{aligned}\mathcal{L}_D = & \mathbb{E}_{(x,t)\sim p_{data}} log D(x, \varphi(t)) \\ & + \mathbb{E}_{(x,\hat{t})\sim p_{data}} log(1 - D(x, \varphi(\hat{t})))\end{aligned} \tag{5.2}$$

The definition and distinction between the semantically relevant text description $\bar{t}$, mismatched text description $\hat{t}$, and the matching text description t are extremely important for the proposed method. Therefore, before we describe the details of the final term s_s^-, we would like to explain the text definition first.

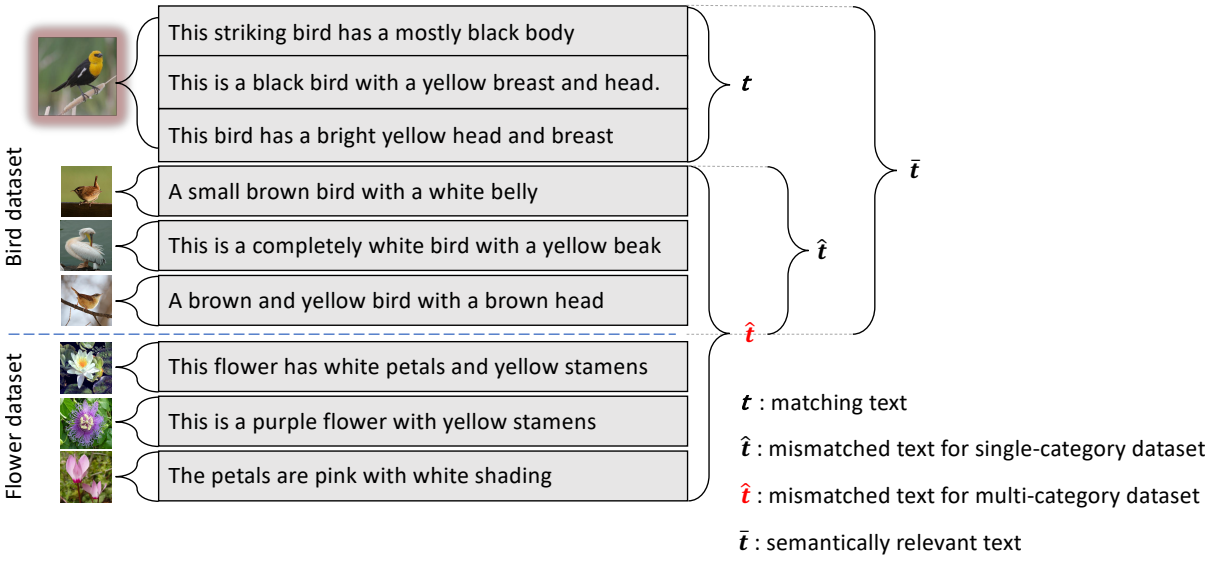

Figure 5.4: Definition of different text description types for single- and multi-category datasets.

Figure 5.4 illustrates the difference between the mismatched text description $\hat{t}$ and the semantically relevant text description $\bar{t}$. For a single category dataset (*e.g.,* bird only), as the yellow-headed bird picture on the top of Figure 5.4 shows, the mismatched text descriptions $\hat{t}$ are the texts apart from

the matching text description t of the specified image, and the semantically relevant texts $\bar{t}$ are all text descriptions in the dataset (*i.e.*, $\bar{t} = t + \hat{t}$).

The reason we denote $\bar{t}$ as "semantically relevant text" rather than "all text" is that, for multi-category datasets, the $\bar{t}$ should not include the texts of other categories. Figure 5.4 illustrated the situation for combing the bird and flower datasets. The mismatched text $\hat{t}$ is still the texts apart from the matching text, while the semantically relevant text $\bar{t}$ is all the text belonging to the category of the specified image. In other words, extending the dataset from a single- to multi-category would not affect the matching texts t and semantic relevant texts $\bar{t}$. Only the mismatched texts $\hat{t}$ would be changed, as the red symbol in Figure 5.4 indicates.

The final term (s_s^-) is one of the key contributions of the proposed method and makes the proposed method work. The first two terms (s_r^+ and s_w^-) already teach the discriminator D to distinguish matching and mismatched pairs. Then for the final term s_s^-, as the generator synthesises images via $\hat{x} \leftarrow G(x, \varphi(\bar{t}))$ and the discriminator classifies the synthesised images as negative examples, the generator is trained to fool the discriminator that outputs a positive result for $D(\hat{x}, \varphi(\bar{t}))$. By doing this, the generator learns to manipulate the input images to synthesise images that can match the text description.

The definition of $\bar{t}$ makes the image synthesis task more reasonable. It does not make sense to synthesise images if the given image and target text description are irrelevant. For example, it is not reasonable to modify a "bird" to a "building". Following the discussion above, extending from Equation (5.2), the final loss functions of the proposed method are defined in Equation (5.3).

$$\begin{aligned}
\mathcal{L}_D &= \mathbb{E}_{(x,t)\sim p_{data}} log D(x, \varphi(t)) \\
&\quad + \mathbb{E}_{(x,\hat{t})\sim p_{data}} log(1 - D(x, \varphi(\hat{t}))) \\
&\quad + \mathbb{E}_{(x,\bar{t})\sim p_{data}} log(1 - D(G(x, \varphi(\bar{t})), \varphi(\bar{t}))) \\
\mathcal{L}_G &= \mathbb{E}_{(x,\bar{t})\sim p_{data}} log(D(G(x, \varphi(\bar{t})), \varphi(\bar{t})))
\end{aligned} \tag{5.3}$$

where t denotes the matching text, $\hat{t}$ denotes the mismatched text, and $\bar{t}$ denotes the semantically relevant text. We update the generator with $\mathcal{L}_G$ and the discriminator with $\mathcal{L}_D$. The first two terms of $\mathcal{L}_D$ are the losses for s_r^+ and s_w^- from Equation (5.2), corresponding to matching text and mismatched

text. The $\mathcal{L}_G$ and the final term of $\mathcal{L}_D$ are the losses for s_s^- corresponding to the semantically relevant text $\bar{t}$. By optimising these loss functions along with our network architecture, the encoder part of the generator can utilise real images to learn the mapping from the images to the latent representations.

In the following, we describe the training details about the semantically relevant text description. At the early stage of training, as the generator cannot synthesise plausible images, the discriminator can learn to distinguish between real and synthesised images while ignoring the text description via the first two terms (s_r^+ and s_w^-). As training continues, the generator gradually learns to synthesise plausible images that may or may not match the text description. Then, the generator starts to learn to modify the images via the text description. At this time, the choice of text descriptions starts making a difference.

If we use the matching text t in s_s^-, the generator can only observe the matching pairs (x and t). Then, the generator can fool the discriminator by learning to over-fit to the input images and ignore the text description. In other words, the synthesised images of the generator will be the same as the input images. In this case, the update signal from the discriminator is always from positive s_r^+. The text information signal from the term s_w^- has never been used. No adversarial process for text description is preserved.

If we use the mismatched text $\hat{t}$ in s_s^-, the input pair to the discriminator is the synthesised images and mismatched texts. The second term, s_w^-, will let the discriminator D pass a strong signal to the generator. Finally, the update will allow the generator only to synthesis images that match $\hat{t}$. It will become a problem when using multi-category datasets. If the generator is forced to transfer two irrelevant objects, the generator will over-fits the text descriptions. Apart from that, $\hat{t}$ does not contain t, and the generator cannot implicitly learn to keep the image unchanged when inputting the matching text description.

In addition, the proposed method can be viewed as a variant of conditional GAN conditioned on the image and text. The loss function allows the generator $G(x, \varphi(\bar{t}))$ to capture the conditional generative distribution $p_G(\hat{x}|x, \bar{t})$ to fit the distribution of the real data $p_{data}(x, t)$. Combined with the proposed network architecture, the loss function adversarially makes $\hat{x}$ plausible and matches what $\bar{t}$ describes. Algorithm 6 illustrates the training process step by step. At every iteration, a batch of images x are first randomly selected, and then randomly select the corresponding mismatched text $\hat{t}$, mismatching text $\hat{t}$ and semantically relevant text $\bar{t}$. All texts are encoded into text embeddings

Algorithm 6 Training algorithm for semantic image synthesis via adversarial learning.

Input: image x, matching text t, mismatched text $\hat{t}$, semantically relevant text $\overline{t}$, number of iterations n

for $i = 1$ **to** n **do**
  $h \leftarrow \varphi(t)$ encode matching text
  $\hat{h} \leftarrow \varphi(\hat{t})$ encode mismatched text
  $\overline{h} \leftarrow \varphi(\overline{t})$ encode semantically relevant text
  $\hat{x} \leftarrow G(x, \overline{h})$ forward generator with real image and arbitrary text
  $s_r \leftarrow D(x, h)$ real image, matching text
  $s_w \leftarrow D(x, \hat{h})$ real image, mismatched text
  $s_a \leftarrow D(\hat{x}, \overline{h})$ synthesised image, semantically relevant text
  $\mathcal{L}_D \leftarrow log(s_r) + (log(1 - s_w) + log(1 - s_s))/2$
  $D \leftarrow D - \alpha\delta\mathcal{L}_D/\delta D$ update discriminator
  $\mathcal{L}_G \leftarrow log(s_s)$
  $G \leftarrow G - \alpha\delta\mathcal{L}_G/\delta G$ update generator
end for

using the pre-trained text encoder. Thus, we denote the text embedding of the matching text as h; $\hat{h}$ for the mismatched text and $\overline{h}$ for the semantically relevant text. Next, semantically relevant text is fed into the generator to obtain the synthesised image $\hat{x}$. Given x, $\hat{x}$, h, $\hat{h}$ and $\overline{h}$, we can obtain the outputs of discriminator s_r for inputting the real image x with its matching text t, s_w for inputting the real image x with its mismatched text $\hat{t}$, and s_a for inputting the synthesised image $\hat{x}$ with the semantically relevant text $\overline{t}$. At the end of one iteration, the generator and discriminator are updated using these three outputs (*i.e.,* s_r, s_w, s_a) according to the loss functions defined in Equation (5.3).

5.3.3 Feature enhancement

The feature representations of the input images are from the encoder part of the generator which is trained along with the entire model. Because of the limited size of the datasets we used, the encoder may not be capable of producing good representations, which limits the quality of the synthesised images. A pre-trained image encoder could help enhance the feature details of the synthesised images. Therefore, we further propose an alternative model that employs a much deeper pre-trained CNN to perform the encoding function.

Instead of feeding 64 × 64 images into the generator, we use 244 × 244 images, which contain more semantic visual information, as the input of the generator. As described previously the encoder part of the generator consists of three convolutional layers that can encode the 64 × 64 images into spatial latent representations with a dimension of 16 × 16 × 512. To enhance the feature details of the synthesised images, we replace the encoder part of the generator with a VGG [45] network that is

pre-trained on ImageNet [4]. More specially, to use the same architecture of the decoder part of the generator, the VGG encoder should have the same output dimensions (*i.e.,* 16 × 16 × 512) with the original encoder. To achieve feeding 244 × 244 images to the VGG, as shown in Figure 2.7, we use the output of the fourth convolutional block as the encoded spatial latent representation. The other parts of the generator are maintained to be the same.

5.3.4 Beyond 64 x 64

All of our methods were only able to synthesise 64 × 64 images. While it would be interesting and more useful to synthesis higher resolution images with the size of 256 × 256. However, our methods failed to synthesise 256 × 256 images. The images were totally distorted (some failed examples are shown in Figure 5.16). We also tried the state-of-the-art generator architectures borrowed from the studies in [13, 25], but our GAN training still failed. In addition, different from the 64× 64 image, we found that using the pre-trained VGG model as the encoder part of the generator did not work for the image size of 256 × 256, and even collapsed faster when using text embedding augmentation for the text [86].

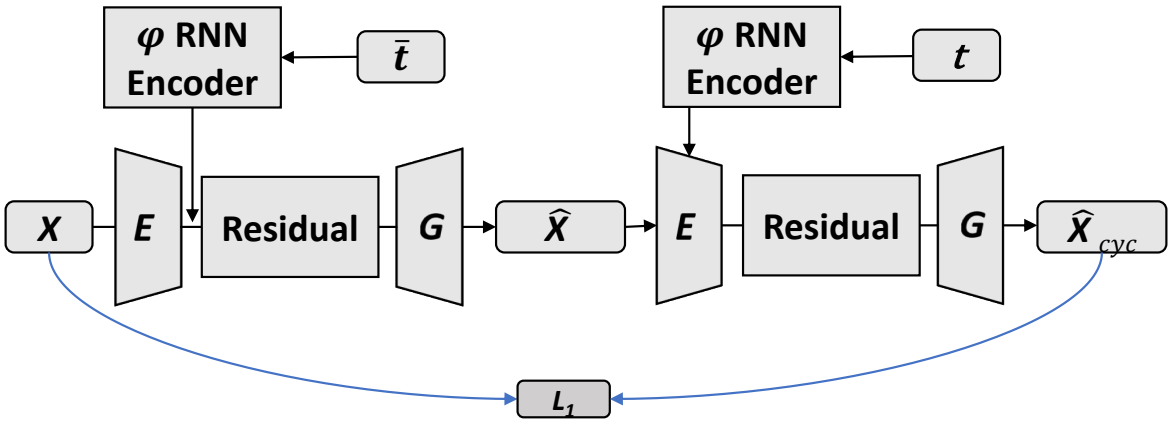

Figure 5.5: The proposed method with cycle loss. Left: The generator first manipulates the input image using the semantically relevant text; Right: The generator translates the synthesised image back to the input image using the matching text.

Since the ground truth images are unknown, the synthesised images from the generator cannot explicitly contain the same background as the input. As discussed in Section 4.5, training the GAN in an autoencoder fashion can help maintain input image background information and avoid GAN collapse because it forces the generator to synthesise the data in the training set. Therefore, to synthesise

256×256 images, inspired by CycleGAN [25], we introduce the cycle loss to improve the background and detail reconstruction. As Figure 5.5 shows, the cycle loss considers manipulating the image using relevant text $\bar{t}$ to be "encoding", and manipulating the synthesised image back to the input image using the matching text t to be "decoding". The cycle loss "forces" the synthesised images to maintain information from the input images; otherwise, the "decoding" process cannot reconstruct the input images well.

In terms of the model architecture, to synthesis images with a size of 256×256, we redesigned the architecture of the generator and discriminator. For the generator, to contain more spatial information from the input, the encoder part of the generator encodes the images into latent representations of $64 \times 64 \times 128$, which is 4 times larger than the original $16 \times 16 \times 512$. As a larger receptive field is required for larger images, after concatenating the convolutional output with the text embedding, we feed the output of the encoder into 16 residual blocks instead of four blocks for the 64×64 image. The decoder part is the same as the original one. For the discriminator, as the image size is increased from 64×64 to 256×256, extending from the discriminator, we apply four extra convolutional layers with a filter size of 4×4 and stride of 2 before connecting with the text embedding.

In terms of training, as the input images are unique, we disable the text embedding augmentation of the generator when converting the synthesised image back to the input image (*i.e.,* the generator on the right-hand side of Figure 5.5). More specifically, LS-GAN [89] uses the squared error to replace the sigmoid cross entropy of DCGAN [89], has been successfully applied to CycleGAN [25]. Following CycleGAN, the loss functions become the following:

$$\begin{aligned}
\mathcal{L}_D &= \mathbb{E}_{(x,t)\sim p_{data}}(1 - D(x, \varphi(t)))^2 \\
&\quad + \mathbb{E}_{(x,\hat{t})\sim p_{data}} D(x, \varphi(\hat{t}))^2 \\
&\quad + \mathbb{E}_{(x,\bar{t})\sim p_{data}} D(G(x, \varphi(\bar{t})), \varphi(\bar{t}))^2 \\
\mathcal{L}_G &= \mathbb{E}_{(x,\bar{t})\sim p_{data}}(1 - D(G(x, \varphi(\bar{t})), \varphi(\bar{t})))^2 \\
&\quad + \mathbb{E}_{(x,\bar{t},t)\sim p_{data}} \lambda \times ||x - G(G(x, \varphi(\bar{t})), \varphi(t))||
\end{aligned} \tag{5.4}$$

where λ is the scale factor of the cycle loss. A larger λ results in maintaining more information of the input image in the synthesised image [25]. In extreme cases, if the λ is too large, the synthesised image will look the same as the input image (*i.e.,* failed translation). Compare with Equation (5.3),

we replace the *log* function by the squared error and an extra term for cycle loss is added to the generator loss $\mathcal{L}_G$.

5.4 Evaluation

In this section, we demonstrate that our method was capable of synthesising plausible images that maintained most of the features of the input images and matched the input text descriptions. To show our generator successfully learnt the features of both the image and text description, we interpolate the latent space of two images with a fixed text description and then interpolate two text descriptions with a fixed image.

5.4.1 Datasets and training details

Two single-category datasets, Caltech-200 bird [102] and Oxford-102 flower datasets [103], were used to evaluate the proposed method. Ten text descriptions for every image in both datasets were labelled by [136], mainly describing the colours and textures of different parts of the objects [24] (*e.g.,* bird head and belly). The bird dataset has 11,788 images with 200 classes of birds. We split 200 classes into 150 for training and 50 for testing. The flower dataset has 8,189 images with 102 classes of flowers, split into 82 for training and 20 for testing.

Both datasets only describe the same object (*i.e.,* either bird or flower). They are single-category datasets. Therefore, as Figure 5.4 shows, we combine the matching text t and mismatched text $\hat{t}$ as the semantically relevant text $\bar{t}$. To stabilise the GAN training, following DCGAN [19], to train the generator and discriminator, we adopted an initial learning rate of 0.0002, with the Adam optimisation [61] and momentum of 0.5. We used a batch size of 64 and trained the network for 600 epochs for both datasets. The learning rate was decayed by 0.5 every 100 epochs. For the image size of 256×256, the learning rate is decreased by 0.5 every 50 epochs. Due to the limits of hardware (*i.e.,* NVIDIA Titan XP GPU), we reduce the batch size to 16. For the generator with the pre-trained VGG encoder, the parameters of the VGG part and text encoder were fixed after pre-training. For data augmentation, we randomly horizontally flip (50%) and rotate ($\pm 15°$) the images in both datasets.

For synthesising 256×256 images, we tried different λ for the cycle loss, varying from 0.01 to 10. According to the experiment in CycleGAN, a large λ can help synthesise images to be more

similar to the input images but reduces the effect of the translation. The choice of λ is dependent on the datasets [25]. Similar to CycleGAN, we found the same phenomenon: a large λ will make the synthesised images ignore the text descriptions (*e.g.,* the text cannot modify the image well). In our experiment, the flower dataset works well for large λ (*e.g.,* 10) but the bird dataset does not. Finally, we chose a λ of 0.1, which works well for both datasets.

In this experiment, we will compare the baseline method, our 64×64 method with and without pre-trained VGG, and our 256×256 method with cycle loss. To speed up the development, GPU acceleration was used to train the network. Training the networks for 64×64 with VGG takes about two days using an NVIDIA Titan XP GPU. The training for 64×64 without VGG and 256×256 image size requires about one and three days, respectively. The code was implemented by TensorLayer 1.7.2rc [112] and TensorFlow 1.0.0 [113], and can be found in the Appendix.

5.4.2 Qualitative comparison

First, we compared the result of the proposed methods with and without using the pre-trained VGG, and the baseline method described in Section 5.2.3. We put the bird results from Reed *et al.* [24] in the left columns in Figure 5.6 to directly compare them with our results. (as the evaluation by Reed *et al.* [24] only demonstrated the results from the training set, we used their training set results to make a fair and direct comparison). Compared with the baseline method, the proposed method kept most of the original background, bird poses, and other information in the original images. For example, the fourth image on the left in Figure 5.6 shows that the baseline method failed to keep the tree branch on the synthesised image, while our method synthesised the tree branch. The quantitative comparison for the background and bird pose can be found in the next subsection. Moreover, our method successfully synthesised unseen birds from the training set. For example, it synthesised the flying birds in different colours that might not exist in the real world.

However, all the input images in Figure 5.6 are from the training set, which can give better results but cannot demonstrate the zero-shot learning ability of the proposed method. Then, we further evaluated the methods on the test set for both the bird and flower datasets and the effect of using pre-trained VGG to enhance the feature representation.

Figure 5.7 compared the baseline method and our method with and without VGG using the test set of birds with more complicated backgrounds. Compared with the baseline method, our birds are

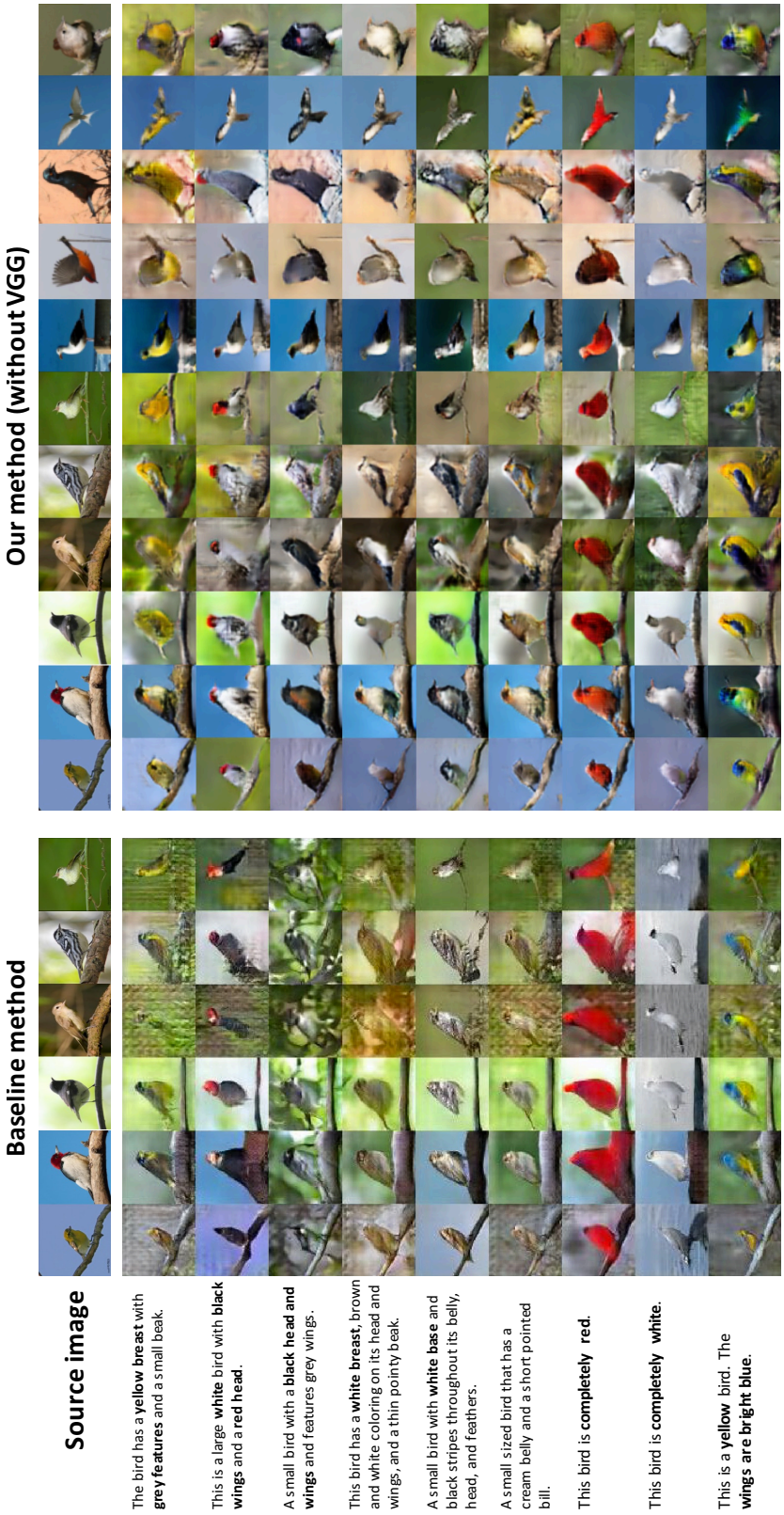

Figure 5.6: Example results of the baseline method and the proposed method without pre-trained VGG on Caltech-200 bird dataset. The baseline results from [24].

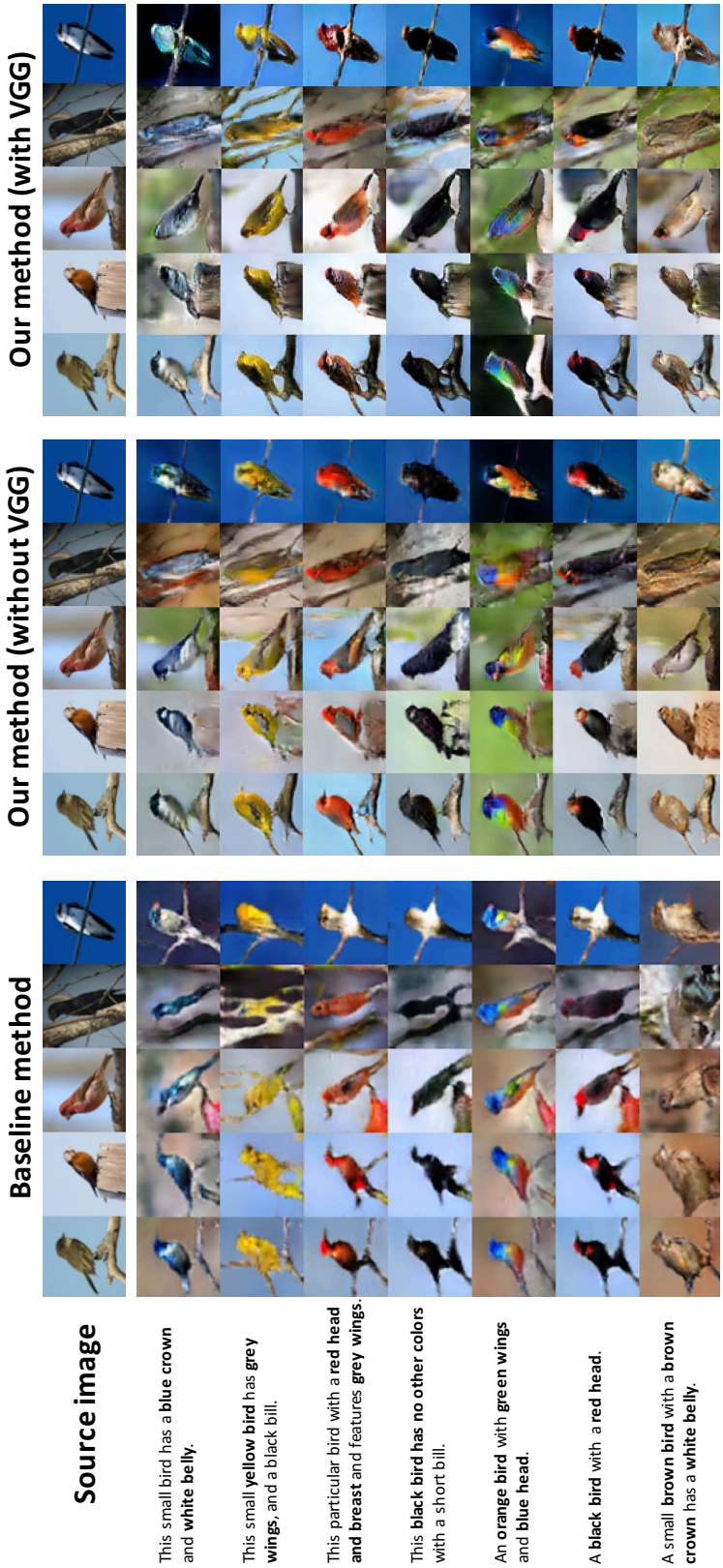

Figure 5.7: Zero-shot results of the baseline method and our method with and without pre-trained VGG encoder on Caltech-200 bird dataset.

not mixed up with the backgrounds. For example, in the right column, we can see that the cable overlapped with the bird can be clearly identified. The results also show that our method can keep more information from the input images. For example, the timber pile in the second column can be reconstructed. For the pre-trained VGG, compared with our method without VGG, the results have clearer backgrounds, such as the cable, timber pile, and branch. This is due to the pre-trained VGG encoder that already learnt the features of various objects from ImageNet, while, in this bird dataset, the number of different background objects is not large enough.

For the flower dataset, Figure 5.8 shows the results of the baseline method and our method without the pre-trained VGG. The baseline method failed to synthesise the details of the petals and outputs distorted images. For example, the third image from the left is a blue flower with five petals, but the synthesised images all failed to keep the shape of the petals. A possible reason for this might be because the flowers have many diverse shapes relatively, which are too difficult for the baseline method to encode such spatial information into a vector. In contrast, the proposed method used implicitly way to learn the encoder which outputting features of 3-dimension volume. Figure 5.9 illustrated our results with and without pre-trained VGG on the test set. The results show that our method successfully synthesised plausible images by modifying the input images based on text descriptions.

Our experiment showed that the generator without residual transformation unit was more likely to synthesise images that lost the features of the input images, *e.g.,* backgrounds, and lacked the details specified by the text descriptions. In addition, to generalise the proposed method, we further conducted an additional experiment on the combined dataset of birds and flowers as a multi-category dataset, employing the same model, training scheme and hyper-parameters. We found that our model can still be able to synthesise reasonable images, as shown in Figure 5.10. The method would be able to produce good results in more complicated and varied datasets with deeper network architecture.

5.4.3 Quantitative comparison

In this chapter, we used a human ranking method to quantitatively compare the different methods: 1) the baseline method, 2) our method without VGG, and 3) our method with VGG. We spread volunteer recruiting advertisements among undergraduate and graduate students in Imperial College London. To prevent bias, we select 10 volunteers who do not know our work to rank the quality of images synthesised by different criteria. We used all the images from test set for the evaluation. Each image

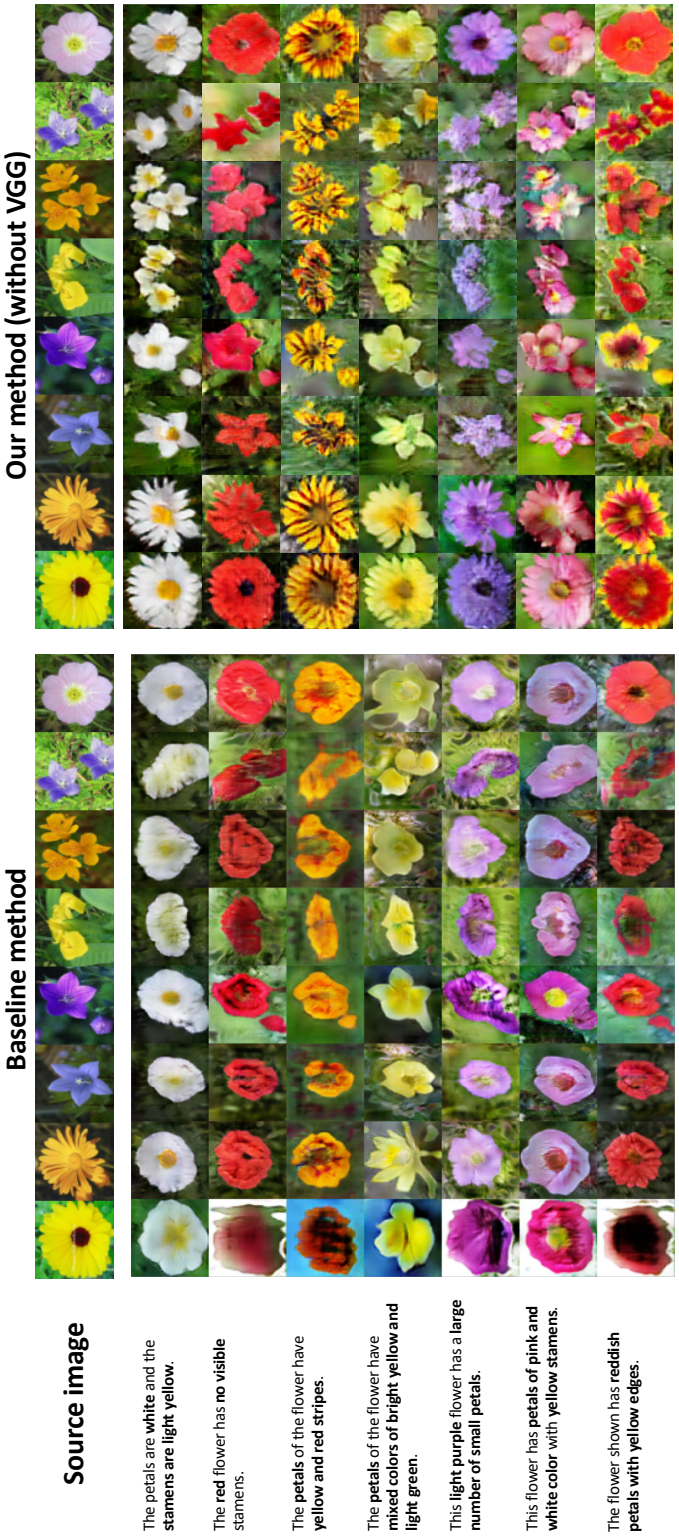

Figure 5.8: Zero-shot results of the baseline method and our method without pre-trained VGG on Oxford-102 flower dataset.

Figure 5.9: Zero-shot results of our method with and without pre-trained VGG encoder on Oxford-102 flower dataset.

Figure 5.10: Zero-shot results of our method without VGG on the combined dataset of Caltech-200 bird and Oxford-102 flower.

is translated by a text description, which is randomly sampled from the test set by assuming a uniform distribution over all the text descriptions in the test set. Then all results are divided into ten portions evenly. As we compared three methods, for every test data, three synthesised image, as well as the original image, were presented to volunteers. To prevent bias, images of different methods are shuffled by Fisher-Yates method [114], so that the participants do not know which image is corresponding to which method. The volunteers were required to rank the images, 1 for the best, 3 for the worst, based on the following three criteria, whether the synthesised image:

- maintains the original pose of the bird and shape of the flower;
- maintains the original background such as the tree branch and leaf;
- matches the text description and appears like real image;

		baseline	**ours**	**ours+VGG**
bird	pose	2.87	1.61	1.52
	background	2.68	1.93	1.39
	text	2.11	1.94	1.95
flower	shape	2.97	1.55	1.49
	background	2.62	1.74	1.64
	text	2.52	1.75	1.72

Table 5.1: Human evaluation of our approach, showing averaged rank scores of Caltech-200 bird dataset and Oxford-102 flower dataset for different aspects.

After that, we averaged the ranks of images from all volunteers to calculate the quality scores (1 for best, and 3 for worst) for all 3 methods. The reason of choosing these criteria to compare on is because the text description of both datasets only related to the colour and texture of the object without describing the background and the pose of the object [24]. Therefore, the original background and object pose should not be changed. In addition, our task requires the synthesised images to not only be able to look plausible but also able to match the text description. Table 5.1 shows the scores, it is clear that the proposed methods outperformed the baseline method on the following three aspects:

- **Maintaining the object pose and shape:** Compare with the baseline method, our methods with and without VGG can synthesise images maintain better bird poses and flower shapes. Because the baseline method failed to synthesise the details of flower petals as Figure 5.8 shows,

our methods have a significant improvement on maintaining the flower shape compare with the baseline method.

- **Maintaining the original background:** Our proposed methods can maintain the background better than the baseline method. For example, as Figure 5.7 shows background objects (*e.g.,* the tree branches and electric wire) of the baseline method are distorted and mixed up with the birds, our methods especially the method with VGG are able to reconstruct better background objects relatively.

- **Matching the text description and appearing like real image:** The ranking results indicate that our methods with and without VGG can synthesise plausible images matching text description better than the baseline method on both bird and flower datasets. As the first column images from left in Figure 5.8 shows, the baseline method tends to synthesise distorted images, so our methods can have a significant improvement on synthesising plausible flowers compare with the baseline method.

For our method with pre-trained VGG, the ranking indicates that it cannot make the text matching better than our method without using VGG. The reason is that it does not show significant improvement on the flower dataset (*i.e.,* from 1.75 to 1.72) and its performance for bird dataset is slightly reduced (*i.e.,* from 1.94 to 1.95). Nevertheless, the pre-trained VGG can help to generate better pose and background details compare with our method without using VGG. Especially, the background details of bird dataset have a better improvement than the flower dataset (*i.e.,* "1.93 to 1.39" vs "1.74 to 1.64"), this may because the bird dataset contains more background objects (*e.g.,* tree branches and other staffs that bird can stand on) than the flower dataset.

5.4.4 Interpolating latent space

To show the image and text encoders learnt the features meaningfully, we visualised the interpolating results for image and text-latent space individually. A meaningful latent space can be interpolated and can be used to produce smooth and linear latent representation [19]. For this reason, here we demonstrated whether the generator supports such interpolation.

- **Interpolating image latent space:** Figure 5.11 are the synthesised images from linearly interpolation between two different input images and a fixed text description. The images are

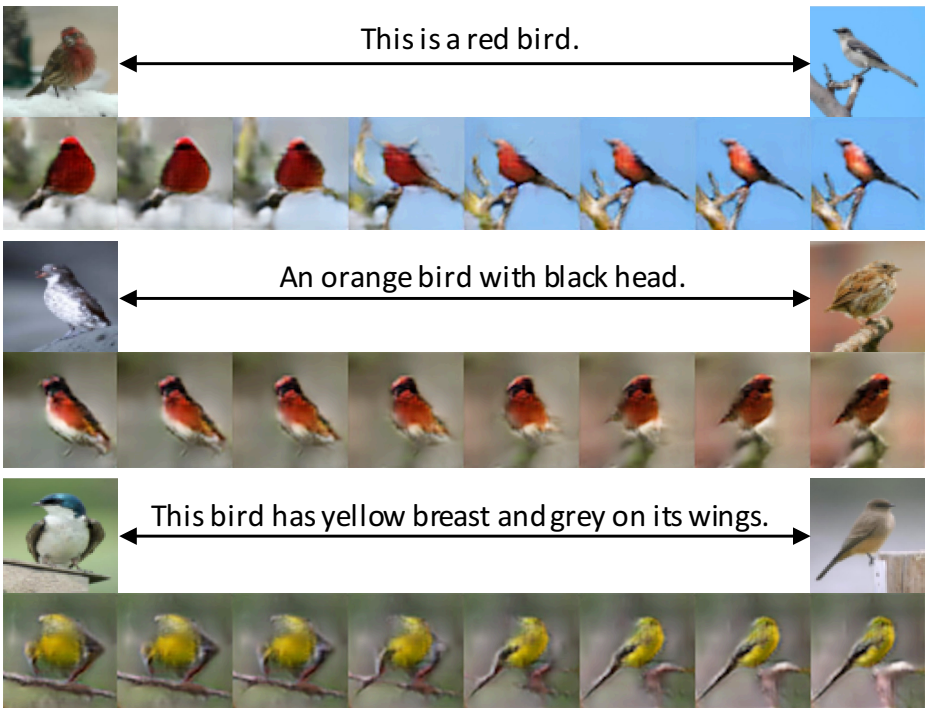

Figure 5.11: Zero-shot results of interpolation between two source images with the fixed text description. The images pointed by arrows are the input images.

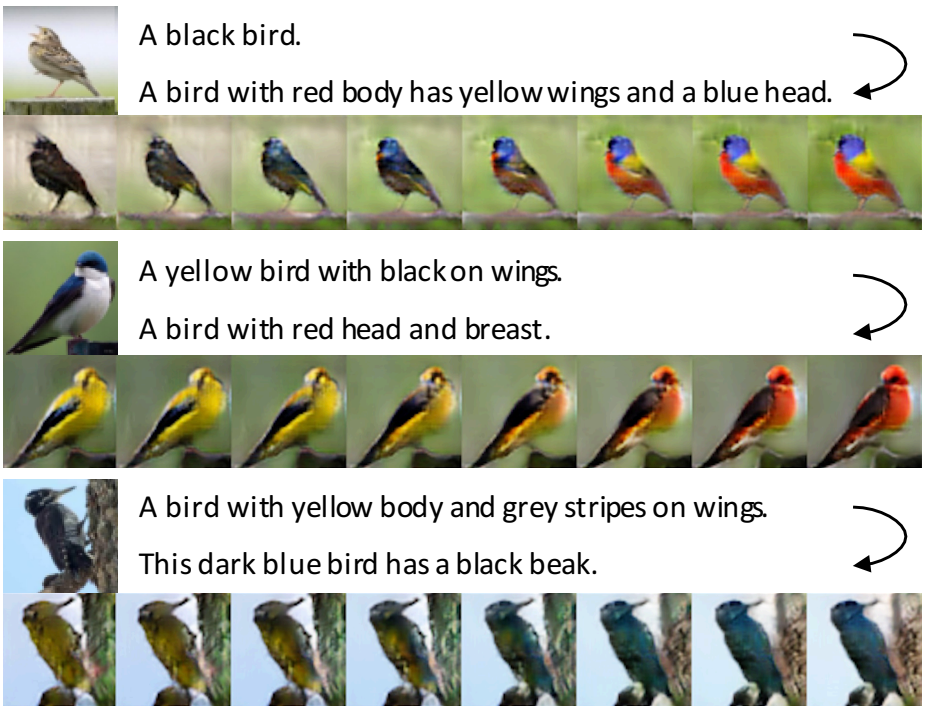

Figure 5.12: Zero-shot results of interpolation between two text descriptions for the same input image. The images on the left-hand side of sentences are the input images.

from the test set, and we applied the interpolation on the outputs of the image encoder. The results reflect the smooth changes in object shape while keeping the same colour and texture to match the text description. Take the second row for example, we can see that the bird poses changes from "facing left" to "facing right" smoothly. In addition, the background object also changes smoothly, the first row shows that the background changed from "snow" to "branch".

- **Interpolating text space:** Figure 5.12 are the synthesised images from two interpolated text embeddings on the output of text encoder. It demonstrated gradual changes in semantic meaning. Take the top row for example, the bird body changes from "black" to "dark blue" and finally, "colourful", while keeping plausible shapes and other details of the bird. Moreover, the background object remains unchanged.

These results demonstrated that we could interpolate both text and image latent space to manipulate the synthesised images smoothly. The encoders successfully learn the features of image and text description for this multi-modal task.

5.4.5 Diversity

Given an image and a text description, the proposed method can generate diverse results. As shown in Figure 5.13, the top row are the input images and text descriptions, the bottom row is the synthesised image. The result shows that given an input image with a single text description, the generator can synthesise different images that can match the text description and input image.

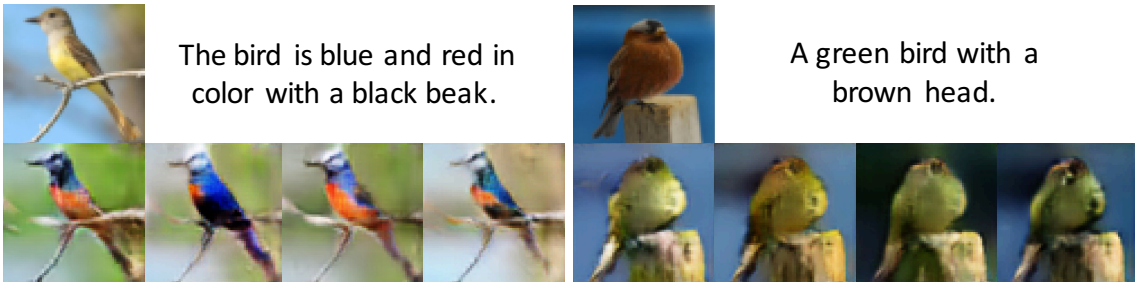

Figure 5.13: Zero-shot results from same input image and target text description for showing diversity.

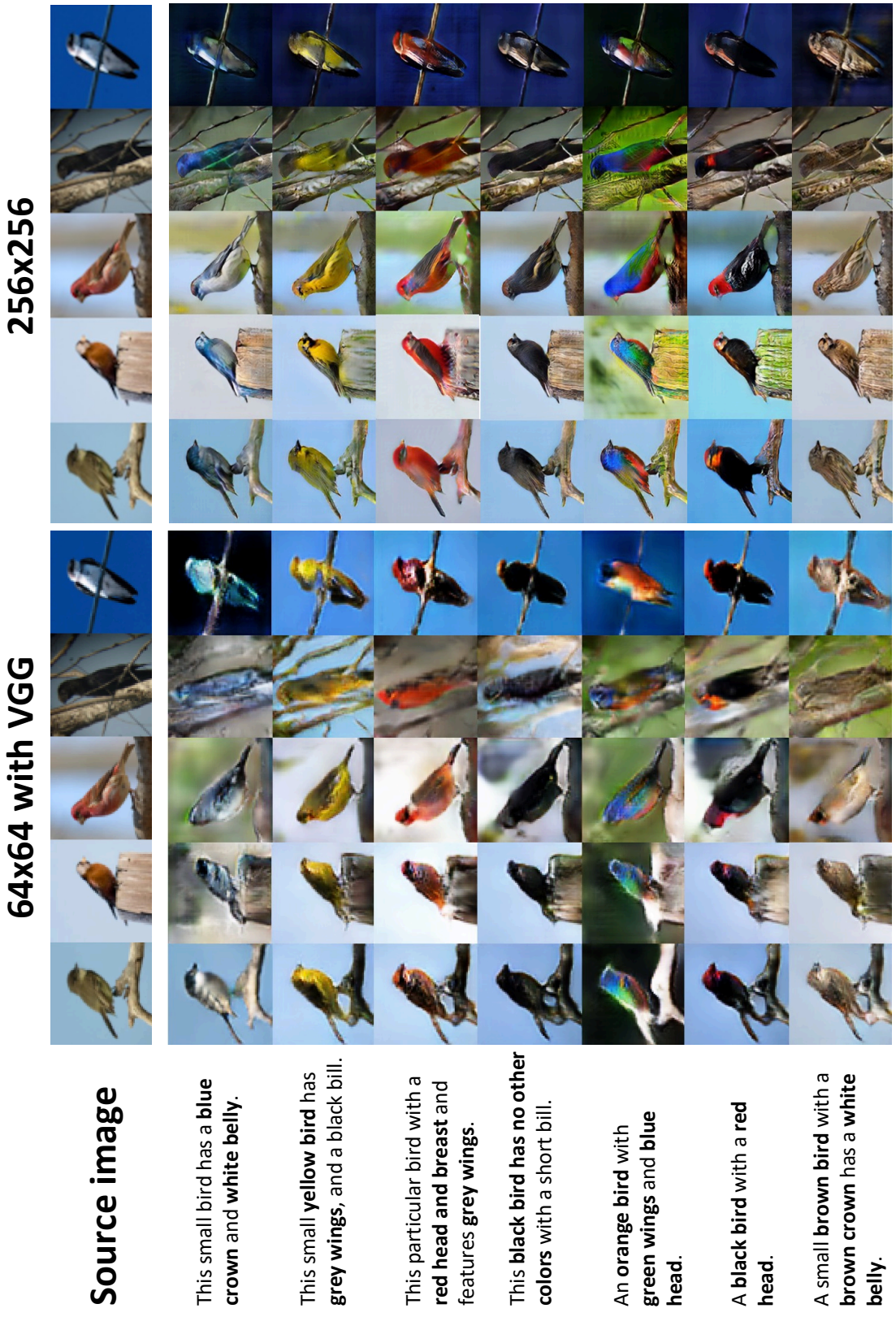

Figure 5.14: Zero-shot results of our 64 × 64 method with VGG and 256 × 256 method on Caltech-200 bird dataset.

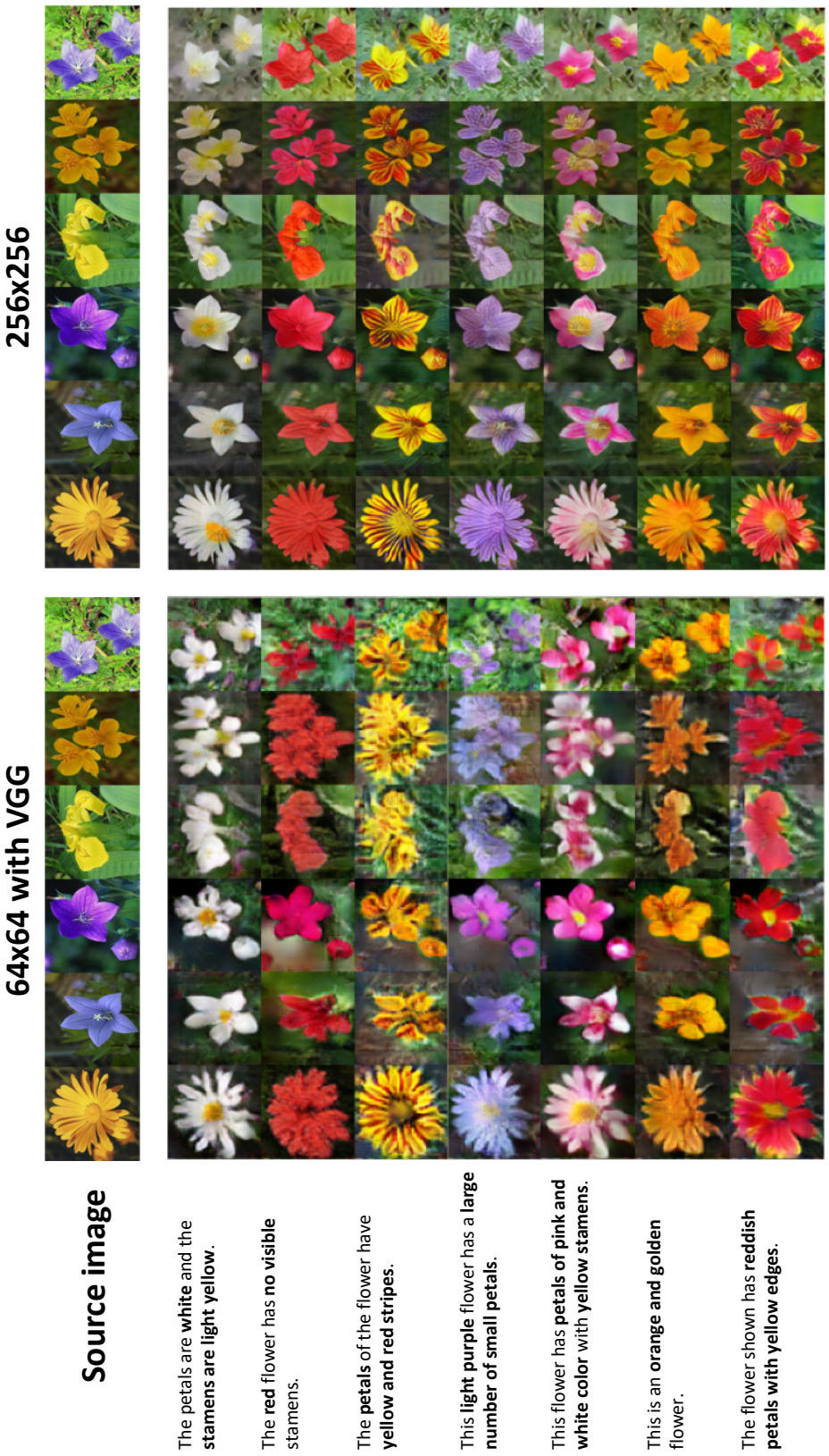

Figure 5.15: Zero-shot results of our 64 × 64 method with VGG and 256 × 256 method on Oxford-102 flower dataset.

Figure 5.16: The 256 × 256 flower results without using cycle loss.

5.4.6 Beyond 64 x 64

To compare the result of 256 × 256, we used the same input images and text description from Figure 5.7 and 5.9 and compare with the 64 × 64 method with VGG. Figure 5.16 illustrates the failed 256 × 256 flower examples when training the model without using cycle loss. Even though, the outline of flower can be determined, the shapes are totally distorted.

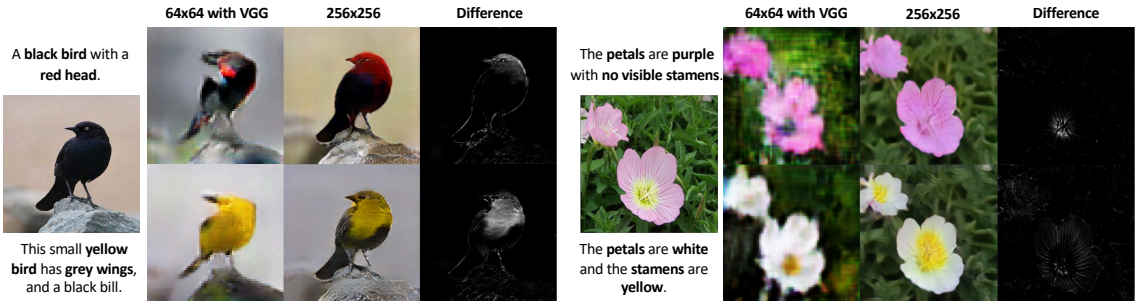

Figure 5.17: Bird and flower results. 64 × 64 vs 256 × 256.

- **High resolution image:** Figure 5.14, 5.15 and 5.17 illustrated the visual details of 64 × 64 (with VGG) and 256 × 256 images, we can see that by increasing the image resolution, the 256 × 256 images can have more visual details compare with 64 × 64 images. For example, the

first column on the left of Figure 5.15 shows that the long petals can have clear edges on the 256×256 images compare with the 64×64 images.

- **Better background reconstruction:** As discussed in Section 5.3.4, the cycle loss is able to "force" the synthesised images contain more information of the input images; otherwise, if the synthesised images do not contain sufficient information of the input images, we cannot translate the synthesised images back to the input images. As Figure 5.14 and 5.15 show, our 256×256 images have more background details information. For example, the second column from the left of Figure 5.14 shows that the timber pile can be better maintained compare with the 64×64 images. Another example is the fourth column from the left of Figure 5.15, the large leaves on the background cannot be maintained on the 64×64 images, but our 256×256 images successfully maintained the leaves.

As the synthesised images are able to keep the unrelated part of the image unchanged and manipulate the parts that the text required, subtracting the hue of input and synthesised images can help to understand the semantic connection between the image and text. In Figure 5.17, the images with black background are the subtraction between the hue of 256×256 input images on the left-hand side, and hue of the 256×256 synthesised images in the middle column. We can see that the colour information can link to the corresponding part of the image. Take the red-head bird on top-left for example, as the input image already is a black bird, then only the keyword "red head" in the text can affect to the synthesis.

5.5 Conclusions and Discussion

The semantic image synthesis in this chapter combines the advantages of the text descriptions and images in which the synthesised images contain the semantic visual information of the input images and the object attribute information of the input text descriptions. To solve the problem of the unknown ground truth images, a novel adversarial loss along with a novel model architecture is proposed. Instead of using supervised learning that requires the synthesised image to be a specific ground truth image, the proposed method learns to change the input image to match the text description by learning to fool the discriminator. By doing so, the propose method successfully achieves the semantic image synthesis using matching image-text pairs only.

The evaluations on Caltech-200 bird [102] and Oxford-102 flower datasets [103] show that the synthesised images can maintain the background and object shape information of the input images while matching the colour and texture information of the input text descriptions. The interpolation results show that the model successfully learns the features of images and text descriptions to produce smooth change linearly.

Chapter 6

Efficient Deep Learning Development

Apart from the algorithm, the issue of efficient deep learning development also need to be considered. In this chapter, we present the details of a deep learning library, called TensorLayer, for facilitating deep learning development. TensorLayer provides model abstraction to speed the building of neural network models as well as a life-cycle management tool to manage the dataset, model, and training pipeline. We evaluate TensorLayer by comparing it with other representative libraries.

6.1 Introduction

To facilitate deep learning development, existing computational engines, such as Theano [137] and TensorFlow [113], support automatic error back-propagation and GPU acceleration. These options provide basic mathematical operators, such as addition, subtraction, matrix multiplication, and convolution that allow for the construction and training of neural network models. However, deep learning models are becoming more sophisticated. First, neural network layers are becoming more complex from conventional layers, such as fully-connected and convolutional layers, to advanced architectures, such as a spatial transformer [138] for producing transformations of input data and DoReFa-net [139] for providing quantised feedforward propagation. Second, neural network models are becoming deeper as is examplified in image classification where Alexnet include only eight convolutional layers to ResNet that include a thousand convolutional layers [10]. This increasing complexity leads to the high workload of the library users when defining a model using basic mathematical operators.

To reduce users' workloads, since the neural network models are considered as the composition of

neural network layers, instead of using basic mathematical operations, wrapper libraries are built on top of the computational engines to provide model abstraction to helps build models using layers. Nonetheless, existing model abstractions, such as Keras and Pytorch, have limitations that require library users or library developers to deal with the output shape of layers, which increases the workload of using or developing the library. Also, to reduce users' workloads for using the pre-trained models, pre-trained model abstractions exist that help users easily reuse the state-of-the-art CNN models, such as VGG, MobileNet, and SqueezeNet. However, these existing pre-trained model abstractions cannot flexibly restore a certain part of the deep model based on different use cases of the pre-trained model, resulting in initialising unused layers that lead to unnecessary computer memory consumption.

Moreover, distinct from conventional software development that features a predefined development pipeline, deep learning development tends to revolve around experimentation. Researchers must experiment with different datasets, model architectures, and training methods repeatedly [2]. Therefore, apart from the workload from defining models, in practice, there are additional issues to be handled, including model and dataset storage for versioning, sharing, further retrieval and provenance [21], archiving the entire deep learning projects (*e.g.,* the model, dataset and training pipeline of one or multiple experiments) for further reproductions of the experiment, and training multiple models concurrently to speed hyper-parameter selection [2, 22]. Researchers also need to manage the entire deep learning workflow, including creating or acquiring the dataset, training the model based on the training pipeline, evaluating the model performance, and saving the model for further use. For these reasons, researches must spend extra effort in managing the model, dataset, and training pipeline. Life-cycle management tools such as AzureML and SeaHorse exist but require users to predefine each component of the training pipeline within a predefined template. This effort restricts algorithm development for models with architecture or connectivity that changes dynamically during training, such as GANs and neural module networks [140].

In this chapter, we present TensorLayer, a novel Python library, to facilitate deep learning research. Specifically, in terms of model abstraction, TensorLayer reduces the workload of users when defining models and the workload of library developers when implementing new layers. Also, TensorLayer's pre-trained model abstraction does not initialise unused layers and avoids unnecessary computer memory consumption. Moreover, TensorLayer provides a life-cycle management tool for managing the model, dataset, and training pipeline without restricting researchers' flexibility in defining training pipelines. We evaluate the model and pre-trained model abstractions of TensorLayer by comparing them with

other libraries released before and after TensorLayer. For life-cycle management, we demonstrate the efficacy of TensorLayer using two case studies of hyper-parameter selection and deep reinforcement learning followed by the comparison between TensorLayer and the existing tools. This library won the Best Open Source Software Award of ACM Multimedia (MM) 2017 [1].

6.2 Related Works

We begin with introducing tools for general machine learning development and why specific libraries for deep learning are required. Next, we describe the computational engines, model abstraction, pre-trained model abstraction, and life-cycle management for deep learning development.

6.2.1 Machine learning development libraries

Many open source libraries exist for general machine learning development. For example, scikit-learn [141] is a well-known library that provides simple and efficient tools for many state-of-the-art classifications, regression and clustering methods, such as support vector machine (SVM) [142], nearest neighbour [143], random forest (RF) [144], k-means clustering [145] as well as data analysis and processing methods like principal component analysis (PCA) [146] and normalisation. Two well-known libraries for natural language processing are Gensim [147] and NLTK [148], which are designed to extract semantic topics from documents, process unstructured digital texts, and perform word embedding [53]. Other general machine learning libraries, such as Mlpack [149] and Shogun [150], can reduce developers' workloads.

While these tools support the implementation of the machine learning algorithms, with increasing sizes of datasets, additional tools may be required to address challenges involving storage and computation. Hadoop [151] is a widely used storage engine that stores and replicates data on thousands of servers and provides simple APIs to read and write data. For processing data on Hadoop, developers leverage Spark [152] to execute complex and high-scale computation jobs. Spark parallelises computation through a dataflow approach and uses resilient distributed datasets (RDD) to cache intermediate results, leading to high performance.

However, Spark often assumes data is processed singularly, so it is difficult to support deep learn-

[1] `http://www.acmmm.org/2017/mm-2017-awardees/`

ing and other machine learning algorithms, such as logistic regression and k-means, which require iteratively training a dataset. This limitation motivated the development of Spark MLlib [153] and SystemML [154], which realise iterative machine learning computations based on the Spark engine. While these do support many machine learning algorithms, such as k-means and nearest neighbour, they cannot support deep learning because it is usually based on tensor operations, such as matrix multiplication and convolution [2].

6.2.2 Deep learning computational engines

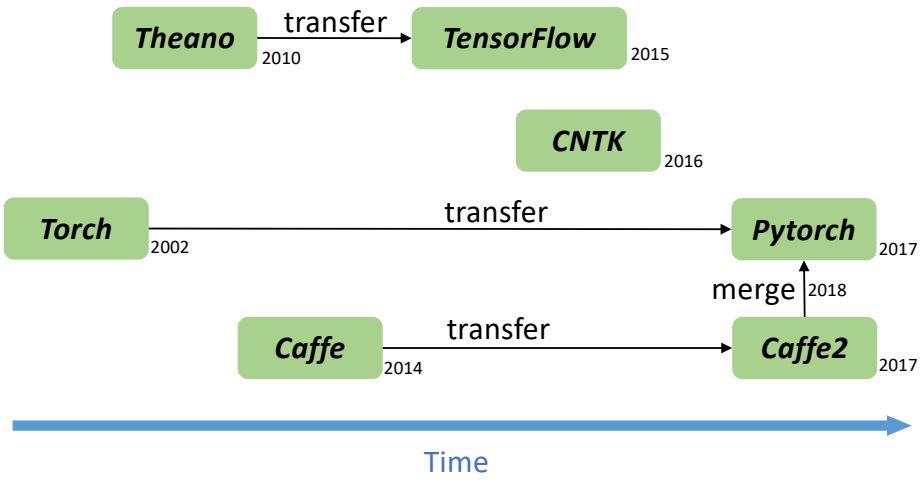

Figure 6.1: Deep learning computational engines by release time and history listed on Github.

Compared with conventional machine learning tasks that can be efficiently executed on a CPU, deep learning heavily relies a GPU to perform the high-dimension tensor computation. This technical requirement motivated deep learning computational engines, such as Theano [137], TensorFlow [113], CNTK [155], Caffe [156], Torch [157], Mxnet [158], and Pytorch [23], to automatically compute the gradient for error back-propagation, which is a crucial processing component in deep learning training.

The "Convolutional Architecture for Fast Feature Embedding" (Caffe) developed by UC Berkeley [156] provides a computer vision model framework with convolutional networks playing an essential role in its deep learning methods. For training a model with Caffe, developers simply write a model file that defines its architecture along with a solver file that defines the training details, including the input data, loss, learning rate, and other configurations necessary for training. Customisation of this frameworks operations requires developers to program in C++.

However, more flexible ways are need for developing general deep learning models. For complicated training processes (*e.g.,* GAN), defining a loss function in a solver file is not sufficient. To support more flexible deep learning development, the Montreal Institute for Learning Algorithms developed Theano [137] as a numerical computation engines based on Python. This library provides basic mathematical operators, such as addition, subtraction, and matrix multiplication to allow users to define a static graph that can be compiled to run. In 2015, by collaborating with the Theano team, Google developed TensorFlow [113] as the successor of Theano to support additional hardware, operational systems, and programming languages.

Facebook developed a general deep learning engine, called Torch, based on the Lua programming language followed by Pytorch as its successor in 2017 to attract more Python users [23]. Evolved from the static graph of Theano, Pytorch supports dynamic graphs run without compiling, which is an advantage useful for developing complex models (*e.g.,* models that are dynamically composed by several sub-models [140]). Later, Facebook merged Caffe2 into the backend of Pytorch in 2018.

Table 6.1 compares available deep learning computational engines with the number of Pypi downloads reported by Pepy Tech [2]. This comparison shows that TensorFlow has a much larger number of downloads and Github stars than the other engines, the success of which is due to the following three reasons:

- Google developed Tensor Processing Unit (TPU) to run TensorFlow, which is more economical than GPUs. In contrast, other engines cannot leverage customised hardware.
- TensorFlow supports many programming languages, such as Python, C++, Go, Java, and JavaScript, to provide friendly interfaces for different target users.
- TensorFlow can run on more operating systems, including Linux, macOS, Windows, Android, and iOS as well as a web page, which enables models to be run on more devices.

TensorFlow and Pytorch rank as the top two active deep learning engines. However, many other computational engines exist that have a smaller user base, such as Mxnet, CNTK, Chainer, BigDL, Darknet, DyNet, and OpenNN, which all support automatic error back-propagation. These engines help make deep learning development more efficient and facilitate deep learning into practical use.

[2] `https://pepy.tech`

	Release Year	Creator	Downloads from Pypi [a]	Github Stars	Platform	Interface	Mode	Hardware
TensorFlow	2015	Google Brain	22,748,848	121,491	Linux, macOS, Windows, Android, iOS, Javascript	Python, C, C++, Java, Go, R Javascript, R, Julia, Swift	Graph, Dynamic	GPU, CPU, TPU
Theano	2010	University of Montreal	9,325,223	8,699	Linux, macOS, Windows	Python	Graph	GPU, CPU
Pytorch	2017	Facebook	220,455	25,247	Linux, macOS, Windows	Python	Dynamic	GPU, CPU
Torch	2002	Facebook	2,511,535	8,214	Linux, macOS, Windows, Android, iOS	Lua, C, C++	Dynamic	GPU, CPU
Caffe2	2017	Facebook	10,066	8,420	Linux, macOS, Windows, Android, iOS	C++, Python	Graph	GPU, CPU
Caffe	2014	UC Berkeley	1,373 [b]	27,166	Linux, macOS, Windows	C++, Matlab, Python	Graph	GPU, CPU
CNTK	2016	Microsoft	201,659	15,838	Linux, Windows	Python, C++	Graph	GPU, CPU

Table 6.1: A comparison of deep learning computational engines.

[a] `https://pypi.org`

[b] Caffe has a relatively small number of downloads is because users typically install from its homepage instead of through Pypi.

6.2.3 Deep learning model abstraction

Even though TensorFlow has the most extensive user base and provides basic mathematical operators and management components, such as graphs, sessions, queue runners, and devices, proper use of these components requires users to have multi-fold expertise in deep learning and computer systems. However, users often interpret deep learning approaches as a collection of neural networks, layers and tensors, and they often focus more on the designs of their algorithms instead of the low-level system details. To build a model from scratch using TensorFlow, a developer must write non-trivial details for deep learning models, such as initialising tensors and applying different basic computational operators. Leveraging a model abstraction can help users easily develop deep models layer-by-layer. Abstraction can also assist in better management of the model, such as storing and restoring parameters of the entire model instead of individually. Taking this model abstraction approach exposes only abstracted components of the model while masking the details of the implementation from the users.

Since the boom of deep learning, many wrapper libraries have been built on top of deep learning computational engines for providing model abstraction to help users define models layer-by-layer. By far, Keras has the largest user base [3] as it supports three different computational engines as its backend, including Theano, CNTK, and TensorFlow. On the other hand, while Pytorch is a computational engine, it also provides a model abstraction. Other libraries exist that have smaller user bases [3], such as Prettytensor that focuses on basic layers only, and Sonnet for building complex models.

Keras and Pytorch are two of the most representative libraries. The former was released and maintained by Google for providing a model abstraction for TensorFlow, and the latter was released and maintained by Facebook for providing basic mathematical operators and model abstraction. Even though Pytorch was released after our TensorLayer, we compare TensorLayer to Keras and Pytorch along with a description of how Keras and Pytorch provide model abstraction and existing issues.

When build a model layer-by-layer, a challenge is that the parameter shape of a new layer is related to the output shape of its previous layer (*i.e.,* the input shape of the new layer). Considering the fully-connected layer represented by Equation 6.1 as an example, given the number of output values of the new layer n_units (*i.e.,* the number of neural units) and the number of output values of the previous layer n_units_{pre}, the shape of the bias vector b is $1 \times n\_units$. This shape only relates to the number

[3] `http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture8.pdf`

of output values of the new layer, but the shape of the weight matrix W is $n\_units_{pre} \times n\_units$, which relates to the number of output values of the new and previous layers.

$$a = f(x * W + b) \tag{6.1}$$

where x is the input, W is the weight matrix, b is the bias vector, f is the activation function, and a is the activation output.

Algorithm 7 The initialisation process of a layer in Pytorch.

Input: the layer setting *setting*, the forward function $forward$, the input shape *input_shape*

1: $params \leftarrow setting, input\_shape$; initialise the parameters
2: $new\_layer \leftarrow forward, params$; get the new layer
3: **return** *new_layer*

Algorithm 8 The initialisation process of a layer in Keras.

Input: the previous layer *layer*, the layer setting *setting*, the forward function $forward$

1: $input\_shape \leftarrow layer$; get the input shape from the previous layer
2: $params \leftarrow setting, input\_shape$; initialise the parameters
3: $output\_shape \leftarrow setting, input\_shape$; compute the output shape
4: $new\_layer \leftarrow layer, forward, params$; get the new layer
5: $new\_layer \leftarrow output\_shape$; put the output shape into the layer object
6: **return** *new_layer*

Therefore, to initialise a new fully connected layer, we must pass the output shape of the previous layer into the new layer. In addition to the fully connected layer, many other layers exist, such as convolutional and deconvolutional layers that require the output shape of the previous layer. For example, in a convolution layer, the filter shape of the new layer is related to the number of the channel of the previous layer [43].

Algorithm 7 shows the layer initialisation process for implementing a layer in Pytorch. The inputs include: (1) the layer settings, such as the number of units of the fully connected layer and the strides, filter size, and padding of the convolutional layer as provided by the users, (2) the forward function, such as the convolution and matrix multiplication, which is defined inside the layers, and (3) the input shape, which is the output shape of the previous layer as provided by the users. The initialisation process first sets the parameter using the input shape and layer settings provided by the user (line 1), and then the process uses the new parameter and forward function to form the new layer (line 2). The drawback of this approach is that users must manually feed the input shape into a layer when initialising. Specifically, considering the fully connected layer as an example in Figure 6.2, Pytorch requires the user to input the number of units of the previous layer to a new layer (denoted by

Keras

```keras
import keras
from keras import backend as K
from keras.layers import Layer

class Dense(Layer):
    def __init__(self, n_units, act=None):
        self.n_units = n_units
        super(Dense, self).__init__()

    def build(self, input_shape):
        # create new parameters for this layer according to the output shape of
        # previous layer (i.e., the input shape of current layer)
        self.W = self.add_weight(name='W', shape=(input_shape[1], self.n_units))
        self.b = self.add_weight(name='b',shape=(self.n_units))
        super(Dense, self).build(input_shape)  # be sure to call this at the end

    def call(self, x):
        # tensor computation
        outputs = K.dot(x, self.W) + self.b
        if self.act is not None:
            return self.act(outputs)
        else:
            return outputs

    def compute_output_shape(self, input_shape):
        # manually compute the output shape of this layer, it will be used in
        # the build function of the next layer
        return (input_shape[0], self.n_units)
```

Pytorch

```pytorch
import torch
import torch.nn as nn

class Dense(nn.Module):
    def __init__(self, in_units, out_units, act=nn.ReLU):
        super(Linear, self).__init__()
        self.act = act
        # create new parameters
        self.W = nn.Parameter(torch.Tensor(in_units, out_units))
        self.bias = nn.Parameter(torch.Tensor(out_units))

    def forward(self, input):
        # tensor computation
        outputs = torch.matmul(input, self.W) + self.b
        if self.act is not None:
            return self.act(outputs)
        else:
            return outputs
```

Figure 6.2: Implementing the fully connected layer with Keras and Pytorch.

in_units of line 5 in Pytorch), and then the new layer internally applies this number to initialise the weight matrix (line 9 of Pytorch). Next, the initialised weight matrix can be used in the feedforward propagation (lines 14-18 in Pytorch) where "matmul" represents the matrix multiplication, and *act* is the activation function. To use this fully connected layer, as shown in lines 9 to 11 in Pytorch of Figure 6.3, the Pytorch model first initialises all fully connected layers using the number of units of the previous and layer settings provided by the users. The first argument of all fully connected layers is the number of outputs of the previous layers. The Pytorch model next uses these initialised layers in the feedforward propagation as shown in lines 12 through 19 and 21. The fully connected layer is one example that needs the output shape of the previous layer as many other layers also require this, such as the convolutional and deconvolutional, sub-pixel, pooling, batch normalisation, and RNNs [2]. Therefore, Pytorchs model abstraction increases the users workload for defining a model using existing

Keras

```
01. import keras
02. nin = keras.layers.Input((None, 784))
03. net = keras.layers.Dropout(0.2)(nin)
04. net = keras.layers.Dense(800, activation='relu')(net)
05. net = keras.layers.Dropout(0.5)(net)
06. net = keras.layers.Dense(800, activation='relu')(net)
07. net = keras.layers.Dropout(0.5)(net)
08. net = keras.layers.Dense(10)(net)
09. model = keras.Model(nin, net)
10. x = tf.placeholder(tf.float32, shape=[None, 784], name='x')
11. outputs = model(x)
```

Pytorch

```
01. import torch
02. import torch.nn as nn
03. class MLP(torch.nn.Module):
04.     def __init__(self):
05.         super(MLP, self).__init__()
06.         self.drop1 = nn.Dropout(p=0.2)
07.         self.drop2 = nn.Dropout(p=0.5)
08.         self.drop3 = nn.Dropout(p=0.5)
09.         self.dense1 = nn.Linear(784, 800)
10.         self.dense2 = nn.Linear(800, 800)
11.         self.dense3 = nn.Linear(800, 10)
12.     def forward(self, x):
13.         x = self.drop1(x)
14.         x = self.dense1(x)
15.         x = self.drop2(x)
16.         x = self.dense2(x)
17.         x = self.drop3(x)
18.         x = self.dense3(x)
19.         return x
20. model = MLP()
21. outputs = model(x)
```

Figure 6.3: Defining an MLP model using Keras and Pytorch. The MLP has two hidden layers with 800 units and ReLU activation, the input has 784 values, and the output has 10 values. Three dropout layers are inserted between each layer. "Dense" and "Linear" denote the fully connected layer, and "Dropout" denotes the dropout layer.

layers.

Algorithm 8 shows the layer initialisation process for Keras. To avoid requiring users to manually feed the output shape of the previous layer into the new layer, distinct from the Pytorch approach, the input shape is provided by the previous layer (line 1), and then the new layer uses the input shape and layer settings to initialise the parameters (line 2). This approach requires Keras to compute the output shape inside the layer (line 3) to be used as the input shape of the next layer (lines 5-6). Considering again the fully connected layer as an example, lines 25 to 28 of Keras in Figure 6.2 obtain the output shape of the current layer for initialising the parameters of the next layer (lines 10-15). Next, the initialised parameters are used in the feedforward propagation (lines 17-23 of Keras) where the "dot" represents matrix multiplication, and the *act* is the activation function. The top of Figure 6.3 shows how to use the fully connected layer in Keras, as all layers are stacked one-by-one directly without requiring the users to feed in the output shape of the previous layer manually.

Even though Keras eliminates the need for users to provide the input shape when initialising new layers, library developers must implement the functions to compute the output shape of every layer (*i.e.,* "compute_output_shape" in Figure 6.2). The fully connected layer is a simple example where the output shape of the current layer is equal to the number of units. However, the implementation of many other layers is more complex. For example, the output shapes of convolution and deconvolution are associated with many factors including the input shape, filter size, stride, padding, and the number of channels [43]. Therefore, Kerass model abstraction increases the workload for library developers when implementing new layers to extend the library.

Comparing with Pytorch and Keras, TensorLayers model abstraction requires neither users to manually calculate and feed the input shape into new layers when initialising nor library developers to implement the function to compute the output shape of the layers, thereby reducing the workload for both library users and developers.

6.2.4 Deep learning pre-trained model abstraction

In deep learning, computer vision and generative tasks typically leverage the standard CNN models such as VGG [45], MobileNet [46], and SqueezeNet [47] for encoding images into latent representations. Moreover, pre-training the image encoder on a large dataset, such as ImageNet [4], improves the deep learning performance without labelling more data [1, 34].

In recent years, the development of state-of-the-art CNN models became more complex ranging from Alexnet [3] with eight layers and VGG16 [45] with 19 layers (*i.e.,* 16 convolutional layers and three fully connected layers) to the recent ResNet with a thousand convolutional layers [10]. To implement these models, library users must define the model layer-by-layer and set each correctly. For example, to define the entire VGG16 model shown as "A" of Figure 6.5, users define the model layer-by-layer as shown in Figure 6.4.

Provide the pre-trained model abstraction for the standard pre-trained CNN models assists in facilitating the development and simplification of the implementation. Instead of defining the model layer-by-layer, the pre-trained model abstraction can allow users to define the entire model as well as download and restore the parameters easily.

The pre-trained CNN models can be abstracted into two sub-models of a feature extractor and a classifier [10, 43, 45, 46]. The feature extractor identifies the image feature, and the classifier uses these

TensorLayer

```python
import tensorflow as tf
import tensorlayer as tl
from tensorlayer.layers import *
x = tf.placeholder(tf.float32, [None, 224, 224, 3])
net_in = InputLayer(x, name='input')
net = Conv2d(net_in, 64, (3, 3), (1, 1), tf.nn.relu, name='conv1_1')
net = Conv2d(net, 64, (3, 3), (1, 1), tf.nn.relu, name='conv1_2')
net = MaxPool2d(net, (2, 2), (2, 2), name='pool1')
net = Conv2d(net, 128, (3, 3), (1, 1), tf.nn.relu, name='conv2_1')
net = Conv2d(net, 128, (3, 3), (1, 1), tf.nn.relu, name='conv2_2')
net = MaxPool2d(net, (2, 2), (2, 2), name='pool2')
net = Conv2d(net, 256, (3, 3), (1, 1), tf.nn.relu, name='conv3_1')
net = Conv2d(net, 256, (3, 3), (1, 1), tf.nn.relu, name='conv3_2')
net = Conv2d(net, 256, (3, 3), (1, 1), tf.nn.relu, name='conv3_3')
net = MaxPool2d(net, (2, 2), (2, 2), name='pool3')
net = Conv2d(net, 512, (3, 3), (1, 1), tf.nn.relu, name='conv4_1')
net = Conv2d(net, 512, (3, 3), (1, 1), tf.nn.relu, name='conv4_2')
net = Conv2d(net, 512, (3, 3), (1, 1), tf.nn.relu, name='conv4_3')
net = MaxPool2d(net, (2, 2), (2, 2), name='pool4')
net = Conv2d(net, 512, (3, 3), (1, 1), tf.nn.relu, name='conv5_1')
net = Conv2d(net, 512, (3, 3), (1, 1), tf.nn.relu, name='conv5_2')
net = Conv2d(net, 512, (3, 3), (1, 1), tf.nn.relu, name='conv5_3')
net = MaxPool2d(net, (2, 2), (2, 2), name='pool5')
net = FlattenLayer(net, name='flatten')
net = DenseLayer(net, 4096, tf.nn.relu, name='fc1_relu')
net = DenseLayer(net, 4096, tf.nn.relu, name='fc2_relu')
net = DenseLayer(net, 1000, name='outputs')
outputs = net.outputs
```

Keras

```python
import tensorflow as tf
import keras
from keras.layers import *
net_in = Input(shape=(224, 224, 3), name='input')
net = Conv2D(64, (3, 3), strides=(1, 1), activation='relu', padding='valid', name='conv1_1')(net_in)
net = Conv2D(64, (3, 3), strides=(1, 1), activation='relu', padding='valid',name='conv1_2')(net)
net = MaxPool2D((2, 2), strides=(2, 2), name='pool1')(net)
net = Conv2D(128, (3, 3), strides=(1, 1), activation='relu', padding='valid',name='conv2_1')(net)
net = Conv2D(128, (3, 3), strides=(1, 1), activation='relu', padding='valid',name='conv2_2')(net)
net = MaxPool2D((2, 2), strides=(2, 2), name='pool2')(net)
net = Conv2D(256, (3, 3), strides=(1, 1), activation='relu', padding='valid',name='conv3_1')(net)
net = Conv2D(256, (3, 3), strides=(1, 1), activation='relu', padding='valid',name='conv3_2')(net)
net = Conv2D(256, (3, 3), strides=(1, 1), activation='relu', padding='valid',name='conv3_3')(net)
net = MaxPool2D((2, 2), (2, 2), name='pool3')(net)
net = Conv2D(512, (3, 3), strides=(1, 1), activation='relu', padding='valid',name='conv4_1')(net)
net = Conv2D(512, (3, 3), strides=(1, 1), activation='relu', padding='valid',name='conv4_2')(net)
net = Conv2D(512, (3, 3), strides=(1, 1), activation='relu', padding='valid',name='conv4_3')(net)
net = MaxPool2D((2, 2), (2, 2), name='pool4')(net)
net = Conv2D(512, (3, 3), strides=(1, 1), activation='relu', padding='valid',name='conv5_1')(net)
net = Conv2D(512, (3, 3), strides=(1, 1), activation='relu', padding='valid',name='conv5_2')(net)
net = Conv2D(512, (3, 3), strides=(1, 1), activation='relu', padding='valid',name='conv5_3')(net)
net = MaxPool2D((2, 2), (2, 2), name='pool5')(net)
net = Flatten(name='flatten')(net)
net = Dense(units=4096, activation='relu', name='fc1_relu')(net)
net = Dense(units=4096, activation='relu', name='fc2_relu')(net)
net = Dense(units=1000, name='outputs')(net)
model = keras.Model(net_in, net)

x = tf.placeholder(tf.float32, [None, 224, 224, 3])
outputs = model(x)
```

Figure 6.4: Defining the complete VGG16 layer-by-layer using TensorLayer and Keras' model abstraction.

Algorithm 9 The initialisation process of a pre-trained model in Pytorch.

Input: the setting of feature extractor $setting_f$, , the setting of classifier $setting_c$

1: $cnn \leftarrow setting_f$; initialise the feature extractor
2: $classifier \leftarrow setting_c$; initialise the classifier
3: $model \leftarrow classifier(cnn)$; stack the classifier on top of the feature extractor as the return model
4: **return** $model$, cnn, $classifier$; return the model

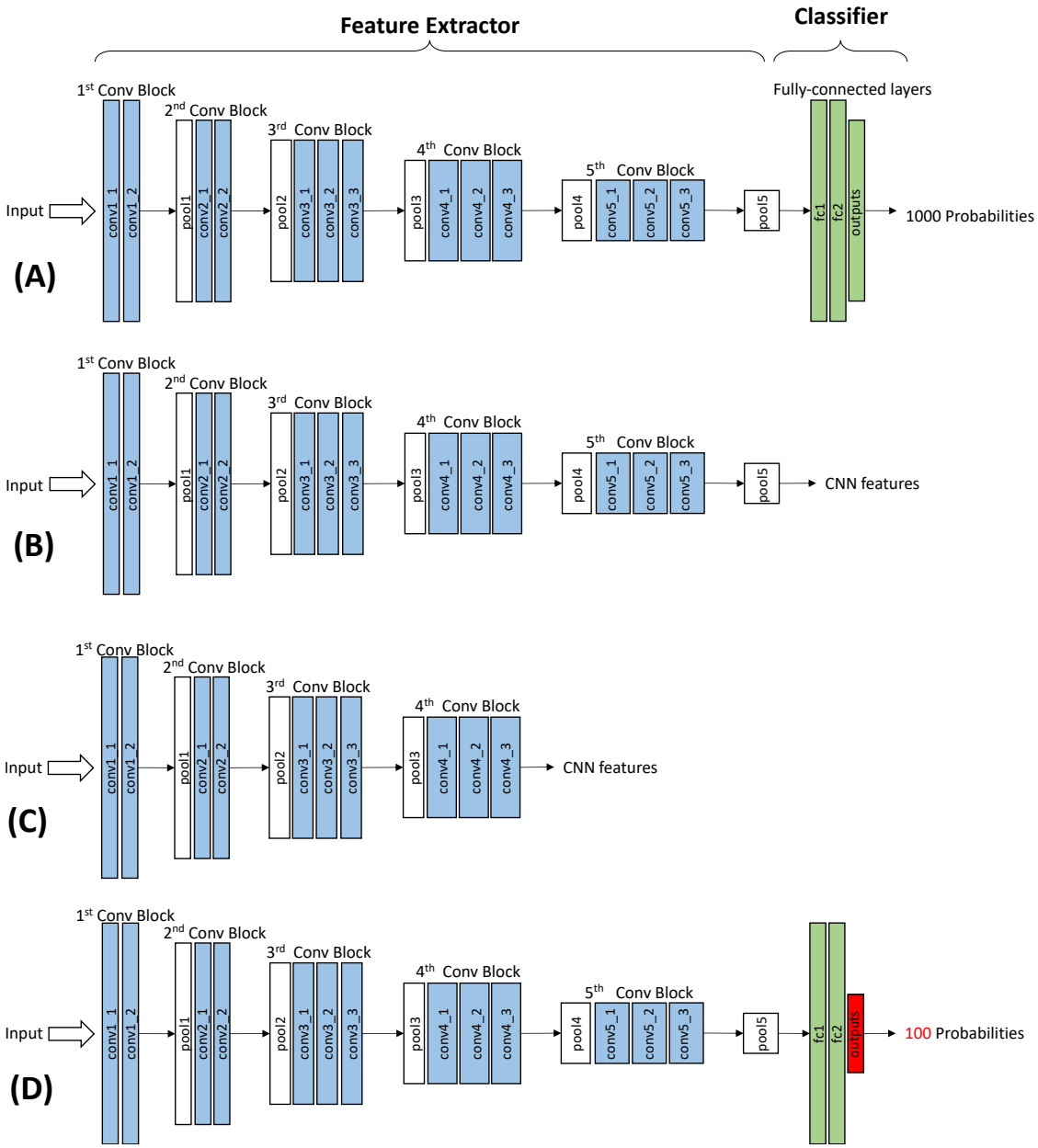

Figure 6.5: Applying the pre-trained VGG16 model in different ways. (A) The entire VGG16 model. (B) Using the entire CNN portion as the feature extractor. (C) Using the output of the fourth convolutional block as the extracted features. (D) Replacing the last fully connected layer of VGG16 for 100 classes classification. For the layer name, "pool" denotes the max pooling layer, "fc" denotes the fully connected layer, and "conv" denotes convolutional layer.

Algorithm 10 The initialisation process of a pre-trained model in Keras.

Input: endpoint marker *include_top*, the setting of feature extractor $setting_f$, , the setting of classifier $setting_c$

1: $cnn \leftarrow setting_f$; initialise the feature extractor
2: **if** $include\_top == True$ **then**
3: $classifier \leftarrow setting_c$; initialise the classifier
4: $model \leftarrow classifier(cnn)$; stack the classifier on top of feature extractor as the return model
5: **else**
6: $model \leftarrow cnn$; use the feature extractor as the return model
7: **end if**
8: **return** *model*

for predicting the probabilities of different classes (*e.g.,* 1,000 classes for ImageNet). Considering VGG16 on the top of Figure 6.5 as an example, all convolutional layers belong to the feature extractor, and all fully connected layers belong to the MLP classifier [45]. The output of the feature extractor is denoted by "pool5", which is the fifth convolutional block after pooling. In practice, the pre-trained CNN models are used for classification as well as the output extracted features from the feature extractor or the change in the number of outputs for classifying different datasets.

Figure 6.5 shows four representative use cases for using a pre-trained CNN model. The "A" model is the case that uses the entire model. The "B" model removes the classifier and uses the output of feature extractor as the extracted image features. Similarly, the "C" model uses the output of the hidden layers of the feature extractor as the extracted image features. These extracted image features can be further fed into other neural network models to achieve specialised tasks. For example, supervised image super-resolution [29] uses the pre-trained VGG to extract the image features of the output images and ground truth images to learn to minimise the MSE between their image features for reconstructing the visual details better. Furthermore, state-of-the-art human pose estimation [34] and object detection [44] use the pre-trained VGG to extract the features of the input image to be fed into the subsequent networks for outputting the pose key-point coordinates or bounding boxes, respectively.

Moreover, the pre-trained CNN could be used for classifying 1,000 classes for ImageNet as well as changing the number of outputs for classifying different datasets with a different number of classes. For example, the "D" model in Figure 6.5 shows an example of replacing the last fully connected layer of the original VGG model by a new fully connected layer with 100 outputs. By updating the last layer only, we can use the new model for learning to classify 100 classes. Therefore, as a pre-trained CNN model can be used in these four ways as shown in Figure 6.5, the pre-trained model abstraction

should help users define the entire model easily as well as to allow users to obtain the output of a specific layer of the model easily.

To achieve this in Keras and Pytorch, the pre-trained model abstraction is considered for all pre-trained CNN models to have two sub-models, a feature extractor and a classifier. Algorithm 9 illustrates the initialisation process of a pre-trained CNN model using Pytorch, which initialises both the feature extractor (line 1) and classifier (line 2) as two Pytorch models containing a list of layers in order. Next, it stacks the classifier on top of the feature extractor to form the entire model (line 3). At the end of this initialisation process, it returns the entire model, feature extractor, and classifier to the users (line 4). Algorithm 10 illustrates the initialisation process of a pre-trained CNN model for Keras, which also initialises the feature extractor at the beginning (line 1). Different than Pytorch, it allows users to control the initialisation of the classifier by inputting a Boolean endpoint marker (line 2). If the endpoint marker is "True" (line 2-4), then Keras initialises the classifier and stacks it on top of the feature extractor. Otherwise, it uses the feature extractor as the returned model (line 6).

When obtaining the output of a layer of the feature extractor (*e.g.,* case "B" and "C" in Figure 6.5), even though the classifier is unused, Pytorch's model abstraction requires the initialisation of the entire model including the feature extractor and the classifier. To obtain the output of the hidden layers inside the classifier (*e.g.,* case "D" in Figure 6.5), Pytorch also initialises the entire model before users to obtain a specific layer output from the classifier. Therefore, a challenge with this abstraction is that the unused layers will always be initialised leading to unnecessary computer memory consumption and model size increases. For example, if we use the output of "conv2_2" shown in Figure 6.5, Pytorch will initialise all unused layers on its right-hand side. To alleviate this problem while obtaining the output on a layer in the feature extractor, Keras only requires the initialisation of the feature extractor potions without initialising the classifier. Then, a layer name (*e.g.,* "conv1_2" or "pool5") can be used to obtain the output of a specific layer. To obtain the output of the hidden layers inside the classifier, Keras initialises the entire model, and a layer name (*e.g.,* "fc1" or "fc2") is used to obtain the output. However, Keras' abstraction also initialises unused layers, with the only difference between Pytorch and Keras is that the latter would not initialise the classifier when obtaining the output of a layer of the feature extractor (*e.g.,* case "B" and "C" in Figure 6.5).

Comparing with Keras and Pytorch, TensorLayer does not abstract the pre-trained CNN models as a feature extractor and a classifier. Instead, we provide fine-grained control to the layer initialisation so that no unused layers are initialised, which reduces unnecessary computer memory consumption.

Also, TensorLayer uses a layer name to determinate which layer is the output layer, enabling users to work with the four representative pre-trained model use cases by merely changing the layer name.

6.2.5 Deep learning life-cycle management

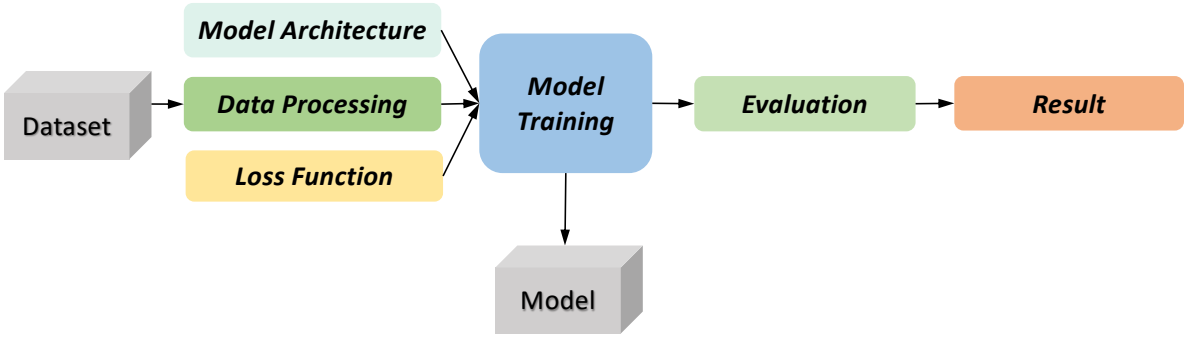

Figure 6.6: A deep learning training pipeline.

Unlike conventional software development with its well-defined development pipeline, deep learning research tends to revolve around experimentation with researchers continually trying different datasets, model architectures, and training methods [2], to improve some evaluation metric, such as MSE and accuracy. A deep learning project consists of one or multiple experiments (*i.e.,* tasks) where each experiment obtains a trained model with a given dataset [21], as in Figure 6.6. To train the model, the training pipeline includes five components of 1) defining the model(s), 2) defining the loss function(s), 3) the data processing and data augmentation for each iteration, 4) training processes, such as how and when to feed data into the model(s), the number of training epochs, the learning rate decay strategy, and 5) evaluation of the trained model(s) with selected metrics, such as MSE [21], and saving the result for further analysis. The deep learning workflow is a pipeline to develop the model that incorporates creating or acquiring a dataset, training the model based on the training pipeline, evaluating the performance, and saving the model for further use [21].

The training pipeline generates a set of data that includes the model parameters, results, and training logs. In practice, researchers perform additional work beyond the core training processes to deal with such data, such as storing the model and dataset for versioning, sharing, further retrieval and provenance as well as archiving the model and dataset along with the training pipeline to enable the reproduction of the experiment. Moreover, to speed the experiment, researchers could train multiple models concurrently for faster hyper-parameter selection [2, 22].

Managing the model, dataset, and training pipeline is mostly left for the users responsibility. Certainly, researches would prefer to focus on the algorithm design instead of spending additional effort on life-cycle management. Therefore, the demand for an efficient deep learning development including both model training and life-cycle management is rising. However, current deep learning libraries, such as Theano, Keras, TensorFlow, and Caffe, only focus on model building and training, so providing a life-cycle management tool for the deep learning workflows is important for helping researchers manage the model, dataset, and training pipeline [21, 159–161].

To manage workflow life-cycles, tech giants Facebook, Google, and Uber built internal machine learning platforms, such as FBLearner Flow, TFX, and Michelangelo for preparing data, training, and deploying models. These platforms have limited adoption in the open source community due to strong bindings with the corresponding companys internal software stacks. These tools are also primarily designed for product development instead of research. For example, Google TFX provides a Graphical User Interface (GUI) that standardises the workflow components into a data generator for preparing data, a data validator for analysing data, a transformer for data augmentation, a trainer for training models, an evaluator for evaluating models, and a pusher for pushing trained models to production model servers. Google engineers must predefine each component following the template in the GUI before execution. For production, at the end of the training, if the evaluator finds that the performance of the model reaches a threshold (*e.g.,* the accuracy is high enough), then the pusher will automatically deploy the trained model onto the server to immediately being providing the service to customers.

In open source, during their early development stage, Caffe built a Github repository [4] to share trained models, and other developers collaboratively added download links to the markdown page enable anyone to download the models. Following this approach, users were offered tools, such as ModelDB [159], Longview [160], and Bismarck [161] for storing and querying models via a database. For example, ModelDB [159] provides a GUI for life-cycle model management that allows users to find the models from a database using keywords, such as the model name or author name. In addition, recent projects exist for managing model versions or summarising training logs using a database, such as ModelHub [21] and Sherlock [162]. For example, ModelHub [21] provides a Git-like interface [5] in which users enter git commands to query Caffe models using a model name, dataset name, loss, accuracy, and hyper-parameters.

[4] `https://github.com/BVLC/caffe/wiki/Model-Zoo`
[5] `https://git-scm.com`

To manage the life-cycle of a deep learning workflow, the management of the model, dataset and training pipeline must be considered. Commercial tools such as AzureML [6] and SeaHorse [7] that provide graphical construction of deep learning workflows. For example, similar to TFX, AzureML users work through a GUI to predefine each component of the workflow, including the dataset, neural network model, and evaluator. Then, AzureML automatically executes the training according to these settings.

However, the existing life-cycle management tools including the commercial tools, require a GUI or Git-like interface to predefine each component of the deep learning workflow before execution. Through interviews with 30 deep learning researchers from Imperial College London, Peking University, and Carnegie Mellon University (10 at each institute), the majority reported that GUI or command line life-cycle management tools are suitable for production. However, in terms of deep learning research, these interfaces can restrict algorithm development because most are created on-the-fly instead of being predefined.

For example, even though many straightforward training pipelines exist, such as image classification, object detection, and segmentation that only require the definition of a fixed model with a fixed loss function [3, 37, 42], we observed most model training involves multiple models, loss functions, and a relatively sophisticated training algorithm, such as the GANs [16, 17, 19, 24, 26]. Determining the number of models and loss functions at the onset of work is difficult for advanced deep learning researches. In some cases, multiple sub-models might be connected dynamically to form a model during training. In neural module networks [140], the training data and model architecture are not independent, and the model architecture changes according to each data sample. These processes preclude the usage of a GUI template because the model and training pipeline cannot be predefined. Also, it is challenging to refactor existing research codes to fit into GUI templates.

Our findings resulted from first asking the researchers to implement one of their deep learning algorithms with an existing life-cycle management tool (*e.g.,* AzureML or SeaHorse), and following up with the survey for the interview. To prevent bias, the TensorLayer authors did not interview the researchers and the researchers were not provided with information about the goals of the survey. Specifically, as Table 6.2 includes, we collected the answers from 30 researchers within the following results. Question 1 of the user survey shows that most deep learning researchers use Python and

[6] `https://azure.microsoft.com`
[7] `https://seahorse.deepsense.io`

Question	Answer	
1. What is your current programming language for deep learning research? (multiple answers if you use different languages)	Python	29
	Matlab	1
	Others	1
2. What is your current development process for research?	Save the code, dataset, and model on a local device.	25
	Store the code on Github, and save the dataset and model on a local device.	5
	GUI-based or common line-based life-cycle management tools.	0
3. Do you change your training pipeline frequently during the deep learning research?	Yes	29
	No	1
4. Do you find the GUI or command line life-cycle management tools convenient for changing the components of the workflow, including model architecture, data processing and augmentation, training process, and evaluation?	Very convenient	0
	Somewhat convenient	0
	Moderate	1
	Inconvenient	20
	Very inconvenient	9
5. In your research, are the training data and each component in the training pipeline, including model architecture, loss function, data processing and augmentation, and training process, independent with each other?	Yes	17
	No	13
6. Do you find the GUI or command line life-cycle management tools convenient for implementing the models with architectures or connectivity that change dynamically, such as GANs and neural module networks?	Very convenient	0
	Somewhat convenient	0
	Moderate	0
	Inconvenient	2
	Very inconvenient	28
7. How difficult would it be to refactor your existing code to fit GUI life-cycle management tools?	Very easy	0
	Easy	0
	Moderate	0
	Difficult	19
	Very difficult	11

Table 6.2: Our user survey inquiring about the use of life-cycle management tools.

Question 2 lists typical development processes followed for deep learning research. Questions 3 and 4 suggest that training pipelines typically change frequently by users during deep learning research. Moreover, most users find the GUI or command line life-cycle management tools are inconvenient for changing workflow components. Question 5 highlights that the training data and each component in the pipeline are related in many models, while Question 6 suggests many users find the GUI and command line life-cycle management tools are inconvenient for implementing models with architecture or connectivity that change dynamically. Finally, Question 7 makes it clear that most users would

find it difficult to refactor existing code to fit GUI life-cycle management tools.

Alleviating the problems presented by GUI life-cycle management tools using a template, TensorLayer provides a lightweight and efficient for researchers that allows for the easy incorporation of life-cycle management into verified training pipelines. Specifically, TensorLayer supports life-cycle management via a set of Python APIs without requiring users to study how to use a GUI or command line. The Python APIs are developed because most training pipelines are implemented in Python, and this allows for seamless connections. To manage a deep learning project of one or multiple experiments, TensorLayer provides a Python object, called the TensorHub, that contains a set of methods to store and obtain the model and dataset via the database. For example, we can obtain the model with the highest accuracy or the dataset with a specific name while TensorHub masks the implementation details so that users do not need to know the details about the database.

Unlike how GUI tools manage the training pipeline, TensorLayer does not require users to split the pipeline into several predefined components, such as the data generator, transformer, trainer, and evaluator to fit into templates. TensorLayer directly stores the Python code that contains all training information into the database. TensorLayer directly stores the Python code containing all training information in the database, and the TensorHub instance contains a set of methods to store, find, and execute tasks (*i.e.,* experiments) via the database to manage the tasks and speed hyper-parameter selection. In addition to life-cycle management, TensorLayer provides a collection of data processing tools for the training pipeline to assist in the processing of image and text data.

6.3 Efficient Model Development

TensorLayers model and pre-trained model abstractions are presented in this section as well as how TensorLayer reduces the workload for library users and developers.

6.3.1 Model abstraction

Section 6.2.3 described existing issues with Keras and Pytorch for model abstraction. To initialise a new layer, Pytorch requires library users to feed the output shape of the previous layer into the new layer manually, which increases the workload for defining models. Keras does not require library

Algorithm 11 The initialisation process of a layer in TensorLayer.

Input: the previous layer $layer$, the layer setting $setting$, the forward function $forward$

1: $input \leftarrow layer$; get the input tensor from the previous layer
2: $input\_shape \leftarrow input$; get the input shape from the input tensor
3: $params \leftarrow setting, input\_shape$; initialise the parameters or reuse the parameters if they exists
4: $outputs \leftarrow input, forward, params$; get the output tensor of the current layer
5: $outputs_{pre}, params_{pre} \leftarrow layer$; get the previous parameters and outputs from the previous layer
6: $new\_layer \leftarrow outputs, params, outputs_{pre}, params_{pre}$; put all parameters and outputs into the new layer
7: **return** new_layer; return the final layer as the model

users to perform this same process, but it does require library developers to implement a function to compute the output shape of the layer, which increases the workload for implementing new layers.

An ideal model abstraction neither requires users to input the output shape of the previous layer when initialising a new layer nor requires developers to implement the function to compute the output shape of the current layer when implementing a new layer. A deep learning model is built by stacking one layer on top of another such that all hidden layers in a model eventually become connected to the final layer [2]. Therefore, separate from Keras and Pytorch that initialise all layers before forming a model and treat the layer and model as two separate classes, TensorLayer uses the final layer of the model to represent the entire model of which are contained in the same Python class. In other words, we use the final layer as the abstraction of the entire model that contains all properties of the model, including parameters and layer outputs, if the layers cumulatively collect them.

Algorithm 11 shows the layer initialisation process of TensorLayer that achieves this cumulative collection. First, the input shape of a layer is directly obtained from the input tensor as shown in lines 1 and 2 where the input tensor is the output of the previous layer. Next, in line 3, the parameters of the new layer are initialised using the input shape and the given layer setting (*e.g.,* the number of units of the fully connected layer). Then, as in line 4, the output tensor of the new layer is obtained using the input tensor, parameters, and the forward function (*e.g.,* the convolution and matrix multiplication for the convolutional and full-connected layers, respectively). To form a model, the new layer first obtains the properties of the previous layer (line 5) and then stores these previous properties and the properties of the new layer into the Python lists for the new layer (line 6). The process in line 6 that enables our model abstraction because when building models layer-by-layer, the new layer cumulatively collects all properties of the model, including its own and those properties of the previous layers.

TensorLayer

```
01. import tensorflow as tf
02. from tensorlayer.layers import Layer
03.
04. class Dense(Layer):
05.   def __init__(self, prev_layer=None, n_units=100, act=tf.nn.relu, name='dense'):
06.       super(Dense, self).__init__(prev_layer=prev_layer, act=act, name=name)
07.       # get the output shape of the previous layer (i.e., the input shape of current layer)
08.       n_in = int(self.prev_layer._shape[-1])
09.       with tf.variable_scope(name) as vs:
10.           # create new parameters
11.           W = tf.get_variable(name='W', shape=(n_in, n_units))
12.           b = tf.get_variable(name='b', shape=(n_units))
13.           # tensor computation
14.           self.outputs = self._apply_activation(tf.matmul(self.prev_layer.outputs, W) + b)
15.       # update the parameter and output lists
16.       self._add_layers(self.outputs)
17.       self._add_params([W, b])
```

Figure 6.7: Implementing the fully connected layer using TensorLayer.

In the following, we use the fully connected layer as an example to show the layer initialisation process. As Figure 6.7 presents, a fully connected layer is represented as the subclass of the layer class (line 4). The layer class provides class methods to cumulatively collect properties from the previous layer, save model parameters in a file, and restore model parameters from a file. Specifically, given the previous layer, the number of units, activation function, and layer name in line 5, initially, we feed the previous layer, activation function, and layer name into the initialisation method provided by the layer class (line 6). This process internally creates two Python lists for storing the parameters and output tensors of the previous layer. To initialise a new parameter, we obtain the number of input values n_in from the previous layer (line 8) and use this number as the shape to initialise a new weight matrix (line 11). The biases vector only relates to the number of units of the current layer (line 12). Next, to obtain the output tensor of the current layer, as in line 14, the input tensor is multiplied by the weight matrix and summed with the biases vector. If an activation function exists, then it is applied from the layer class to the activation function of the output tensor. Finally, as lines 16 and 17 show, the added layer and parameters are two methods provided by the layer class (line 2) that receive the new output tensor and new parameters of the current layer and append these new properties to the two Python lists of the current layer.

To define a model, as Figure 6.8 demonstrates, at the beginning (line 4), the input tensor x (line 3) is fed into an input layer that does not contain any parameters. By feeding the layer into the next layer sequentially (lines 5 through 10), the final layer net, in line 10 contains the properties of all layers in the model, including all parameters and outputs. Therefore, the final layer of a model can be used to represent the model abstractly.

We use TensorFlow as the computational engine, but the model abstraction is not limited to this

framework as it also works with Theano and CNTK. This abstraction can also work for the layers with multiple inputs, and the add layer and add params methods shown in line 16 and 17 of Figure 6.7 will internally remove repeated properties in the Python lists.

TensorLayer

```
01. import tensorflow as tf
02. import tensorlayer as tl
03. x = tf.placeholder(tf.float32, shape=[None, 784], name='x')
04. net = tl.layers.InputLayer(x, name='input')
05. net = tl.layers.DropoutLayer(net, keep=0.8, name='drop1')
06. net = tl.layers.DenseLayer(net, 800, tf.nn.relu, name='relu1')
07. net = tl.layers.DropoutLayer(net, keep=0.5, name='drop2')
08. net = tl.layers.DenseLayer(net, 800, tf.nn.relu, name='relu2')
09. net = tl.layers.DropoutLayer(net, keep=0.5, name='drop3')
10. net = tl.layers.DenseLayer(net, 10, name='outputs')
11. outputs = net.outputs
```

Figure 6.8: Defining an MLP model using TensorLayer. The input of the MLP has 784 values, and the output has ten values. "DenseLayer" denotes the fully connected layer, "DropoutLayer" denotes the dropout layer.

6.3.2 Pre-trained model abstraction

Algorithm 12 The initialisation process of a pre-trained model in TensorLayer.

Input: endpoint marker *end_with*, the list of layer settings in order *settings*

1: *layer* ← *input_layer*; initialise the model by an input layer
2: **for** *setting* **in** *settings* **do**
3: *new_layer* ← *setting*; get a new layer according to the layer settings
4: *layer* ← *new_layer*(*layer*); stack the new layer on top of the previous layer as the model
5: *layer_name* ← *layer*; get the layer name of the new layer
6: **if** *layer_name* == *end_with* **then**
7: *break*; stop to initialise the new layer and return the model
8: **end if**
9: **end for**
10: **return** *layer*; return the final layer as the model

Section 6.2.4 described the existing problems of the pre-trained model abstraction for Keras and Pytorch. These frameworks consider the pre-trained CNN models as two sub-models of a feature extractor and a classifier, Pytorch initialises both feature extractor and classifier resulting in unnecessary computer memory consumption when using the output of the hidden layer of the model. To alleviate this problem, Keras allows users to control initialising the classifier manually but it still includes unnecessary computer memory consumption when using the hidden output of the feature extractor or the classifier.

TensorLayer's model abstraction is distinct from Keras and Pytorch as it uses the final layer to represent the model. A model is built by cumulatively collecting all properties from the previous layers including all parameters and output tensors. Therefore, instead of considering the pre-trained CNN models as two sub-models, we return the last layer as the model by using the layer names (*e.g.,* "conv2_1", "pool2", and "fc1" as shown in Figure 6.5) to determinate which layer is the output, which avoids initialising unused layers. Specifically, Algorithm 12 illustrates this initialisation process of the pre-trained CNN model in TensorLayer. At the start, TensorLayer initialises the model as an input layer (line 1) that does not contain any parameters. Then, this process initialises the new layers sequentially in order by using the pre-defined layer settings (lines 2 to 3) and stacks the new layer on top of the previous layer (line 4). When the name of the new layer is the same as the given layer name (*i.e.,* the endpoint marker provided by the users) (lines 6 to 7), then we return this layer as the model.

Compared with Keras and Pytorch, the model initialisation process of TensorLayer provides fine-grained control of the layer initialisation without initialising any unused layers and avoids unnecessary computer memory consumption and model size increases. Also, compared with the Boolean endpoint marker of Keras, the endpoint marker of TensorLayer is a string comprised of the name of the last layer. Therefore, instead of asking users to manually determine whether to initialise the classifier, as does Keras, TensorLayer automatically identifies which layer should be initialised by using the layer name.

6.4 Efficient Life-Cycle Management

6.4.1 Components of the deep learning workflow: model, dataset, and task

The deep learning workflow is the pipeline for train the model that includes dataset acquisition, model training, model evaluation, and saving the model for further use. A deep learning project includes one or multiple experiments, which train models using a dataset and training pipeline [2]. TensorLayer's life-cycle management tool, called TensorHub, provides the three basic components, as shown in Figure 6.9, of: 1) the dataset that provides the data samples to train the model, 2) the neural network model, including both the model architecture and parameters, and 3) the task that contains the Python script related to the training pipeline including the description of the model architecture,

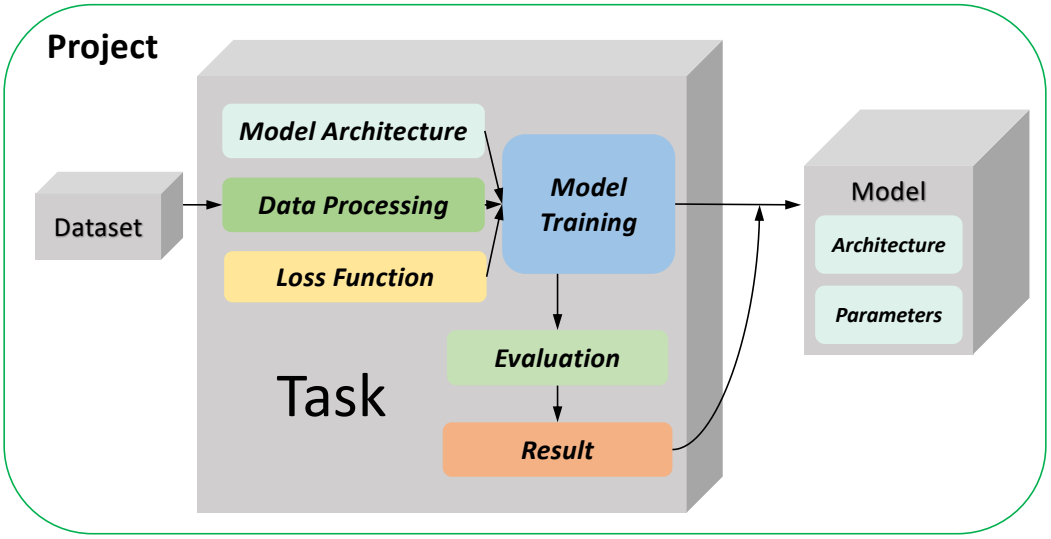

Figure 6.9: Abstracting a deep learning project using TensorLayer.

data processing, loss function, training method, and evaluation method. All information of a deep learning project can be represented by using these three components.

As described in Section 6.2.5, existing life-cycle management tools require users to predefine each component of the training pipeline to fit a template before executing the experiment. This approach restricts the algorithm development especially when different workflow components are interdependent and change dynamically. An example is the model architecture of neural module networks [140], which change dynamically according to each data sample such that the model architecture and training pipeline cannot be separated.

To address this problem, instead of splitting the training pipeline into several components as do other GUI life-cycle management tools, we directly use the Python script of the task to represent the training pipeline. Researches can then implement their training pipeline freely without restricting their implementation to fit with life-cycle management templates. For example, researches can use arbitrary Python packages in the training pipeline, such as OpenCV [8] and NLTK [9] for image and text processing, respectively. Also, researchers can dynamically connect multiple sub-models to form a model in the Python script, such as a neural module network [140], which cannot be done if the model architecture is predefined.

To manage the model, dataset, and task for versioning, sharing, further retrieval, and provenance, we

[8] `https://opencv.org`
[9] `http://www.nltk.org`

store everything into a database. Separate database tables are created for the model, dataset, and task, and each component (*e.g.,* a model or a dataset) is a record in the database table. The reason for using a database instead of a distributed file system is to address the data management challenges found in deep learning projects. These issues include 1) allowing users to mark the model, dataset, and task with user-defined keys, such as the version number, creation time, and accuracy, and 2) allowing users to easily query the model, dataset, and task by applying the user-defined keys. In the database system, the user-defined keys can be stored in the database table, allowing the model, dataset, and task to be queried by the database.

We incorporate a NoSQL database instead of an SQL database to store the model, dataset, and task for the following reasons. 1) The training process usually generates unstructured data, such as the logs and model parameters with sizes that change dynamically. SQL databases require determining the data size before storing data. However, in practice, it is hard to predefine the sizes of the data for the model and dataset. NoSQL, on the other hand, does not require pre-determining of the data size. 2) The user-defined keys can be extended or added dynamically. As deep learning research is explored around experimentation, researches will determining try new evaluation metrics, new models, and hyper-parameters, which requires the database to insert new keys into the table schemas. SQL requires the data structure to be known in advance to ensure the data conforms to the predefined table schemas. Table schemas in NoSQL, on the other hand, are relatively easy to extend. We selected MongoDB as our NoSQL database system as it offers out-of-box deployability, and it is simple to use with rich collections of third-party management and visualisation tools. However, our life-cycle management is not limited to this database as other NoSQL databases, such as Google TableStore, Amazon DynamoDB, and Azure CosmosDB are acceptable.

Specifically, TensorLayer provides a Python object, called TensorHub, as an instance that contains a set of methods to store, restore, and query the model, dataset, and task. All the implementation details are masked from the users, so users do not have to know the details about the database. Users can initialise a TensorHub instance by providing a project name, database IP, and port number, and the TensorHub instance will connect to the MongoDB using the IP and port number when it is created. Next, when storing the model, dataset, and task using the TensorHub instance, the project name is automatically added to the database records. In addition to versioning, sharing, further retrieval, and provenance, TensorHub also facilitates deep learning research experiments, such as hyper-parameter selection. By storing the task into the database, it can be accessed from different nodes (*e.g.,* different

processes from one or multiple machines). Therefore, these nodes can obtain the Python script from the task record for local execution, and save the results into the task record. The following describes how TensorLayer helps researchers manage models and datasets (Section 6.4.2) as well as how to manage and execute tasks concurrently (Section 6.4.3).

6.4.2 Managing the model and dataset

Model						
automatically fill in		mandatory field	from model object		customised field	
project name	time	model name	architecture	parameters	accuracy	...

Dataset					
automatically fill in		mandatory field		customised field	
project name	time	dataset name	dataset	description	...

Table 6.3: The database tables of the model and dataset management in TensorLayer. When using a TensorHub instance to store the model and database, the project name and create time are automatically added to the records. For storing a model, users must provide a model name and a TensorLayer model object to the TensorHub instance. The TensorHub instance first splits the model object into a model architecture description and parameters, then stores them in the record separately. Similarly, when storing a dataset, users must provide the data to the TensorHub instance. Moreover, users can add any customised files into the tables. For example, the model accuracy can be used to query model by performance, and the dataset description can provide more information about the dataset.

A model includes two components of its architecture and parameters. To describe the model architecture, inspired by the model configuration file of Caffe, TensorLayer uses a list of layer settings to describe the model architecture. As shown at the top of Figure 6.10, users can describe a model architecture in Python. However, for storing the model architecture as described in Python, it should be converted to a list of layer settings as shown at the bottom of Figure 6.10. Specifically, the metadata of a layer includes 1) the layer name, 2) layer class, such as "DenseLayer" for a fully connected layer and "DropoutLayer" for a dropout layer, 3) the layer settings, such as the number of units, activation function, and the probability for dropout, and 4) the layer name of the previous layer, which provides the connection information between layers. To extract the metadata, the layer name, layer settings, and name of the previous layer can be obtained from the input arguments of the layer (*i.e.,* the input arguments of the Python class). The layer class is the name of the Python class of the current layer. Similar to collecting the layer parameters cumulatively, as described in Section 6.3.1, the new layer also cumulatively collects the metadata from all previous layers enabling us to obtain a list of model parameters and a list of the layer settings from the final layer of the model.

Model architecture in Python

```python
x = tf.placeholder(tf.float32, shape=[None, 784], name='x')
net = tl.layers.InputLayer(x, name='input')
net = tl.layers.DropoutLayer(net, keep=0.8, name='drop1')
net = tl.layers.DenseLayer(net, 800, tf.nn.relu, name='dense1')
net = tl.layers.DropoutLayer(net, keep=0.5, name='drop2')
net = tl.layers.DenseLayer(net, 800, tf.nn.relu, name='dense2')
net = tl.layers.DropoutLayer(net, keep=0.5, name='drop3')
net = tl.layers.DenseLayer(net, n_units=10, act=None, name='output')
```

Model architecture in a list of layer settings

```python
[('x:0',
  {'class': 'placeholder',
   'dtype': 'float32',
   'prev_layer': None,
   'shape': [None, 784]}),
 ('input',
  {'class': 'InputLayer',
   'name': 'input',
   'prev_layer': 'x:0'}),
 ('drop1',
  {'class': 'DropoutLayer',
   'is_fix': False,
   'is_train': True,
   'keep': 0.8,
   'name': 'drop1',
   'prev_layer': 'input'}),
 ('dense1',
  {'act': {'func_name': 'relu',
      'module_path': 'tensorflow.python.ops.gen_nn_ops'},
   'class': 'DenseLayer',
   'n_units': 800,
   'name': 'dense1',
   'prev_layer': 'drop1'}),
 ('drop2',
  {'class': 'DropoutLayer',
   'is_fix': False,
   'is_train': True,
   'keep': 0.5,
   'name': 'drop2',
   'prev_layer': 'dense1'}),
 ('dense2',
  {'act': {'func_name': 'relu',
      'module_path': 'tensorflow.python.ops.gen_nn_ops'},
   'class': 'DenseLayer',
   'n_units': 800,
   'name': 'dense2',
   'prev_layer': 'drop2'}),
 ('drop3',
  {'class': 'DropoutLayer',
   'is_fix': False,
   'is_train': True,
   'keep': 0.5,
   'name': 'drop3',
   'prev_layer': 'dense2'}),
 ('output',
  {'class': 'DenseLayer',
   'n_units': 10,
   'name': 'output',
   'prev_layer': 'drop3'})]
```

Figure 6.10: A neural network model architecture in Python and a list of layer settings using TensorLayer's model abstraction. Top: A TensorLayer model in Python. Bottom: A TensorLayer model in a list of layer settings.

To store a model, the top portion of Table 6.3 shows the database table schema of the model. The TensorHub instance first automatically adds the project name and create time (*i.e.*, the "automatically

fill in" fields in Table 6.3) into the model record. Since a deep learning project can have multiple models (*e.g.,* GAN has a generator and a discriminator), the TensorHub instance requires users to provide a model name (*i.e.,* the "mandatory field" in Table 6.3) when storing a model to the database. Furthermore, the TensorHub instance will obtain a list of layer settings from the final layer of the model and store it as the description of the model architecture (*ie.,* the "architecture" of "from model object" in Table 6.3).

Unlike the description of the model architecture, the model parameters are relatively large [45,46,163] (*e.g.,* the size of the MobileNet parameter is 17MB). We use a specification called GridFS from MongoDB to store and retrieve model parameters that exceed 16MB. Specifically, the model parameter is first stored into blob storage managed by the GridFS, which is designed for storing large files. Then, the TensorHub instance obtains the location pointer representing the model parameter in the blob storage. Next, the TensorHub instance stores the location pointer into the parameter field of the model table (*i.e.,* the "parameters" of "from model object" in Table 6.3). When retrieving the parameters from the database, the TensorHub instance obtains the location pointer of the parameter from the table and then obtains the parameter by indexing the blob storage. TensorHub masks these processes from users, so knowledge of the database is not necessary. To support querying with user-defined keys, users may add new keys (*i.e.,* the "customised field" in Table 6.3) into the model table. For example, users can add accuracy into the model enabling them to query the model based on its accuracy.

When querying the model, the TensorHub instance provides a method to obtain a model from the database according to the create time, model name or the user-defined metric, such as accuracy. This method returns the model that best matches the query condition. For example, users can find the model with the highest accuracy from the database without manually checking each model. Given the model architecture and model parameter, the entire model object can be restored. Also, users can check the model information using third-party MongoDB management and visualisation tools, such as Mongo Management Studio [10].

For managing the dataset, the bottom of Table 6.3 shows its table schema in the database. Similar to the model, the project name and create time are automatically added to the dataset record when created. Also, since a deep learning project can have multiple datasets, the TensorHub instance requires users to provide a dataset name when storing one into the database. To support querying with user-defined keys or provide more information about the dataset, users are allowed to add new

[10] `https://mms.litixsoft.de`

keys to the table (*i.e.,* to the customised field). For example, users can add a description of the dataset to allow other users to know more information. Similar to the model parameters, the dataset is usually large, so we leverage the same GridFS approach to store and restore the dataset. The TensorHub instance provides two methods to search the dataset from the database according to the create time, dataset name or the user-defined keys, such as the dataset version. The first method returns the dataset that best matches the query condition, such as finding the dataset with a specific dataset name. The second method returns a list of datasets that match the query condition.

6.4.3 Managing and executing tasks

Managing tasks

Task					
automatically fill in				mandatory field	
project name	time	status	result	task name	python script

optional field		customised field	
hyper-parameters	saved result keys	description	...

Table 6.4: The database table for task management in TensorLayer. By using the TensorHub instance to store a task, the project name and create time are automatically added to the task record when created.

Different from the model and dataset, a task does not contain data at large scale, but all the detailed information of the training pipeline and the result. Storing a task in the database assists researchers in dealing with versioning, further retrieval, and provenance. Moreover, other nodes (*e.g.,* different processes in the same machine or alternate machine) can access the task via the database allowing the task to be executed in different nodes. Based on the database, researchers can easily execute multiple tasks concurrently using multiple nodes, facilitating the experiments of deep learning research, such as hyper-parameter selection. The following describes how to store a task, execute a task, and store the result followed by how our task abstraction helps execute multiple tasks concurrently.

The schema of the task management is shown in Table 6.4 where the TensorHub instance provides a method to create a task record in the task table. First, as is similar to the model and dataset, when users create a task record, the TensorHub instance requires users to provide a task name, which then automatically adds the task name, project name and create time in the task record. As described in Section 6.4.1, instead of splitting the training pipeline into several predefined components, we directly

use the Python script to store all detailed information. Therefore, users need to provide the code path of the Python script from which the TensorHub instance will read it and store the code as a string into the "python script" field of the task table.

Specifically, the "hyper-parameters" field is a dictionary (*e.g.,* {'parameter1' : 100, 'parameter2': 200}), specifying the variables that are passed into the Python script. For example, the hyper-parameters can be the number of units of the fully connected layer, and the Python script uses these variables to initialise the model. Running the same Python script with alternate hyper-parameters can achieves hyper-parameter selection. Moreover, the "hyper-parameters" field is not restricted to the hyper-parameter selection for the model as it can also be used for other tasks, such as cross-validation in which the hyper-parameter is not related to the model but to how to split the dataset for training and validating. In terms of saving results, the "saved result keys" is a list of strings that specifies which variables in the Python script need to be saved to the "result" field when the execution of the script is complete. For example, we can compute the testing accuracy in the Python script and save it to the "result" field. The "hyper-parameters" and "saved result keys" are optional because the task may be used for other purposes other than hyper-parameter selection, such as process data only. Moreover, when a task is just being created, the "status" field is set as "pending" to represent the task not yet being executed.

Algorithm 13 An example of creating a deep learning training task using TensorHub.

Input: the hyper-parameter *hyper_parameter*, the saved result keys *result_keys*, the python script *script*, the dataset *dataset*, and project name *project_name*

1: $TensorHub.init \leftarrow project\_name$; initialise a TensorHub instance.
2: $TensorHub.dataset \leftarrow dataset$; create a dataset record in the dataset table.
3: $TensorHub.task \leftarrow script, hyper\_parameter, result\_keys$; create a task record in the task table.

In terms of usage, tasks can be created in the task table by following Algorithm 13. We first initialise a TensorHub instance using the project name (line 1) and then put the dataset that the Python script requires into the dataset table (line 2). Finally, the process creates a task record using the Python script, hyper-parameters, and saved result keys (line 3). In the following, I describe how to execute the task in the task table.

Executing tasks

TensorHub instance provides a method to find and execute a pending task in the database as Algorithm 14 shows. Specifically, given a query condition for searching a task, such as a task name or the

Algorithm 14 The process for TensorHub to execute a task.

Input: the query condition *query*

1: $task \leftarrow TensorHub.task, query$; find a pending task from the database
2: **if** *task* exists **then**
3: $task.status \leftarrow running$; update the task status as "running"
4: $script, hyper\_parameter, result\_keys \leftarrow task$; get the Python script, hyper-parameter, and saved result keys from the task.
5: execute the task script with the given hyper-parameter.
6: $task.results \leftarrow results$; insert the results to the task with the given saved result keys.
7: $task.status \leftarrow finished$; update the task status as "finished".
8: **return** True
9: **else**
10: **return** False
11: **end if**

Algorithm 15 An example of a task runner to monitor the database and execute the pending task using TensorHub

Input: the project name *project_name*, the query condition *query*

1: $TensorHub \leftarrow project\_name$; connect to the database via TensorHub
2: **while** True **do**
3: $TensorHub.run\_top\_task(query)$; try to execute a pending task if exists
4: **end while**

ascending order by the create time, TensorHub first finds a task that matches the query condition and has a status of "pending" (line 1). If no pending task exists or no pending task matches the query condition, then it returns "False" (line 10). Otherwise, if a pending task exists and matches the query condition, TensorHub updates the task status to "running" (line 3) to specify that this task is in progress. Then, TensorHub obtains the Python script, the hyper-parameters, and the saved result keys from the task record (line 4) and executes the Python script using the hyper-parameters (line 5). When the execution concludes, TensorHub stores the result variables specified by the saved result keys in the task record (line 6), updates the task status to "finished" (line 7) specifying the task is executed, and finally returns "True" (line 8).

All processes in Algorithm 14 are masked from users, so through the TensorHub instance, users can implement a task runner to monitor the database and execute a pending task if one exists. Specifically, Algorithm 15 shows an example of a task runner in which users first initialise a TensorHub instance with the project name (line 1) and then continuously monitors the database (line 2), and, if a pending task exists, then the TensorHub instance will execute it using Algorithm 14 (line 3).

Executing tasks concurrently

Algorithm 16 An example of creating multiple deep learning tasks using TensorHub.

Input: a list of hyper-parameter *hyper_parameter_list*, the saved result keys *result_keys*, the Python script *script*, the dataset *dataset*, the project name *project_name*, and the query condition of model *query*

1: $TensorHub.init \leftarrow project\_name$; initialise a TensorHub instance
2: $TensorHub.dataset \leftarrow dataset$; insert the dataset into the database
3: **for** *hyper_parameter* **in** *hyper_parameter_list* **do**
4: $TensorHub.task \leftarrow script, hyper\_parameter, result\_keys$; create a task in the database
5: **end for**
6: **while** $TensorHub.check\_unfinished\_task()$ **do**
7: sleep 1 second;
8: **end while**
9: $model \leftarrow TensorHub.model(query)$; when all tasks are finished, get the model with the best performance
10: **return** *model*; return the best model

A deep learning project often contains many tasks. For example, hyper-parameter selection needs to train multiple models with different settings [2]. To speed the training process, researchers usually manually run multiple training scripts across many nodes simultaneously, and save the results and models from each node separately. When all trainings are finished, to analyse the results, users must collect the models and results from all nodes manually to analyse the results.

To automate this process, by extending Algorithm 13, users can create multiple tasks with different hyper-parameters in the database as shown in lines 3 through 5 of Algorithm 16. Then, by starting N task runners (Algorithm 15) on N nodes, these N tasks can be executed simultaneously (where N is less than or equal to the total number of tasks). Moreover, the status of the task record prevents a task runner from executing complete or in-progress tasks. The status enables fault-tolerant execution because a task runner can only run a single task that has a status of "pending".

As hyper-parameter selection looks to find the model with the best performance (*e.g.,* accuracy), instead of requiring users to check the results of each model manually, our life-cycle management automatically returns the model with the best result. Specifically, the TensorHub instance provides a method to check if all tasks are finished (line 6 of Algorithm 16), and the method will return "True" if unfinished tasks exist (*i.e.,* the tasks with a status of "pending" or "running"). When all task statuses are "finished," the method returns "False" and breaks the while loop in line 6. Then, users can find the model with the best performance from the database using the descending order of the performance (*e.g.,* accuracy) (line 9).

Hyper-parameter selection is an example that benefits from running multiple tasks concurrently.

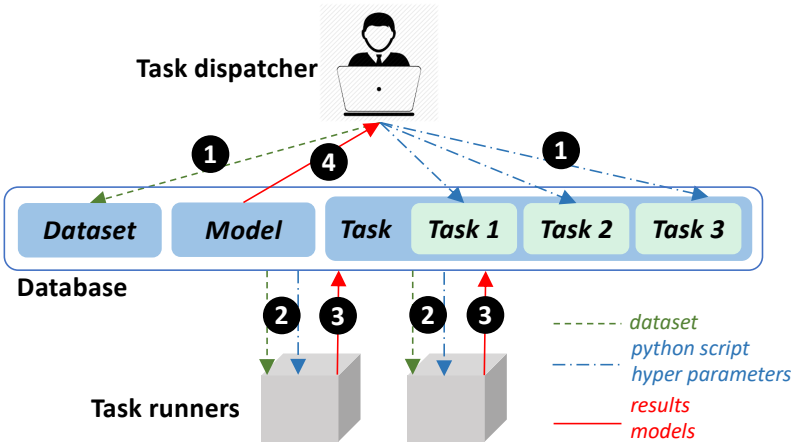

Figure 6.11: Execute multiple tasks concurrently for training multiple models using TensorLayer.

Specifically, Figure 6.11 shows a concurrent process for training multiple models with different hyper-parameters. In the first step, as indicated by "1," Algorithm 16 is run locally as a task dispatcher to insert a dataset and multiple tasks into the database. The status key of all tasks are set to "pending" when they are created, and the task dispatcher starts to wait for all task statuses to be "finished." In the second step, as indicated by "2", when a task runner finds a task record with a status of "pending," it updates the status to be "running" and executes the corresponding Python script with its hyper-parameters. In the third step, "3," when training is finished, the task runner updates the status of the task record to be "finished," and saves the variables corresponding to the "saved result keys" to the "result" field of the task record as well as saves the trained model into the database. Finally, in "4," when all tasks are finished, following lines 6 to 9, the task dispatcher obtains the model with the best performance (*e.g.*, accuracy) from the database.

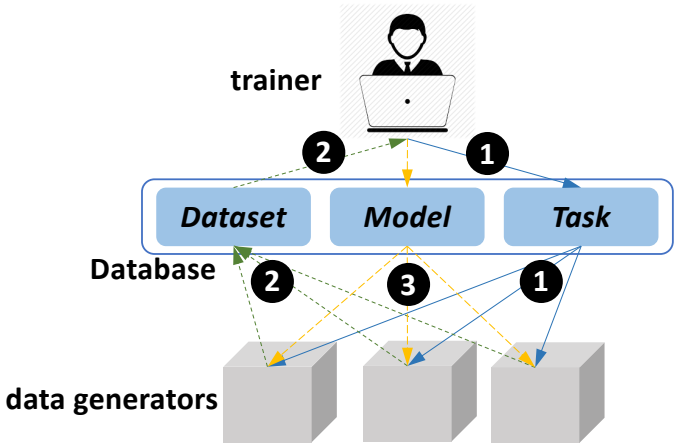

Figure 6.12: Execute multiple tasks concurrently for training one model with multiple data generators using TensorLayer.

Another example that benefits from executing multiple tasks concurrently is deep reinforcement learning (DRL) [12,164]. Distinct from other deep learning tasks where the dataset is given and is constant, such as in image classification [10] and visual question answering [165], the training data of DRL is generated continuously from a simulated environment. In other words, the dataset is not predefined. The challenge of training a DRL model is the speed at which to generate training data from the simulated environment. The acceleration of training usually requires multiple nodes for the simulation [12]. To facilitate training, our life-cycle management tool allows for the building of a DRL training system in which multiple nodes concurrently generate data samples and pass them into the training program via the database. Specifically, Figure 6.12 shows the process of executing multiple tasks concurrently while training one DRL model with multiple data generators. In the first step, indicated by "1," users have a local model trainer (*i.e.,* a task dispatcher) for coordinating the tasks to the task runners for starting multiple data generators. In this case, as all task runners generate the data samples in the same way, the "hyper-parameter" and "saved result keys" do not need to be provided. In "2," the data generators create the training data samples and store them into the dataset table. The trainer continuously queries the training data samples from the database and updates the model locally. In "3," after updating the model locally, the trainer updates the model in the database, and the data generators synchronise their model parameters with the database.

6.5 Evaluation

We compares TensorLayer, Keras, and Pytorch to evaluate the model and pre-trained model abstractions. To evaluate the life-cycle management, we demonstrate the efficacy of TensorLayer using two case studies followed by a comparison between TensorLayer and existing life-cycle management tools. The relevant source codes can be found in the Appendix.

6.5.1 Model abstraction

A concern towards TensorLayer is speed degradation, which we investigate by running three classic deep learning tasks [11] with the implementations of TensorLayer and a pure TensorFlow on a Titan X Pascal GPU. We run each task 20 times and summarise the running speed for TensorLayer and TensorFlow in Table 6.5. As the models are compiled to run and the data iterations use the same

[11] `https://www.tensorflow.org/tutorials`

		TensorLayer	TensorFlow
CIFAR 10	averaged images/sec	2530.5	2529.8
	standard derivation	5.18	5.05
	min/max images/sec	2520/2541	2519/2541
PTB LSTM	averaged words/sec	18066.3	18070.0
	standard derivation	16.50	17.41
	min/max words/sec	18038/18095	18032/18098
Word2Vec	averaged words/sec	58173.8	58171.4
	standard derivation	26.37	22.28
	min/max words/sec	58130/58209	58139/58210

Table 6.5: Comparison of TensorLayer and TensorFlow on classic benchmarks. CIFAR-10: image classification, Penn TreeBank (PTB): language modelling, and Word2Vec: word embedding.

methods, the results show that TensorLayer and pure TensorFlow implementations offer nearly the same running speed suggesting that the model abstraction would not sacrifice running speed.

Question	Ranking Score	
The simplicity of implementing new layers	TensorLayer	1.1
	Keras	3
	Pytorch	1.7
The simplicity of defining a model using existing layers	TensorLayer	1
	Keras	1
	Pytorch	2.93

Table 6.6: The ranking scores of the model abstractions for TensorLayer, Keras, and Pytorch. A lower score indicates improved simplicity.

For evaluating the model abstraction, we recruited 30 researchers from Imperial College London, Peking University, and Carnegie Mellon University. The participants were asked to finish two deep learning development tasks and complete a user survey. The first task of evaluating the simplicity of implementing new layers requires they implement a fully connected layer and convolutional layer using TensorLayer, Keras, and Pytorch, respectively. The second task is to evaluate the simplicity of defining models using the existing layers as they are required to define a VGG16 model using the existing layer APIs of TensorLayer, Keras, and Pytorch, respectively.

Through the survey, users rank the ease of usage of TensorLayer, Keras, and Pytorch with a score varying from 1 to 3, where 1 indicates the easiest and 3 indicates the hardest. If two methods have similar simplicity, then the users can rank them to the same ranking score. We average the ranking scores as our final scores for each method where a lower score means better simplicity. Table 6.6 shows the survey results, and the ranking scores show that TensorLayer has the best simplicity of implementing new layers. It also presents equal simplicity with Keras for defining the model using the

existing layer APIs.

6.5.2 Pre-trained model abstraction

TensorLayer

```
01. model = tl.models.VGG16(x)
02. outputs = model.outputs
```

Keras

```
01. model = keras.applications.vgg16.VGG16(input_tensor=x)
02. outputs = model.output
```

Pytorch

```
01. model = torchvision.models.vgg16().eval()
02. outputs = model(x)
```

Figure 6.13: Defining the entire VGG16 using TensorLayer, Keras, and Pytorch's model abstractions.

TensorLayer

```
01. model = tl.models.VGG16(x, end_with='pool5')
02. pool5 = model.outputs
```

Keras

```
01. model = keras.applications.vgg16.VGG16(include_top=False, input_tensor=x)
02. pool5 = model.output
```

Pytorch

```
01. class myVGG(torch.nn.Module):
02.     def __init__(self):
03.         super(myVGG, self).__init__()
04.         VGG = torchvision.models.vgg16().eval()
05.         self.features = VGG.features
06.
07.     def forward(self, x):
08.         x = self.features(x)
09.         return x
10.
11. model = myVGG()
12. pool5 = model(x)
```

Figure 6.14: Obtaining the output of the feature extractor (*i.e.,* "pool5") of VGG16 using TensorLayer, Keras, and Pytorch's pre-trained model abstractions where "pool5" is the output of the fifth (last) convolutional block. For Keras, to reduce the unnecessary parameter initialisation, users disable the initialisation of the classifier (*i.e.,* the fully connected layers) by setting "include_top" to "False." For Pytorch, after initialising the entire VGG, the feature extractor denoted by "features" can be used in the feedforward propagation.

The following compares the pre-trained model abstraction through TensorLayer, Keras, and Pytorch for which we use to define VGG16 for four representative use cases, including 1) using the output

TensorLayer

```
01. model = tl.models.VGG16(x, end_with='conv4_3')
02. conv4_3 = model.outputs
```

Keras

```
01. model = keras.applications.vgg16.VGG16(include_top=False, input_tensor=x)
02. conv4_3 = model.get_layer('block4_conv3').output
```

Pytorch

```
01. class myVGG(torch.nn.Module):
02.     def __init__(self):
03.         super(myVGG, self).__init__()
04.         VGG = torchvision.models.vgg16().features.eval()
05.         self.all_layers = []
06.         for i, layer in enumerate(VGG.children()):
07.             self.all_layers.append(layer)
08.             if i == 22:
09.                 break
10.
11.     def forward(self, x):
12.         for layer in self.all_layers:
13.             x = layer(x)
14.         return x
15.
16. model = myVGG()
17. conv4_3 = model(x)
```

Figure 6.15: Obtaining the output of the fourth convolutional block (*i.e.,* "conv4_3") of VGG16 using TensorLayer, Keras, and Pytorch's pre-trained model abstractions. For Keras, to reduce unnecessary layer initialisation, users can disable the initialisation of the classifier (*i.e.,* the fully connected layers) by setting "include_top" to "False." For Pytorch, after initialising the entire VGG, all layers before"conv4_3" can be used from the feature extractor in the feedforward propagation.

of the entire model, 2) using the output of the feature extractor, 3) using the hidden output of the feature extractor, and 4) using the hidden output of a classifier. These four cases can cover all possible use cases of the pre-trained CNN models as described in Section 6.2.4.

First, as Figure 6.13 illustrates, users apply TensorLayer, Keras, and Pytorch to define the entire VGG16 model. They all initialise the entire VGG16 (line 1) and then get the 1,000 probabilities denoted by "outputs" for the ImageNet classification (line 2). In this case, as all libraries initialise all layers of VGG16, they initialise the same number of parameters.

Second, to obtain the output of the feature extractor of VGG16 (as shown in the "B" model of Figure 6.5), TensorLayer and Keras only initialise the feature extractor portion while Pytorch initialises the entire model including the classifier. Specifically, as Figure 6.14 shows, the last layer of the feature extractor is the max pooling layer of the fifth convolutional block, and TensorLayer uses "pool5" as the endpoint marker (line 1 of TensorLayer) to obtain the output of the feature extractor (line 2 of TensorLayer). No unused layers will be initialised. Similarly, Keras provides an "include_top"

TensorLayer

```
01. model = tl.models.VGG16(x, end_with='fc2_relu')
02. fc2 = model.outputs
```

Keras

```
01. model = keras.applications.vgg16.VGG16(input_tensor=x)
02. fc2 = model.get_layer('fc2').output
```

Pytorch

```
01. class myVGG(torch.nn.Module):
02.     def __init__(self):
03.         super(myVGG, self).__init__()
04.         VGG = torchvision.models.vgg16().eval()
05.         self.features = VGG.features
06.         self.fcn_layers = []
07.         for i, layer in enumerate(VGG.classifier.children()):
08.             self.fcn_layers.append(layer)
09.             if i == 4:
10.                 break
11.
12.     def forward(self, x):
13.         x = self.features(x)
14.         x = x.view(x.size(0), -1)
15.         for layer in self.fcn_layers:
16.             x = layer(x)
17.         return x
18.
19. model = myVGG()
20. fc2 = model(x)
```

Figure 6.16: Obtaining the output of the second last fully connected layer (*i.e.*, "fc2") of VGG16 using TensorLayer, Keras, and Pytorch's pre-trained model abstractions. For Keras and Pytorch, users need to initialise the entire classifier.

argument to initialise a VGG model without initialising its classifier (line 1 of Keras), and will not initialise unused layers. Pytorch's CNN models have two attributes of"features" and "classifier" for the feature extractor and classifier, respectively. After initialising the entire VGG model (line 4 of Pytorch), Pytorch uses the feature extractor (line 5 of Pytorch) in the feedforward propagation directly (line 8 of Pytorch). In this case, Pytorch initialises the unused classifier, while TensorLayer and Keras would not.

Third, to obtain the hidden output of the feature extractor, the "C" model of Figure 6.5 shows an example of using the output of the fourth convolutional block. In this case, TensorLayer uses "conv4_3" as the endpoint marker (line 1 of TensorLayer) to obtain the output of the fourth convolutional block (line 2 of TensorLayer), and no unused layers are initialised. In contrast, Keras still initialises the entire feature extractor (line 1 of Keras), and Pytorch initialises the entire model to obtain the feature

extractor (line 4 of Pytorch). Similar with TensorLayer, to obtain a specific layer output, Keras provides a "get layer function" (line 2 of Keras) that indexes the layer output from the model using a specific layer name. In this case, the unused fifth convolutional block is initialised by Keras. For Pytorch, given the feature extractor (line 4 of Pytorch), users need to manually collect all layers before "conv4_3" by using a layer index (lines 5 through 9 of Pytorch), and then these layers are used in the feedforward propagation (lines 12 through 13). In this case, all layers after "conv4_3" are unused while still being initialised by Pytorch.

Fourth, to obtain the output of the hidden layers of the classifier, the "D" model of Figure 6.5 shows an example of using the output of the second last fully connected layer. In this case, as seen in the code of Figure 6.16, both Keras and Pytorch initialise the entire model (line 1 of Keras and line 4 of Pytorch). Next, to obtain the output of the second fully connected layer, Keras uses the "get layer function" (line 2 of Keras) to index the layer output from the model with a specific layer name. For Pytorch, users manually collect the first two fully connected layers and the ReLU layers from the classifier by using a layer index (lines 7 through 10 of Pytorch). Then, these layers and the feature extractor (line 5 of Pytorch) are used in the feedforward propagation (lines 13 through 16). In this case, both Keras and Pytorch initialise the unused last fully connected layer. In contrast, to obtain the output of second last fully connected layer of VGG16, TensorLayer uses "fc2" as the endpoint marker (line 1 of TensorLayer) and does not initialise the last fully connected layer, unlike Keras and Pytorch.

Based on these four representative use cases, TensorLayer's pre-trained model abstraction avoids initialising unused layers when applying the pre-trained CNN models differently. Therefore, compared with Keras and Pytorch, it avoids unnecessary use of computer memory. In terms of its API interface, compared with Pytorch, TensorLayer does not require users to define the feedforward propagation manually. Compared with Keras, TensorLayer does not require users to manually control initialising the classifier.

In the following, we compare the computer memory usage of TensorLayer, Keras, and Pytorch. VGG16 and MobileNet are the two most commonly used pre-trained CNN models for academia [28, 34, 166] and industry [46]. TensorLayer's pre-trained model abstraction is compared with Keras and Pytorch in Table 6.7 with lists of the number of initialised parameters for obtaining different layer outputs of VGG16 and MobileNet. As Pytorch always initialises both feature extractor and classifier, it includes more unused parameters compared with TensorLayer and Keras when obtaining the hidden output

VGG16			
	TensorLayer	Keras	Pytorch
outputs	138357544	138357544	138357544
fc2	134260544	138357544	138357544
fc1	117479232	138357544	138357544
pool5	14714688	14714688	138357544
pool4	7635264	14714688	138357544
pool3	1735488	14714688	138357544
pool2	260160	14714688	138357544
pool1	38720	14714688	138357544
MobileNet			
	TensorLayer	Keras	Pytorch
outputs	4253864	4253864	4253864
depth13	3228864	3228864	4253864
depth12	2162880	3228864	4253864
depth11	1627840	3228864	4253864
depth10	1356992	3228864	4253864
depth9	1086144	3228864	4253864
depth8	815296	3228864	4253864
depth7	544448	3228864	4253864
depth6	273600	3228864	4253864
depth5	137152	3228864	4253864
depth4	67264	3228864	4253864
depth3	31808	3228864	4253864
depth2	13248	3228864	4253864
depth1	3712	3228864	4253864
conv	992	3228864	4253864

Table 6.7: The number of initialised parameters for obtaining different layer outputs of VGG16 and MobileNet using TensorLayer, Keras, and Pytorch's pre-trained model abstractions. The VGG16 feature extractor has five convolutional blocks with outputs denoted by "pool1" through "pool5", the classifier has three fully connected layers with outputs denoted by "fc1," "fc2," and "outputs." MobileNet's feature extractor consists of 14 convolutional blocks with outputs denoted by "conv" and "depth1" through "depth13", and the classifier has only one layer denoted by "outputs."

of the models. For Keras, while users can manually disable the initialisation of the classifier when obtaining the output of the feature extractor, it still includes unused parameters when obtaining the hidden output of the feature extractor or classifier.

Quantitatively, Table 6.7 demonstrates that when getting the output of the feature extractor from VGG16, Pytorch initialises 9.4 times (*i.e.,* 138357544/14714688) the number of parameters than TensorLayer and Keras. When obtaining the hidden output of the feature extractor, both Pytorch and Keras initialise more parameters than TensorLayer. For example, for the output of the fourth convolutional block (*i.e.,* "pool4"), Keras initialises 1.9 times (*i.e.,* 14714688/7635264) the number of parameters than TensorLayer. As the classifier of MobileNet include only one layer, Pytorch initialises 1.3

times (*i.e.,* 4253864/3228864) the number of parameters than TensorLayer and Keras when obtaining the output of the feature extractor, which is less than for VGG16. However, with the state-of-the-art CNN models, the feature extractors usually contain more layers than the classifier [3, 45–48] so Keras will initialise many more parameters than TensorLayer when obtaining the hidden output of the feature extractor. In summary, as TensorLayer does not initialise unused parameters and layers, it result in fewer or equal initialised parameters compared with Keras and Pytorch for different use cases, which directly reduces unnecessary computer memory consumption.

Question	Ranking Score	
The simplicity of using the entire pre-trained models	TensorLayer	1
	Keras	1
	Pytorch	1.03
The simplicity of using the output of the feature extractor of pre-trained models	TensorLayer	1.03
	Keras	1.03
	Pytorch	3
The simplicity of using the hidden output of the feature extractor of pre-trained models	TensorLayer	1
	Keras	1.97
	Pytorch	3
The simplicity of using the hidden output of the classifier of pre-trained models	TensorLayer	1
	Keras	1.93
	Pytorch	3

Table 6.8: The ranking scores of the pre-trained model abstractions of TensorLayer, Keras, and Pytorch.

To evaluate the pre-trained model abstraction, we follow the same survey approach as introduced in Section 6.5.1 except that the participants were required to use TensorLayer, Keras, and Pytorch's pre-trained model abstraction APIs to define the VGG16 for four use cases described above, including: a) the entire model, b) the feature extractor, c) the hidden output of the feature extractor, and d) the hidden output of the classifier. They were then required to rank the simplicity of using TensorLayer, Keras, and Pytorch for all use cases. A lower score indicates better simplicity.

The ranking scores presented in Table 6.8 show that TensorLayer, Keras, and Pytorch offer the same simplicity when defining the entire pre-trained CNN models. However, TensorLayer and Keras provide better simplicity than Pytorch when using the output of the feature extractor. TensorLayer has a significantly better simplicity than Keras and Pytorch when using the hidden output of either the feature extractor or classifier of the pre-trained CNN models.

6.5.3 Life-cycle management

The life-cycle management approach of TensorLayer allows researchers to store the model, dataset, and tasks for versioning, sharing, further retrieval, and provenance while also helping to facilitate the deep learning research experiments through the execution of multiple tasks concurrently. In the following, the life-cycle management is analysed in two cases that offer great examples of jointly leveraging the model abstraction and life-cycle management tools. The first trains multiple models concurrently for increasing the speed of hyper-parameter selection. The second trains a DRL model using multiple data generators for speeding the training.

Case study 1: Hyper-parameter selection

We apply TensorLayer to implement a classic digits classification task for evaluating the lifecycle management tool for hyper-parameter selection. MNIST [12] is a dataset for handwritten digits classification with 60,000 training images and 10,000 testing images, each having a size of 28×28 pixels, *i.e.,* 784 floating-point pixel values. We split the training images into 50,000 for training and 10,000 for validating. An MLP model with two hidden fully connected layers each having 800 output units with ReLU activation. The output layer of MLP is fully connected with ten output units representing the digits between 0 to 9. To avoid overfitting, we insert a dropout layer between the two hidden layers and another dropout layer between the last hidden layer and the output layer. The hyper-parameters we search are the dropout keeping probabilities set for the two dropout layers.

By varying the dropout keeping probabilities from 10% to 100% at an interval of 10% (*i.e.,* applying grid search), we generate 100 tasks (*i.e.,* experiments that vary the hyper-parameter settings) for hyper-parameter selection. For each task, to have a fair comparison, we use the same training settings of a batch size of 500, training epochs of 100, and an Adam optimiser with a learning rate of 0.0001. The hyper-parameter selection follows the process described in Section 6.4.3. Specifically, the task dispatcher follows the process of Algorithm 16, and the task runners follow the process of Algorithm 15. The experiments are run on a cluster with 8 Titan XP GPUs to test the system where each GPU starts a task runner.

The red line in Figure 6.17 shows the relationship between the number of task runners and the total run time to finish 100 tasks. As the number of task runner increases, the total running time is reduced

[12] `http://yann.lecun.com/exdb/mnist/`

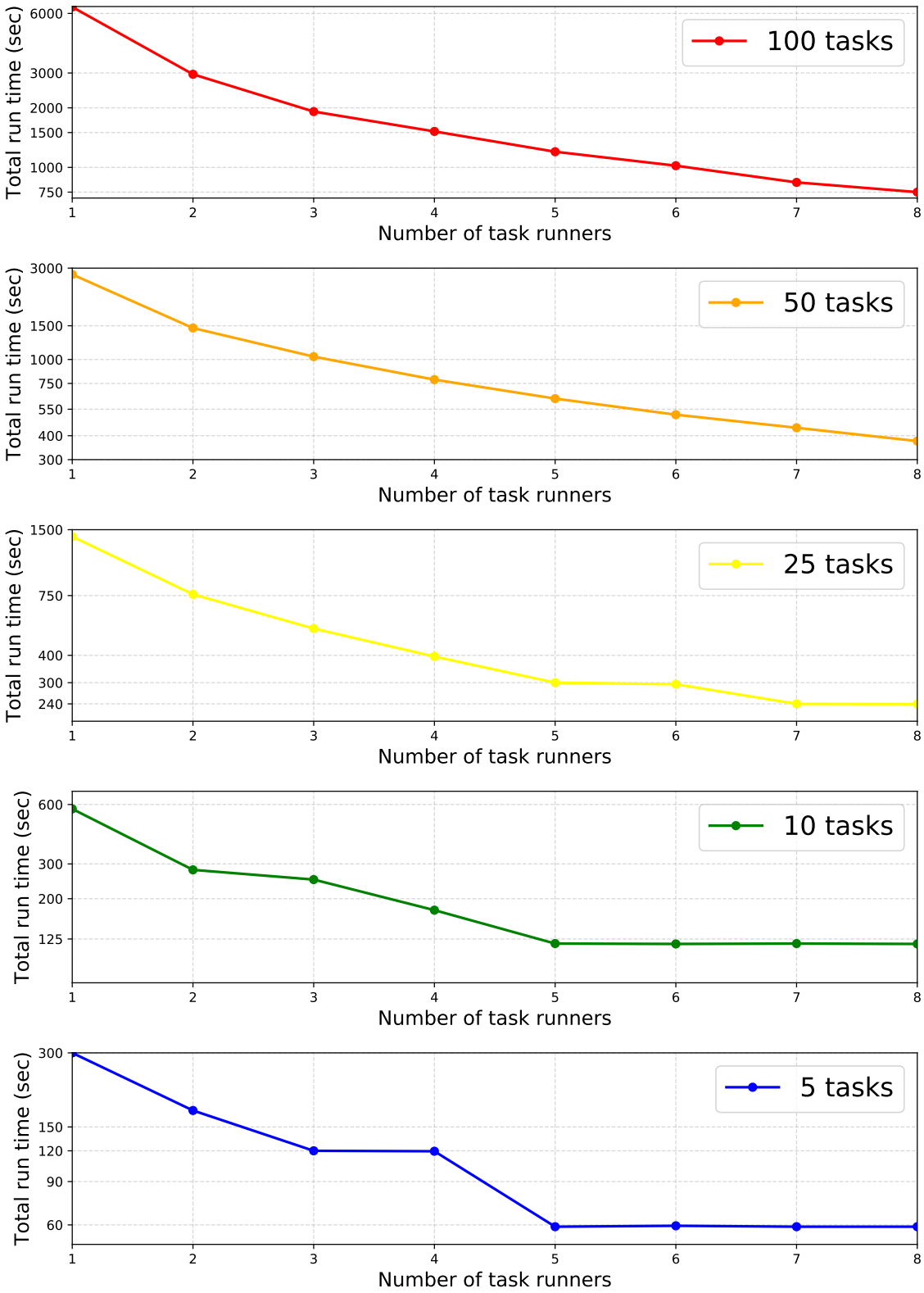

Figure 6.17: Total run time vs the number of task runners used for a range in the number of tasks of the hyper-parameter selection.

proportionally. For instance, as the red line indicates, when the number of task runners doubles from 1 to 2, 2 to 4 or 4 to 8, the total run time is halved. The reason is due to different tasks of the

hyper-parameter selection are fully independent because the task runners only need to obtain the dataset from the database before the training and save the model to the database after the training.

Moreover, by changing the interval of the dropout keeping probabilities, we evaluate the system using 50, 25, 10, and 5 tasks (*i.e.,* experiments). When the number of tasks M is less than the number of task runners N, then $N - M$ task runners become redundant. For instance, as the blue line in Figure 6.17 shows, when the number of task runners is larger than five, there is no further speed improvement. Also, if the number of tasks M cannot be evenly divided by the number of task runner N, then there are $N - M\%N$ task runners in idle when running the last $M\%N$ experiments. For example, the green line in Figure 6.17 shows the result for ten tasks, and when the number of task runners is two, each task runner runs five tasks. When the number of task runners is three, all task runners first finish tree tasks, then only a single task runner processes the remaining one task setting, the other two task runners are idle resulting in no proportionally speed improvement. Another example is in the blue line of Figure 6.17 showing that given three or four task runners, they will first finish three or four tasks, respectively, and then one or two task runners will idle when processing the remaining one or two tasks resulting in no speed improvement when increasing the number of task runners from three to four.

In summary, when the running time of each task is the same, the total speedup ratio is the round up to the integer of the fraction, which is the number of tasks M divided by the number of task runners N *i.e.,* $\lceil \frac{M}{N} \rceil$. Therefore, when the number of tasks is far greater than the number of task runners, the speedup ratio becomes linear as $\lim_{M \to +\infty}(\lceil \frac{M}{N} \rceil) = \frac{M}{N}$.

Case study 2: Deep reinforcement learning

We use TensorLayer to implement a classic deep reinforcement learning (DRL) task in a cluster that has 10 Gbps connectivity for a second evaluation of our life-cycle management tool. As described in Section 6.4.3, a bottleneck of DRL training is the speed to generate the training data. Unlike the hyper-parameter selection where each task is independent, the acceleration of DRL training requires multiple task runners (*i.e.,* data generators) to generate training data concurrently, and send the data to a model trainer via the database. We use the classic task of the Atari ping-pong game [13] [164] that requires the DRL model to learn to play the game by directly observing the computer screen. The model input of the Atari ping-pong game includes screen images of the game each composed of an

[13] `https://gym.openai.com/envs/Pong-v0/`

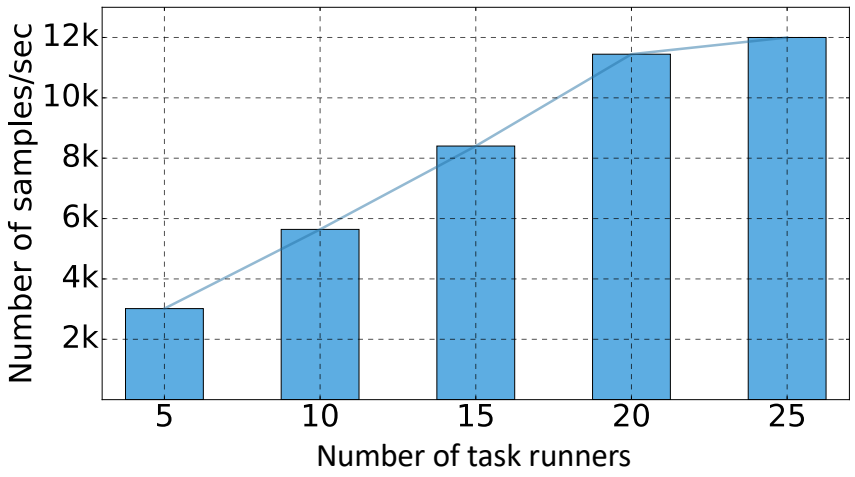

Figure 6.18: Training throughput vs the number of task runners used for generating training samples.

80 × 80 pixel matrix, *i.e.,* 6,400 floating-point values. The model output is the probabilities of three actions of move-left, stop, and move-right.

For the DRL algorithm, we follow the verified model architecture and training pipeline introduced in Karpathy *et al.* [14]. The model includes one hidden fully connected layer with 200 units and ReLU activation. The output layer is a fully connected layer with three units representing the probabilities of three actions, and the model is trained using the REINFORCE algorithm [167]. An episode of the ping-pong game is represented by a period from the start of the game to the ball hitting the border of the ping-pong table. The model is updated once based on the images from ten episodes using the RMS optimiser [62] with a learning rate of 0.0001. We start multiple Atari ping-pong games on multiple nodes as data generators that simulate the game playing and send the generated samples including the observations (*i.e.,* images), actions and rewards, to the database.

In our experiment, each task runner is a data generator running on a 2.5 GHz Intel Core i7 CPU for simulating the game environment and saving the generated data samples to the dataset at the end of each episode. Meanwhile, we start a trainer on the local machine with a single GTX 980 GPU. The trainer continuously obtains data samples from the database, updates the local model parameter, and synchronises its model parameters to the database every ten updates. The data generators synchronise the model parameter from the database every 20 episodes. Figure 6.18 illustrates the scalability of TensorLayer in powering such a system. The row axis is the number of task runners equal to the

[14] `http://karpathy.github.io/2016/05/31/rl/`

number of data generators, so the number of samples per second the system can generate increases proportionally to the number of task runners. The column axis is the training throughput representing the number of samples the model trainer can use every second. We increase the number of task runners starting at five with an interval of five. The results show that when the number of task runners is less than 20, the training throughput increases proportionally with the number of task runners suggesting that the speed of generating training data is slower compared to the speed of model training. However, when the number of task runners is increased from 20 to 25, the training throughput does not increase proportionally suggesting that the speed of generating training data is no longer a bottleneck for DRL training as the model trainer reaches maximum capacity.

Comparing to existing life-cycle management tools

	TensorLayer	**ModelDB**	**ModelHub**	**AzureML**	**SeaHorse**
Supports model management	Yes	Yes	Yes	Yes	Yes
Supports dataset management	Yes	No	Yes	Yes	Yes
Supports training pipeline management	Yes	No	Yes	Yes	Yes
Supports concurrent training for hyper parameter selection	Yes	No	Yes	Yes	No
Supports concurrent data generation for deep reinforcement learning	Yes	No	No	No	No
Supports management of neural module networks	Yes	No	No	No	No
Supports automatic model deployment	No	No	No	Yes	Yes
Easy to use	8.6	8.83	4.3	7.1	6.5
Easy to migrate	9.2	–	4.4	4.1	3.7
Easy for research	8.3	–	4.0	3.0	2.8

Table 6.9: Comparison of TensorLayer with other management tools.

To further evaluate the efficacy of our life-cycle management, we compare with four management tools. The top seven rows of Table 6.9 show the availability of functions for different tools. First, ModelDB [159] only helps users store and find models using a database via a GUI while others support the life-cycle management of deep learning workflows, including the model, dataset and training pipeline.

TensorLayer supports dispatching multiple tasks on multiple nodes for training multiple models concur-

rently, which speeds the training of hyper-parameter selection and running multiple tasks on multiple nodes for generating data samples simultaneously, which also speeds the training of deep reinforcement learning. Other tools only support the concurrent hyper-parameter selection where users must manually start multiple trainings when using ModelHub and SeaHorse because these tools do not support automatic task dispatching. In addition, TensorLayer is the only framework that supports the life-cycle management of neural module networks because the model architecture changes dynamically based on each training samples [140], and the data processing and model architecture cannot be predefined separately. To avoid this problem, TensorLayer directly stores the Python code containing all training information into the database instead of splitting each component of the training pipeline to fit a predefined template.

While TensorLayer is competent for the above functionalities, its life-cycle management does not support automatic model deployment to push trained model onto a model server when the model performance matches a predefined evaluation metric (*e.g.,* accuracy). Such automatic deployment is desirable for a production system, but as the current design goal of TensorLayer is for research purpose, this feature remains open for a future release.

In addition to reviewing the available functions, we further interviewed the same research participants from the previous survey to evaluate the life-cycle management framework by rating the tools with a score scaled from 1 to 10 representing very bad to very good for how straightforward the tools are to integrate into their model building environments. The scoring included questions on:

- Easy to use: rate the ease of use without considering the research purpose.
- Easy to migrate: rate how easy it is to migrate the verified training pipeline into the tools.
- Easy for research: rate the ease of use for their research applications.

The averaged scores from all participants are included in the bottom three rows of Table 6.9 with results suggesting that TensorLayer scores significantly higher for easy to migrate and easy for research. These responses are attributed to the pure Python API that does not require researchers to learn a GUI or command line interface as they can directly store the Python script into the task record without restricting the implementation to conform to a template for the life-cycle management. While ModelDB resulted in a higher score for easy to use compared to TensorLayer, ModelDB only supports model management without supporting model training and lifecycle management [159]. This feature

limitation is the reason why participants were not asked to rate MondelDB for how easy it is to migrate the verified training pipeline and how easy it is for research.

6.6 Conclusions and Discussion

In this chapter, we presented a research-oriented library for efficient deep learning development, which plays an important role in practical deep learning. We first proposed a new abstraction method for model development that reduces the workload of library users when defining models as well as that of library developers when implementing new layers. Moreover, we further proposed a pre-trained model abstraction to provide fine-grained control to the layer initialisation for avoiding initialising unused layers and reducing the unnecessary computer memory consumption. We evaluated our model and pre-trained model abstractions by comparing them with two representative libraries, Keras and Pytorch. In this study, we chose to use TensorFlow as our computational engine, but our model abstraction is not limited to a specific engine.

In terms of life-cycle management, we abstract the deep learning workflow into three components of the model, dataset and task. Instead of splitting each component of the training pipeline to fit a pre-defined template, we directly store the Python code that contains all training details into the database. By doing so, we enable users to store the model, dataset, and task for versioning, sharing, further retrieval, and provenance while also helping users to facilitate the deep learning research experiments via assisting the execution of multiple tasks concurrently. We evaluated the life-cycle management using two case studies of running multiple tasks for model training and data generation for increasing the speed for the training of the hyper-parameter selection and deep reinforcement learning, respectively. We chose to use MongoDB, but our life-cycle management design is not limited to a specific database.

Impact and award

TensorLayer has been released on Github [15] since September 2016. By September 2018, it has received more than 4,300 stars, 100,000 installations, and has formed an active development community. The contributors consist of PhD students from universities in the UK, such as Imperial College and Edinburgh University, and universities in US and China, such as CMU, Stanford, Tsinghua, and Zhejiang

[15] `https://github.com/tensorlayer/tensorlayer`

universities, as well as many engineers from companies such as Google, Alibaba, Baidu, and Bloomberg. The contributor list can be found in the Github contributor page [16]. This library was awarded the Best Open Source Software Award issued by ACM Multimedia (MM) 2017, and is featured in a published book that introduces TensorLayer for deep learning development [39]. In addition, TensorLayer has been used in many researches such as image super resolution [168], biomedical analysis [169], social data analysis [170], action recognition [8], multi-model research [171], image transformation [90, 166], medical signal processing [41, 172], and medical image reconstruction [173, 174].

Availability

TensorLayer is open sourced under the license of Apache 2.0 and supports Linux, Mac OS, and Windows environments. It provides English and Chinese documentation, abundant tutorials and thorough examples, such as CNN models (*e.g.,* VGG, ResNet, and Inception), text-related applications (*e.g.,* text generation, Word2vec, machine translation, and image captioning), GAN models (*e.g.,* text-to-image synthesis, CycleGAN, StackGAN, and SRGAN), and reinforcement learning algorithms (*e.g.,* Deep Q-Network, REINFORCE, and Asynchronous Advantage Actor-Critic (A3C)).

[16] `https://github.com/tensorlayer/tensorlayer/graphs/contributors`

Chapter 7

Conclusion

7.1 Summary of Thesis

Recently, deep learning technologies have achieved great success by outperforming many non-deep learning methods in many applications [1, 3, 13, 25, 27, 29]. Despite the availability of deep neural networks, it remains challenging to learn from limited training data, especially for generative tasks [2, 25]. One reason for this is because it is expensive to collect labelled data for supervised training and may be impossible to collect such labelled data in some cases. Apart from the challenge of limited training data, another challenge faced by generative tasks is that results are difficult to control [2, 16, 17]. Moreover, deep learning development is explored around experimentation, so researches continuously experiment with new evaluation metrics, new model architectures, new training pipelines, and new hyper-parameter settings [21]. As deep learning algorithms become more complex, researchers require more effort to deal with deep learning development. The studies in this thesis target the improvement in the practicability of deep learning by exploring efficient deep learning algorithms for controllable generative tasks with limited training data as well as designing a deep learning library for efficient deep learning development. The contributions from the thesis include the following:

- A method for text-to-image synthesis that synthesises images using the object attribute information from the input text descriptions. The results demonstrated that without extra domain knowledge, such as more manual labelled data or pre-defined synonyms and grammar, image's visual quality could be improved. Moreover, we also demonstrated that our method, for the first time, successfully learned the text-to-image synthesis on an unlabelled image dataset via

transfer learning.

- A method for image-to-image translation that learns to synthesise images using the semantic visual information from the input images in an unsupervised way. We show that by first training a generator to synthesise images for images of two different domains, and then train an encoder that maps images to a latent representation, then we can achieve image-to-image translation without using paired images to supervise the training. We illustrated our method using applications of face swapping, portrait gender transformation, and image inpainting.

- A method for semantic image synthesis that synthesises images using both the object attribute information and the semantic visual information from the input text descriptions and images, respectively. Instead of requiring the synthesised images to be the specific ground truth images, the generator learns to change the input images by learning to fool the discriminator. By doing so, we show that, without using any ground truth images, our method successfully achieved semantic image synthesis. We also demonstrated that our method could learn the features of the images and text descriptions to produce smooth interpolation results.

- A deep learning library, called TensorLayer, for efficient deep learning development. The model and pre-trained model abstractions reduce the workload for defining models and implementing new layers. By abstracting deep learning workflow into the three components of the model, dataset and task, a life-cycle management tool helps users to manage the deep learning workflow, and, more importantly, it helps facilitate the deep learning development by running multiple tasks concurrently. We demonstrated its effectiveness by comparing it with other libraries. Our result also demonstrated that our pre-trained model abstraction is easier to use and can avoid unnecessary use of computer memory. In the end, we demonstrated the effectiveness of our life-cycle management tool using two case studies of hyper-parameter selection and deep reinforcement learning. This library is being used in the studies of other chapters of this thesis.

These methods have been shown to improve the efficient use of data, controllable generative tasks as well as efficient deep learning development. Even though all of our generative tasks are based on images, we hope that the methods in this thesis can be reused to solve problems with different data types and help inspire future works.

7.2 Future Works

While this thesis proposed methods for efficient use of data and a new deep learning development framework, it remains clear that many challenges exist before we can achieve general artificial intelligence. To make further progress, we believe the following three research areas should be addressed next.

7.2.1 Unsupervised and weakly-supervised methods for concept learning

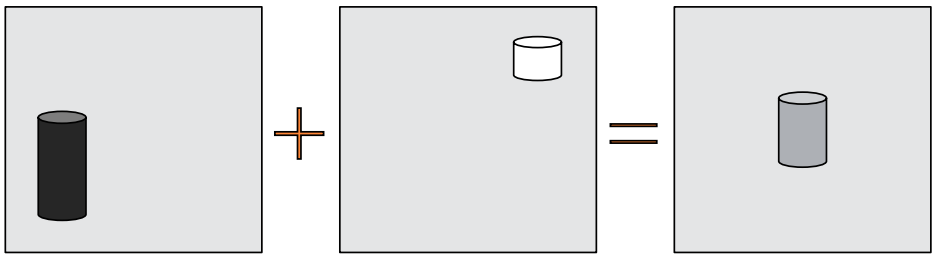

Figure 7.1: An example of interpolating the high-level concept of two images. Concepts include the height, colour, and location of the object. Given a tall and black cylinder on the bottom-left, and a short and white cylinder on the top-right, their averaged interpolation is expected to be a grey and medium height cylinder located in the middle.

As Richard Feynman said, "what I cannot create, I do not understand.". We believe general artificial intelligence is the combination of machine creativity and machine learning. What a computer cannot generate is what it cannot understand. A human can learn object concepts such as colour, location, and size, and imagine new data samples, such as given a red cup, the human can easily imagine how a cup in different colours should appear. Therefore, it is meaningful if machines can learn to create new data from the given data.

Even though our semantic image synthesis work [166] successfully learned the locations and colour of the different parts of an object (*e.g.,* the location and colour of the petals of flowers) without a known the ground truth, many other concepts, such as the size, shape, and behaviour of the object, are difficult to be learned in unsupervised or weakly-supervised ways. Therefore, future research is to investigate new methods for concept learning in unsupervised and weakly-supervised scenarios. Based on this research, we will further investigate the applications in the computer vision field. Details of this plan are included in the following.

- Learn interpolatable and explainable latent representations for high-level concepts, such as location, size, and shape of the object. Figure 7.1 shows an example of interpolating two images with two different cylinders, and we expect the colour, shape, and location can be changed linearly. To our best knowledge, there does not exist a method that can achieve these tasks in unsupervised or weakly-supervised approaches.

- Given the interpolatable latent representation, we will improve the performance of computer vision applications. For example, the video frame interpolation [175], which input two frames to output their middle frame, if the latent representation contains the information of human action, then the interpolating results will have a smooth change in the human activity.

- Create new data that can be used for further learning. Given the interpolatable latent representation, the interpolation can synthesise new data samples that do not exist in the original dataset. We can study how to use these synthesised data as the auxiliary information for improving the model training.

Overall, we believe the investigation on unsupervised and weakly-supervised concept learning is meaningful for both machine learning theory and computer vision applications.

7.2.2 Combing deep learning with computer graphic techniques

Previous studies [41,166,171] have used many data augmentation methods, which are all based on real data samples. A new research direction is to combine deep learning and computer graphic techniques for simulating data that can be used for developing computer vision applications [116]. Using simulated data from the virtual environment to develop reinforcement learning algorithms have shown great success [12, 164] as well as attention from the computer vision community [115, 116]. For example, a recent study [116] utilised the screenshots from a video game (GTA5) to synthesise a dataset for image segmentation tasks. Combing deep learning and computer graphic techniques can potentially achieve data-free learning which trains models without using real data.

Moreover, comparing with two-dimensional images, the labelling of three-dimensional scene data is more labour-intensive, which can limit the development of three-dimensional vision tasks [115]. On the other hand, three-dimensional environment simulation is a mature technique in computer graphic field where the advanced computer graphic algorithms synthesise plausible images and automatically model

large-scale scenes, such as procedural modelling of cities [176]. The simulated environment can provide zero-error three-dimensional labelled data to facilitate deep learning development in three-dimensional vision tasks. The details for the plan are included in the following.

- Study the methods to improve the simulated images by making it closer to the real images. Even though there are some studies, including CycleGAN [25], that can be used to improve the simulated images [116], synthesising ultra-high-resolution (*e.g.,* 4096 × 4096) and complex images in an unsupervised way remains still unsolved. It may be interesting to combine unsupervised and supervised techniques to achieve this goal.

- Apart from learning on the improved simulated images, we can investigate the methods to train deep learning models directly on simulated data, and then use domain adaptation method [177] to allow the model work for the real environment. This study can reduce the data requirement of many computer vision tasks, such as traffic analysis [177] and pose estimation [115].

- Study three-dimensional objects and scene synthesis. Previous generative studies [90, 166, 171] were based on two-dimensional images. As our world consists of three-dimensional physical objects, using three-dimensional information to represent the physical objects is more accurate than for the two-dimensional image. For example, rotating an object only needs to change one parameter in three-dimensional information, but must change all pixels in the two-dimensional image. Therefore, combining computer graphics with deep learning for studying unsupervised and weakly-supervised concept learning methods for synthesising three-dimensional objects and scenes is interesting to explore in the future work.

Overall, computer graphics can provide a fully controllable three-dimensional environment, we believe it has the potential to largely reduce the data required for training the deep neural networks.

7.2.3 Next generation deep learning development platform

Given these plans, we believe the next generation deep learning platform should the training of deep neural networks as well as the data generation by combing computer graphics with deep learning techniques. There are existing simulators for training artificial intelligence systems, such as AirSim [1]

[1] `https://github.com/microsoft/airsim`

from Microsoft for autonomous car research, and PyBullet [2] for robotics simulation and reinforcement learning research. However, they do not combine deep learning techniques to improve the simulated data further. A recent study from VisDa [116] showned the potential of combing computer graphics and GAN techniques to improve the simulated data for further learning. Therefore, to facilitate our deep learning research in computer vision and concept learning, we plan to develop a simulator that simulates data and improves the simulated data in real-time.

Moreover, existing distributed training methods are designed for training a single network [178–180]. Even though these methods can be used to speed the training of multi-networks, it will be interesting to develop the distributed training methods specifically for speeding up multi-networks. GANs, as one of the most successful networks in recent years, is an example of multi-networks that consists of at least a generator and discriminator [13, 25, 26, 166]. There are many variants of GAN, for example, CycleGAN [25], that requires two generators and two discriminators, and SRGAN [13] requires a generator, a discriminator, and an image encoder (*i.e.,* VGG). So, it will be interesting to develop a distributed training method that can fit with the training of all multi-networks.

In addition, to further improve the deep learning development efficiency and popularise deep learning techniques, the following three main functions must be considered for inclusion into TensorLayer. First, despite CNN models beging currently provided for fast feedforward propagation, such as MobileNet [46] and SqueezeNet [47], there exist many advanced methods to accelerate the deep neural network. For example, recent studies in channel pruning [163] can compress and accelerate the trained models, which is useful in practice. Providing a simple API that allows developers to compress their trained models before deploying automatically is also of great interest. Second, to speed the process of production, we could provide tools for model deployment, such as by converting models into TensorRT [3] for float16 and int8 forward propagation. Third, apart from providing domain-specific layers, models for common deep learning applications, such as object detection, pose estimation, speech-to-text, and text-to-speech could be provided. These advanced layers would help developers without a strong deep learning background to build their applications quickly.

Overall, the above development of a simulator, model APIs, and distributing a training method can improve the three components of deep learning workflow, including the dataset, model and training pipeline. These features are all essential to deep learning development and should be supported by

[2] `https://pybullet.org`
[3] `https://developer.nvidia.com/tensorrt`

future release.

Appendix A

Appendix

A.1 Supplementary Information for Chapter 3

A.1.1 Implementation

The network architectures are defined as follow:

```
image_size = 64                    # image size
vocab_size = 12000                 # vocabulary size
word_embedding_size = 512          # word embedding size,
rnn_hidden_size = 256              # lstm hidden size
K = 300                            # embedding size for image and text mapping
keep_prob = 0.7                    # dropout keeping probability
z_dim = 100                        # noise dimension
t_dim = 128                        # text representation dimension
c_dim = 3                          # 3 channels for rgb image

# the generator for text-to-image synthesis
def generator_txt2img_resnet(input_z, net_rnn_embed=None, is_train=True, reuse=False):
    s = image_size # output image size [64]
    s2, s4, s8, s16 = int(s/2), int(s/4), int(s/8), int(s/16)
    gf_dim = 196
    w_init = tf.random_normal_initializer(stddev=0.02)
    gamma_init = tf.random_normal_initializer(1., 0.02)

    with tf.variable_scope("generator", reuse=reuse):
```

```
tl.layers.set_name_reuse(reuse)
net_in = InputLayer(input_z, name='g_inputz')

if net_rnn_embed is not None:
    net_rnn_embed = DenseLayer(net_rnn_embed, n_units=t_dim,
            act=lambda x: tl.act.lrelu(x, 0.2), W_init = w_init, name='g_reduce_text/dense')
    net_in = ConcatLayer([net_in, net_rnn_embed], concat_dim=1, name='g_concat_z_seq')
else:
    raise Exception("No text info is used")

net_h0 = DenseLayer(net_in, gf_dim*8*s16*s16, act=tf.identity,
     netG:add(SpatialFullConvolution(opt.nz + opt.nt, ngf * 8, 4, 4))
        W_init=w_init, name='g_h0/dense')
net_h0 = ReshapeLayer(net_h0, [-1, s16, s16, gf_dim*8], name='g_h0/reshape')
net_h0 = BatchNormLayer(net_h0,
         is_train=is_train, gamma_init=gamma_init, name='g_h0/batch_norm')

net_h1 = Conv2d(net_h0, gf_dim*2, (1, 1), (1, 1),
        padding='VALID', act=None, W_init=w_init, name='g_h1/conv2d')
net_h1 = BatchNormLayer(net_h1, act=tf.nn.relu, is_train=is_train,
        gamma_init=gamma_init, name='g_h1/batch_norm')
net_h2 = Conv2d(net_h1, gf_dim*2, (3, 3), (1, 1),
        padding='SAME', act=None, W_init=w_init, name='g_h2/conv2d')
net_h2 = BatchNormLayer(net_h2, act=tf.nn.relu, is_train=is_train,
        gamma_init=gamma_init, name='g_h2/batch_norm')
net_h3 = Conv2d(net_h2, gf_dim*8, (3, 3), (1, 1),
        padding='SAME', act=None, W_init=w_init, name='g_h3/conv2d')
net_h3 = BatchNormLayer(net_h3,
         is_train=is_train, gamma_init=gamma_init, name='g_h3/batch_norm')
net_h3.outputs = tf.add(net_h3.outputs, net_h0.outputs)
net_h3.outputs = tf.nn.relu(net_h3.outputs)

net_h4 = DeConv2d(net_h3, gf_dim*4, (4, 4), out_size=(s8, s8), strides=(2, 2),
        padding='SAME', batch_size=batch_size, act=None, W_init=w_init, name='g_h4/decon2d')
net_h4 = BatchNormLayer(net_h4,
         is_train=is_train, gamma_init=gamma_init, name='g_h4/batch_norm')

net_h5 = Conv2d(net_h4, gf_dim, (1, 1), (1, 1),
        padding='VALID', act=None, W_init=w_init, name='g_h5/conv2d')
```

```
        net_h5 = BatchNormLayer(net_h5, act=tf.nn.relu, is_train=is_train,
                gamma_init=gamma_init, name='g_h5/batch_norm')
        net_h6 = Conv2d(net_h5, gf_dim, (3, 3), (1, 1),
                padding='SAME', act=None, W_init=w_init, name='g_h6/conv2d')
        net_h6 = BatchNormLayer(net_h6, act=tf.nn.relu, is_train=is_train,
                gamma_init=gamma_init, name='g_h6/batch_norm')
        net_h7 = Conv2d(net_h6, gf_dim*4, (3, 3), (1, 1),
                padding='SAME', act=None, W_init=w_init, name='g_h7/conv2d')
        net_h7 = BatchNormLayer(net_h7,
                is_train=is_train, gamma_init=gamma_init, name='g_h7/batch_norm')
        net_h7.outputs = tf.add(net_h4.outputs, net_h7.outputs)

        net_h8 = DeConv2d(net_h7, gf_dim*2, (4, 4), out_size=(s4, s4), strides=(2, 2),
                padding='SAME', batch_size=batch_size, act=None, W_init=w_init, name='g_h8/decon2d')
        net_h8 = BatchNormLayer(net_h8, act=tf.nn.relu, is_train=is_train,
                gamma_init=gamma_init, name='g_h8/batch_norm')

        net_h9 = DeConv2d(net_h8, gf_dim, (4, 4), out_size=(s2, s2), strides=(2, 2),
                padding='SAME', batch_size=batch_size, act=None, W_init=w_init, name='g_h9/decon2d')
        net_h9 = BatchNormLayer(net_h9, act=tf.nn.relu, is_train=is_train,
                gamma_init=gamma_init, name='g_h9/batch_norm')

        net_ho = DeConv2d(net_h9, c_dim, (4, 4), out_size=(s, s), strides=(2, 2),
                padding='SAME', batch_size=batch_size, act=None, W_init=w_init, name='g_ho/decon2d')
        logits = net_ho.outputs
        net_ho.outputs = tf.nn.tanh(net_ho.outputs)
    return net_ho, logits

# the discriminator for text-to-image synthesis
def discriminator_txt2img_resnet(input_images, net_rnn_embed=None, is_train=True, reuse=False):
    w_init = tf.random_normal_initializer(stddev=0.02)
    gamma_init=tf.random_normal_initializer(1., 0.02)
    df_dim = 196
    with tf.variable_scope("discriminator", reuse=reuse):
        tl.layers.set_name_reuse(reuse)
        net_in = InputLayer(input_images, name='d_input/images')
        net_h0 = Conv2d(net_in, df_dim, (4, 4), (2, 2), act=lambda x: tl.act.lrelu(x, 0.2),
                padding='SAME', W_init=w_init, name='d_h0/conv2d')
        net_h1 = Conv2d(net_h0, df_dim*2, (4, 4), (2, 2), act=None,
```

```
            padding='SAME', W_init=w_init, name='d_h1/conv2d')
    net_h1 = BatchNormLayer(net_h1, act=lambda x: tl.act.lrelu(x, 0.2),
            is_train=is_train, gamma_init=gamma_init, name='d_h1/batchnorm')
    net_h2 = Conv2d(net_h1, df_dim*4, (4, 4), (2, 2), act=None,
            padding='SAME', W_init=w_init, name='d_h2/conv2d')
    net_h2 = BatchNormLayer(net_h2, act=lambda x: tl.act.lrelu(x, 0.2),
            is_train=is_train, gamma_init=gamma_init, name='d_h2/batchnorm')
    net_h3 = Conv2d(net_h2, df_dim*8, (4, 4), (2, 2), act=None,
            padding='SAME', W_init=w_init, name='d_h3/conv2d')
    net_h3 = BatchNormLayer(net_h3,
            is_train=is_train, gamma_init=gamma_init, name='d_h3/batchnorm')

    # resnet
    net_h = Conv2d(net_h3, df_dim*2, (1, 1), (1, 1), act=None,
            padding='VALID', W_init=w_init, name='d_h3/conv2d2')
    net_h = BatchNormLayer(net_h, act=lambda x: tl.act.lrelu(x, 0.2),
            is_train=is_train, gamma_init=gamma_init, name='d_h3/batchnorm2')
    net_h = Conv2d(net_h, df_dim*2, (3, 3), (1, 1), act=None,
            padding='SAME', W_init=w_init, name='d_h3/conv2d3')
    net_h = BatchNormLayer(net_h, act=lambda x: tl.act.lrelu(x, 0.2),
            is_train=is_train, gamma_init=gamma_init, name='d_h3/batchnorm3')
    net_h = Conv2d(net_h, df_dim*8, (3, 3), (1, 1), act=None,
            padding='SAME', W_init=w_init, name='d_h3/conv2d4')
    net_h = BatchNormLayer(net_h,
            is_train=is_train, gamma_init=gamma_init, name='d_h3/batchnorm4')
    net_h3.outputs = tl.act.lrelu( tf.add(net_h.outputs, net_h3.outputs), 0.2)

    if net_rnn_embed is not None:
        net_reduced_text = DenseLayer(net_rnn_embed, n_units=t_dim,
                act=lambda x: tl.act.lrelu(x, 0.2),
                W_init=w_init, name='d_reduce_txt/dense')
        net_reduced_text = ExpandDimsLayer(net_reduced_text, axis=1, name='expand_dims1')
        net_reduced_text = ExpandDimsLayer(net_reduced_text, axis=2, name='expand_dims2')
        net_reduced_text = TileLayer(net_reduced_text, multiples=[1, 4, 4, 1], name='tile')

        net_h3_concat = ConcatLayer([net_h3, net_reduced_text], concat_dim=3, name='d_h3_concat')
        net_h3 = Conv2d(net_h3_concat, df_dim*8, (1, 1), (1, 1),
                padding='VALID', W_init=w_init, name='d_h3/conv2d_2')
        net_h3 = BatchNormLayer(net_h3, act=lambda x: tl.act.lrelu(x, 0.2),
```

```
                is_train =is_train , gamma_init=gamma_init, name='d_h3/batch_norm_2')
        else:
            raise Exception("No text info is used")
        net_h4 = Conv2d(net_h3, 1, (4, 4), (1, 1), padding='VALID', W_init=w_init, name='d_h4/conv2d_2')
        net_h4 = FlattenLayer(net_h4, name='d_h4/flatten')
        logits = net_h4.outputs
        net_h4.outputs = tf.nn.sigmoid(net_h4.outputs)
    return net_h4, logits

# the image encoder for training text encoder
def cnn(input_imgs, is_train , reuse):
    from tensorflow.contrib.slim.python.slim.nets.inception_v3 import inception_v3_base, inception_v3_arg_scope
    tl.layers.set_name_reuse(reuse)
    with slim.arg_scope(inception_v3_arg_scope()):
        net_img_in = tl.layers.InputLayer(input_imgs, name='input_image_layer')
        network = tl.layers.SlimNetsLayer(layer=net_img_in, slim_layer=inception_v3,
                                          slim_args= {
                                              'trainable' : is_train ,
                                              'is_training ' : is_train ,
                                              'reuse' : reuse,
                                             },
                                          name='InceptionV3')# (64, 2048)
    return network

def cnn_embed(network, reuse):
    with tf.variable_scope("rnn", reuse=reuse):
        network = DenseLayer(network, n_units = K,
                             act = tf.identity , W_init = initializer ,
                             b_init = None, name='image_embedding')
    return network

# the text encoder
def rnn_embed(input_seqs, is_train , reuse, return_embed=True):
    w_init = tf.random_normal_initializer(stddev=0.02)
    with tf.variable_scope("rnn", reuse=reuse):
        tl.layers.set_name_reuse(reuse)
        network = EmbeddingInputlayer(
                    inputs = input_seqs,
                    vocabulary_size = vocab_size,
```

```
                    embedding_size = word_embedding_size,
                    E_init = w_init,
                    name = 'wordembed')
        network = DynamicRNNLayer(network,
                    cell_fn = tf.nn.rnn_cell.LSTMCell,
                    n_hidden = rnn_hidden_size,
                    dropout = (keep_prob if is_train else None),
                    initializer = w_init,
                    sequence_length = tl.layers.retrieve_seq_length_op2(input_seqs),
                    return_last = True,
                    name = 'dynamic')
    if return_embed:
        with tf.variable_scope("rnn", reuse=reuse):
            net_embed = DenseLayer(network, n_units = K,
                                act = tf.identity, W_init = initializer,
                                b_init = None, name='hidden_state_embedding')
            return net_embed
    else:
        return network
```

The training script is as follows:

```
import math
import os
import numpy as np
import scipy
import time
from PIL import Image
import logging
LOG_FILENAME = 'record.log'
logging.basicConfig(filename=LOG_FILENAME, level=logging.DEBUG)

import tensorflow as tf
import tensorlayer as tl
from tensorlayer.prepro import *
from tensorlayer.layers import *
import nltk
from model_im2txt import *
```

```
from utils import *

# load image files list
images_train_dir = 'path to images'
images_train_list = tl. files . load_file_list (path=images_train_dir, regx='\\.jpg', printable=False)
images_train_list = [images_train_dir + s for s in images_train_list ]
images_train_list = np.asarray( images_train_list )
n_images_train = len( images_train_list )

checkpoint_path = "path to the checkpoint"
vocab_file = "path to the vocabulary file"
mode = 'inference' # the mode of image captioning module
top_k = 3

## Build graph for pre-trained image captioning module
images = tf.placeholder('float32', [batch_size, image_height, image_width, 3])
input_seqs = tf.placeholder(dtype=tf.int64, shape=[batch_size, None], name='input_seqs')
net_image_embeddings = Build_Image_Embeddings(mode, images, train_inception=False)
net_seq_embeddings = Build_Seq_Embeddings(input_seqs)
softmax, net_img_rnn, net_seq_rnn, state_feed = Build_Model(mode, net_image_embeddings, net_seq_embeddings,
    target_seqs=None, input_mask=None)

if tf. gfile .IsDirectory(checkpoint_path):
    checkpoint_path = checkpoint_path + '/model.ckpt-1000000'
    if not checkpoint_path:
        raise ValueError("No im2txt checkpoint file found in: %s" % checkpoint_path)

saver = tf. train .Saver()
def _restore_fn (sess):
    tf .logging. info("Loading model from im2txt checkpoint: %s", checkpoint_path)
    saver. restore(sess, checkpoint_path)
    tf .logging. info("Successfully loaded im2txt checkpoint: %s",
                      os.path.basename(checkpoint_path))

restore_fn = _restore_fn

## get the vocabulary.
vocab = tl.nlp.Vocabulary(vocab_file)
```

```
## define the graph for image and text mapping
t_rnn_image = tf.placeholder('float32', [batch_size, image_height, image_width, 3], name = 'image_txt/image')
t_rnn_image_w = tf.placeholder('float32', [batch_size, image_height, image_width, 3], name = 'image_txt/image_w')
t_rnn_caption = tf.placeholder(dtype=tf.int64, shape=[batch_size, None], name='image_txt/text')
t_rnn_caption_w = tf.placeholder(dtype=tf.int64, shape=[batch_size, None], name='image_txt/text_w')

net_cnn = cnn_embed(cnn(t_rnn_image, is_train=False, reuse=True), reuse=False)
x = net_cnn.outputs
net_rnn = rnn_embed(t_rnn_caption, is_train=True, reuse=False)
v = net_rnn.outputs
x_w = cnn_embed(cnn(t_rnn_image_w, is_train=False, reuse=True), reuse=True).outputs
v_w = rnn_embed(t_rnn_caption_w, is_train=True, reuse=True).outputs

alpha = 0.2 # margin alpha
e_loss = tf.reduce_mean(tf.maximum(0., alpha - cosine_similarity(x, v) + cosine_similarity(x, v_w))) + \
         tf.reduce_mean(tf.maximum(0., alpha - cosine_similarity(x_w, v)))

## define the graph for text-to-image synthesis.
t_real_image = tf.placeholder('float32', [batch_size, image_size, image_size, 3], name = 'real_image')
t_wrong_image = tf.placeholder('float32', [batch_size, image_size, image_size, 3], name = 'wrong_image')
t_real_caption = tf.placeholder(dtype=tf.int64, shape=[batch_size, None], name='real_caption_input')
t_z = tf.placeholder(tf.float32, [batch_size, z_dim], name='z_noise')

net_rnn2 = rnn_embed(t_real_caption, is_train=False, reuse=True, return_embed=False)
net_fake_image, _ = generator_txt2img_resnet(t_z,
                    net_rnn2,
                    is_train=True, reuse=False)
net_d, disc_fake_image_logits = discriminator_txt2img_resnet(
                    net_fake_image.outputs,
                    net_rnn2,
                    is_train=True, reuse=False)
_, disc_real_image_logits = discriminator_txt2img_resnet(
                    t_real_image,
                    net_rnn2,
                    is_train=True, reuse=True)
_, disc_wrong_image_logits = discriminator_txt2img_resnet(
                    t_wrong_image,
                    net_rnn2,
                    is_train=True, reuse=True)
```

```
# testing inference for txt2img
net_g, _ = generator_txt2img_resnet(t_z,
                    rnn_embed(t_real_caption, is_train =False, reuse=True, return_embed=False),
                     is_train =False, reuse=True)

## loss for text-to-image synthesis
d_loss1 = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits( disc_real_image_logits ,
      tf. ones_like ( disc_real_image_logits )))
d_loss2 = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(disc_wrong_image_logits,
      tf. zeros_like (disc_wrong_image_logits)))
d_loss3 = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits( disc_fake_image_logits ,
      tf. zeros_like ( disc_fake_image_logits )))

cls_weight = 0.5
d_loss = d_loss1 + cls_weight * d_loss2 + (1-cls_weight) * d_loss3

g_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits( disc_fake_image_logits ,
      tf. ones_like ( disc_fake_image_logits )))

net_fake_image.print_params(False)
net_fake_image. print_layers ()

## the cost for txt2im (real = 1, fake = 0)
lr = 0.0002     # initial learning rate for adam
beta1 = 0.5     # momentum term of adam
lr_decay = 0.5  # decay factor for adam
decay_every = 40  # decay every number of epoch
n_step_epoch = int(80000/batch_size)
e_vars = tl. layers .get_variables_with_name('rnn', True, True)
d_vars = tl. layers .get_variables_with_name('discriminator', True, True)
g_vars = tl. layers .get_variables_with_name('generator', True, True)

with tf. variable_scope('learning_rate'):
    lr_v = tf. Variable( lr , trainable=False)
d_optim = tf.train .AdamOptimizer(lr_v, beta1=beta1).minimize(d_loss, var_list=d_vars )
g_optim = tf.train .AdamOptimizer(lr_v, beta1=beta1).minimize(g_loss, var_list=g_vars )
grads, _ = tf.clip_by_global_norm(tf.gradients( e_loss , e_vars), 10)
optimizer = tf. train .AdamOptimizer(lr_v, beta1=beta1)
```

```
e_optim = optimizer.apply_gradients(zip(grads, e_vars))

with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    ## Restore the im2txt model from checkpoint.
    restore_fn(sess)

    ## Restore the txt2im model from checkpoint
    net_c_name = os.path.join(save_dir, 'net_c.npz')
    net_e_name = os.path.join(save_dir, 'net_e.npz')
    net_g_name = os.path.join(save_dir, 'net_g.npz')
    net_d_name = os.path.join(save_dir, 'net_d.npz')
    if True:
        print(" [*] Loading RNN checkpoints ...")
        if not (os.path.exists(net_e_name) and os.path.exists(net_c_name)):
            print(" [!] Loading RNN checkpoints failed!")
        else:
            net_c_loaded_params = tl.files.load_npz(name=net_c_name)
            tl.files.assign_params(sess, net_c_loaded_params, net_cnn)
            net_e_loaded_params = tl.files.load_npz(name=net_e_name)
            tl.files.assign_params(sess, net_e_loaded_params, net_rnn)
            print(" [*] Loading RNN checkpoints SUCCESS!")
        print(" [*] Loading G and D checkpoints ...")
        if not (os.path.exists(net_g_name) and os.path.exists(net_d_name)):
            print(" [!] Loading G and D checkpoints failed!")
        else:
            net_d_loaded_params = tl.files.load_npz(name=net_d_name)
            tl.files.assign_params(sess, net_d_loaded_params, net_d)
            print(" [*] Loading D checkpoints SUCCESS!")
            net_g_loaded_params = tl.files.load_npz(name=net_g_name)
            tl.files.assign_params(sess, net_g_loaded_params, net_g)
            print(" [*] Loading G checkpoints SUCCESS!")

    max_caption_length = 25
    n_step = 1000000
    n_check_step = 1
    total_d_loss, total_g_loss, total_e_loss = 0, 0, 0
    total_e_loss = 0
    total_d1, total_d2, total_d3 = 0, 0, 0
```

```
errD, d1, d2, d3, errG, errE = 0, 0, 0, 0, 0, 0
for step in range(0, n_step):
    try:
        if step !=0 and ((step / n_step_epoch) % decay_every == 0):
            new_lr_decay = lr_decay ** ((step / n_step_epoch) // decay_every)
            sess.run(tf.assign(lr_v, lr * new_lr_decay))
            log = " ** new learning rate: %f" % (lr * new_lr_decay)
            print(log)
            logging.debug(log)
        elif step == 0:
            log = " ** init lr: %f, n_step_epoch: %d, decay_every_epoch: %d, lr_decay: %f" % (lr, n_step_epoch,
                decay_every, lr_decay)
            print(log)
            logging.debug(log)

        start_time = time.time()

        idexs = get_random_int(min=0, max=n_images_train-1, number=batch_size)
        b_image_file_name = images_train_list[idexs]
        b_images = threading_data(b_image_file_name, prepro_img, mode='read_image')
        b_images = threading_data(b_images, prepro_img, mode='random_size_to_346')
        b_images_im2txt = threading_data(b_images, prepro_img, mode='346_central_crop_to_299')

        idexs_w = get_random_int(min=0, max=n_images_train-1, number=batch_size)
        b_image_file_name_w = images_train_list[idexs_w]
        b_images_w = threading_data(b_image_file_name_w, prepro_img, mode='read_image')
        b_images_w = threading_data(b_images_w, prepro_img, mode='random_size_to_346')
        b_images_rnn_w = threading_data(b_images_w, prepro_img, mode='346_random_flip_crop_to_299')
        b_images_txt2im_w = threading_data(b_images_rnn_w, prepro_img, mode='resize_to_64')

        init_state = sess.run(net_img_rnn.final_state, feed_dict={images: b_images_im2txt})
        state = np.hstack((init_state.c, init_state.h))
        ids = [[vocab.start_id]] * batch_size

        b_sentences = [[] for _ in range(batch_size)]
        b_sentences_ids = [[] for _ in range(batch_size)]
        for _ in range(max_caption_length - 1):
            softmax_output, state = sess.run([softmax, net_seq_rnn.final_state],
                                             feed_dict={input_seqs: ids,
```

```
                                        state_feed : state,
                                        })
        state = np.hstack((state.c, state.h))
        ids = []
        temp = threading_data(softmax_output, sample_fn, top_k=top_k, vocab=vocab)
        i = 0
        for a_id, word in temp:
            b_sentences[i].append(word)
            b_sentences_ids[i].append(int(a_id))
            ids = ids + [[a_id]]
            i = i + 1

    b_sentences_ids = process_sequences(b_sentences_ids, end_id=vocab.end_id, pad_val=0, is_shorten=True,
        remain_end_id=True)
    b_images_rnn = threading_data(b_images, prepro_img, mode='346_random_flip_crop_to_299')
    b_images_txt2im = threading_data(b_images_rnn, prepro_img, mode='resize_to_64')
    b_sentences_ids_w = b_sentences_ids[-1:]+b_sentences_ids[:-1]
    b_z = np.random.normal(loc=0.0, scale=1.0, size=(sample_size, z_dim)).astype(np.float32)

    # update RNN - text-image mapping
    for _ in range(1):
        errE, _ = sess.run([e_loss, e_optim], feed_dict={
                                        t_rnn_image : b_images_rnn,
                                        t_rnn_image_w : b_images_rnn_w,
                                        t_rnn_caption : b_sentences_ids,
                                        t_rnn_caption_w : b_sentences_ids_w,
                                        })
        total_e_loss += errE

    # update D
    errD, d1, d2, d3, _ = sess.run([d_loss, d_loss1, d_loss2, d_loss3, d_optim], feed_dict={
                        t_real_image : b_images_txt2im,
                        t_wrong_image : b_images_txt2im_w,
                        t_real_caption : b_sentences_ids,
                        t_z : b_z})

    # update G
    errG, _ = sess.run([g_loss, g_optim], feed_dict={
                        t_real_caption : b_sentences_ids,
```

```
                    t_z : b_z})

            total_d_loss += errD; total_g_loss += errG; total_e_loss += errE
            total_d1 += d1; total_d2 += d2; total_d3 += d3
            print("step %d: d_loss: %.4f (%.3f, %.3f, %.3f), g_loss: %.4f, e_loss: %.5f (%.2f sec), ept: %d" %
                (step, errD, d1, d2, d3, errG, errE, time.time()-start_time, prepro_img_failed_counter))
            if step != 0 and step % n_check_step == 0:
                ## print loss
                log = " ** avg step: %d d_loss: %.4f (%.3f, %.3f, %.3f), g_loss: %.4f, e_loss: %.5f: " % (step,
                    total_d_loss/n_check_step,
                        total_d1/n_check_step, total_d2/n_check_step, total_d3/n_check_step,
                            total_g_loss/n_check_step, total_e_loss/n_check_step)
                logging.debug(log)
                total_d_loss, total_g_loss, total_e_loss = 0, 0, 0
                total_d1, total_d2, total_d3 = 0, 0, 0
                ## save model to npz format
                if step % (n_check_step*10) == 0:
                    tl.files.save_npz(net_cnn.all_params, name=net_c_name, sess=sess)
                    tl.files.save_npz(net_rnn.all_params, name=net_e_name, sess=sess)
                    tl.files.save_npz(net_g.all_params, name=net_g_name, sess=sess)
                    tl.files.save_npz(net_d.all_params, name=net_d_name, sess=sess)
                    print(" [*] Saving txt2im checkpoints SUCCESS!")

        except Exception as err:
            print(err)
```

A.2 Supplementary Information for Chapter 4

A.2.1 Implementation

The network architectures are defined as follow:

```
import tensorflow as tf
import tensorlayer as tl
from tensorlayer.layers import *

# the architecture of generator
def generator(inputs, is_train =True, reuse=False):/
```

```
FLAGS = tf.app.flags.FLAGS
image_size = 64
s2, s4, s8, s16 = int(image_size/2), int(image_size/4), int(image_size/8), int(image_size/16)
gf_dim = 128
c_dim = FLAGS.c_dim
batch_size = FLAGS.batch_size

w_init = tf.random_normal_initializer(stddev=0.02)
b_init = None
gamma_init = tf.random_normal_initializer(1., 0.02)

with tf.variable_scope("generator", reuse=reuse):
    tl.layers.set_name_reuse(reuse)
    net_in = InputLayer(inputs, name='g/in')
    net_h0 = DenseLayer(net_in, n_units=gf_dim*8*s16*s16, W_init=w_init, b_init=b_init,
        act = tf.identity, name='g/h0/lin')
    net_h0 = ReshapeLayer(net_h0, shape=[-1, s16, s16, gf_dim*8], name='g/h0/reshape')
    net_h0 = BatchNormLayer(net_h0, act=tf.nn.relu, is_train=is_train,
        gamma_init=gamma_init, name='g/h0/batch_norm')

    net_h1 = DeConv2d(net_h0, gf_dim*4, (5, 5), out_size=(s8, s8), strides=(2, 2),
        padding='SAME', batch_size=batch_size, act=None, W_init=w_init, b_init=b_init, name='g/h1/decon2d')
    net_h1 = BatchNormLayer(net_h1, act=tf.nn.relu, is_train=is_train,
        gamma_init=gamma_init, name='g/h1/batch_norm')

    net_h2 = DeConv2d(net_h1, gf_dim*2, (5, 5), out_size=(s4, s4), strides=(2, 2),
        padding='SAME', batch_size=batch_size, act=None, W_init=w_init, b_init=b_init, name='g/h2/decon2d')
    net_h2 = BatchNormLayer(net_h2, act=tf.nn.relu, is_train=is_train,
        gamma_init=gamma_init, name='g/h2/batch_norm')

    net_h3 = DeConv2d(net_h2, gf_dim, (5, 5), out_size=(s2, s2), strides=(2, 2),
        padding='SAME', batch_size=batch_size, act=None, W_init=w_init, b_init=b_init, name='g/h3/decon2d')
    net_h3 = BatchNormLayer(net_h3, act=tf.nn.relu, is_train=is_train,
        gamma_init=gamma_init, name='g/h3/batch_norm')

    net_h4 = DeConv2d(net_h3, c_dim, (5, 5), out_size=(image_size, image_size), strides=(2, 2),
        padding='SAME', batch_size=batch_size, act=None, W_init=w_init, name='g/h4/decon2d')
    logits = net_h4.outputs
    net_h4.outputs = tf.nn.tanh(net_h4.outputs)
```

```
    return net_h4, logits

# the architecture of discriminator
def discriminator(inputs, is_train=True, reuse=False):
    FLAGS = tf.app.flags.FLAGS
    df_dim = 64 # Dimension of discrim filters in first conv layer. [64]
    c_dim = FLAGS.c_dim # n_color 3
    batch_size = FLAGS.batch_size # 64

    w_init = tf.random_normal_initializer(stddev=0.02)
    b_init = None
    gamma_init = tf.random_normal_initializer(1., 0.02)

    with tf.variable_scope("discriminator", reuse=reuse):
        tl.layers.set_name_reuse(reuse)

        net_in = InputLayer(inputs, name='d/in')
        net_h0 = Conv2d(net_in, df_dim, (5, 5), (2, 2), act=lambda x: tl.act.lrelu(x, 0.2),
            padding='SAME', W_init=w_init, name='d/h0/conv2d')

        net_h1 = Conv2d(net_h0, df_dim*2, (5, 5), (2, 2), act=None,
            padding='SAME', W_init=w_init, b_init=b_init, name='d/h1/conv2d')
        net_h1 = BatchNormLayer(net_h1, act=lambda x: tl.act.lrelu(x, 0.2),
            is_train=is_train, gamma_init=gamma_init, name='d/h1/batch_norm')

        net_h2 = Conv2d(net_h1, df_dim*4, (5, 5), (2, 2), act=None,
            padding='SAME', W_init=w_init, b_init=b_init, name='d/h2/conv2d')
        net_h2 = BatchNormLayer(net_h2, act=lambda x: tl.act.lrelu(x, 0.2),
            is_train=is_train, gamma_init=gamma_init, name='d/h2/batch_norm')

        net_h3 = Conv2d(net_h2, df_dim*8, (5, 5), (2, 2), act=None,
            padding='SAME', W_init=w_init, b_init=b_init, name='d/h3/conv2d')
        net_h3 = BatchNormLayer(net_h3, act=lambda x: tl.act.lrelu(x, 0.2),
            is_train=is_train, gamma_init=gamma_init, name='d/h3/batch_norm')

        net_h4 = FlattenLayer(net_h3, name='d/h4/flatten')
        net_h4 = DenseLayer(net_h4, n_units=1, act=tf.identity,
            W_init = w_init, name='d/h4/output_real_fake')
```

```
        logits = net_h4.outputs
        net_h4.outputs = tf.nn.sigmoid(net_h4.outputs)

        net_h5 = FlattenLayer(net_h3, name='d/h5/flatten')
        net_h5 = DenseLayer(net_h5, n_units=2, act=tf.identity,
            W_init = w_init, name='d/h5/output_classes')
        logits2 = net_h5.outputs
        net_h5.outputs = tf.nn.softmax(net_h5.outputs)
    return net_h4, logits, net_h5, logits2, net_h3

# the architecture of encoder
def imageEncoder(inputs, is_train=True, reuse=False):
    # it uses the same architecure with the discriminator except the output layer
    FLAGS = tf.app.flags.FLAGS
    df_dim = 64
    c_dim = FLAGS.c_dim
    batch_size = FLAGS.batch_size

    w_init = tf.random_normal_initializer(stddev=0.02)
    b_init = None
    gamma_init = tf.random_normal_initializer(1., 0.02)

    with tf.variable_scope("imageEncoder", reuse=reuse):
        tl.layers.set_name_reuse(reuse)

        net_in = InputLayer(inputs, name='p/in')
        net_h0 = Conv2d(net_in, df_dim, (5, 5), (2, 2), act=lambda x: tl.act.lrelu(x, 0.2),
            padding='SAME', W_init=w_init, name='p/h0/conv2d')

        net_h1 = Conv2d(net_h0, df_dim*2, (5, 5), (2, 2), act=None,
            padding='SAME', W_init=w_init, b_init=b_init, name='p/h1/conv2d')
        net_h1 = BatchNormLayer(net_h1, act=lambda x: tl.act.lrelu(x, 0.2),
            is_train=is_train, gamma_init=gamma_init, name='p/h1/batch_norm')

        net_h2 = Conv2d(net_h1, df_dim*4, (5, 5), (2, 2), act=None,
            padding='SAME', W_init=w_init, b_init=b_init, name='p/h2/conv2d')
        net_h2 = BatchNormLayer(net_h2, act=lambda x: tl.act.lrelu(x, 0.2),
            is_train=is_train, gamma_init=gamma_init, name='p/h2/batch_norm')
```

```
        net_h3 = Conv2d(net_h2, df_dim*8, (5, 5), (2, 2), act=None,
            padding='SAME', W_init=w_init, b_init=b_init, name='p/h3/conv2d')
        net_h3 = BatchNormLayer(net_h3, act=lambda x: tl.act.lrelu(x, 0.2),
            is_train =is_train , gamma_init=gamma_init, name='p/h3/batch_norm')

        net_h4 = FlattenLayer(net_h3, name='p/h4/flatten')
        net_h4 = DenseLayer(net_h4, n_units=FLAGS.z_dim)
        W_init = w_init, name='p/h4/output_real_fake')
    return net_h4
```

The training script is as follows:

```
import os
import pprint
import numpy as np
import tensorflow as tf
import tensorlayer as tl
from tensorlayer.layers import *
from tensorlayer.prepro import *
from random import shuffle
import argparse

pp = pprint.PrettyPrinter()

flags = tf.app.flags
flags.DEFINE_integer("epoch", 100, "Epoch to train [100]")
flags.DEFINE_float("learning_rate", 0.0002, "Learning rate of for adam [0.0002]")
flags.DEFINE_float("beta1", 0.5, "Momentum term of adam [0.5]")
flags.DEFINE_integer("batch_size", 64, "The number of batch images [64]")
flags.DEFINE_integer("image_size", 64, "The size of image to use (will be center cropped) [64]")
flags.DEFINE_integer("z_dim", 100, "Size of Noise embedding")
flags.DEFINE_integer("class_embedding_size", 5, "Size of class embedding")
flags.DEFINE_integer("sample_size", 64, "The number of sample images [64]")
flags.DEFINE_integer("c_dim", 3, "Dimension of image color. [3]")
flags.DEFINE_integer("sample_step", 500, "The interval of generating sample. [500]")
flags.DEFINE_integer("save_step", 100, "The interval of saveing checkpoints. [200]")
flags.DEFINE_integer("imageEncoder_steps", 30000, "Number of train steps for image encoder")
flags.DEFINE_string("dataset", "svhn", "The name of dataset [celebA, obama_hillary, svhn]")
flags.DEFINE_string("checkpoint_dir", "data/Models", "Directory name to save the checkpoints [checkpoint]")
```

```
flags.DEFINE_string("sample_dir", "data/samples", "Directory name to save the image samples [samples]")

FLAGS = flags.FLAGS

os.system('mkdir data')
os.system('mkdir {}'.format(FLAGS.sample_dir))
os.system('mkdir {}'.format(FLAGS.sample_dir+'/step1'))
os.system('mkdir {}'.format(FLAGS.sample_dir+'/step2'))

import data_loader
import model
from utils import *

generator = model.generator
discriminator = model.discriminator
imageEncoder = model.imageEncoder

def train_ac_gan():
    z_dim = FLAGS.z_dim
    z_noise = tf.placeholder(tf.float32, [FLAGS.batch_size, z_dim], name='z_noise')
    z_classes = tf.placeholder(tf.int64, shape=[FLAGS.batch_size, ], name='z_classes')
    real_images = tf.placeholder(tf.float32, [FLAGS.batch_size, FLAGS.image_size, FLAGS.image_size,
        FLAGS.c_dim], name='real_images')

    # z embedding
    if FLAGS.class_embedding_size != None:
        net_z_classes = EmbeddingInputlayer(inputs = z_classes, vocabulary_size = 2, embedding_size =
            FLAGS.class_embedding_size, name ='classes_embedding')
    else:
        net_z_classes = InputLayer(inputs = tf.one_hot(z_classes, 2), name ='classes_embedding')

    # z --> generator for training
    net_g, _ = generator(tf.concat(1, [z_noise, net_z_classes.outputs]), is_train =True, reuse=False)

    # generated fake images --> discriminator
    net_d, d_logits_fake, _, d_logits_fake_class, _ = discriminator(net_g.outputs, is_train =True, reuse=False)

    # real images --> discriminator
    _, d_logits_real, _, d_logits_real_class, _ = discriminator(real_images, is_train =True, reuse=True)
```

```
# sample_z --> generator for evaluation, set is_train to False
net_g2, _ = generator(tf.concat(1, [z_noise, net_z_classes.outputs]), is_train=False, reuse=True)

# cost for updating discriminator and generator
# discriminator: real images are labelled as 1
d_loss_real = tl.cost.sigmoid_cross_entropy(d_logits_real, tf.ones_like(d_logits_real), name='dreal')
# discriminator: images from generator (fake) are labelled as 0
d_loss_fake = tl.cost.sigmoid_cross_entropy(d_logits_fake, tf.zeros_like(d_logits_fake), name='dfake')

d_loss = d_loss_real + d_loss_fake + d_loss_class

# generator: try to make the the fake images look real (1)
g_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(d_logits_fake,
    tf.ones_like(d_logits_fake)))
g_loss_class = tl.cost.cross_entropy(d_logits_fake_class, z_classes, name='g')
g_loss = g_loss_fake + g_loss_class

t_vars = tf.trainable_variables()
g_vars = [var for var in t_vars if 'generator' in var.name]
e_vars = [var for var in t_vars if 'classes_embedding' in var.name]
d_vars = [var for var in t_vars if 'discriminator' in var.name]

# optimizers for updating discriminator and generator
d_optim = tf.train.AdamOptimizer(FLAGS.learning_rate, beta1=FLAGS.beta1) \
                  .minimize(d_loss, var_list=d_vars)
g_optim = tf.train.AdamOptimizer(FLAGS.learning_rate, beta1=FLAGS.beta1) \
                  .minimize(g_loss, var_list=g_vars + e_vars)

sess=tf.Session()
tl.ops.set_gpu_fraction(sess=sess, gpu_fraction=0.5)
sess.run(tf.initialize_all_variables())

net_g_name = os.path.join(FLAGS.checkpoint_dir, '{}_net_g.npz'.format(FLAGS.dataset))
net_d_name = os.path.join(FLAGS.checkpoint_dir, '{}_net_d.npz'.format(FLAGS.dataset))
net_e_name = os.path.join(FLAGS.checkpoint_dir, '{}_net_e.npz'.format(FLAGS.dataset))

if not FLAGS.retrain:
    if not (os.path.exists(net_g_name) and os.path.exists(net_d_name) and os.path.exists(net_e_name)):
```

```
            print(" [!] Could not load weights from npz files")
        else:
            net_g_loaded_params = tl.files.load_npz(name=net_g_name)
            net_d_loaded_params = tl.files.load_npz(name=net_d_name)
            net_e_loaded_params = tl.files.load_npz(name=net_e_name)
            tl.files.assign_params(sess, net_g_loaded_params, net_g)
            tl.files.assign_params(sess, net_d_loaded_params, net_d)
            tl.files.assign_params(sess, net_e_loaded_params, net_z_classes)
            print(" [*] Loading checkpoints SUCCESS!")
    else:
        print(" [*] Retraining AC GAN")

    class1_files, class2_files, class_flag = data_loader.load_data(FLAGS.dataset, split = "train")
    is_class_balance = True

    if is_class_balance:
        total_batches = 2 * max(len(class1_files), len(class2_files))/FLAGS.batch_size
        class1_files = np.asarray(class1_files)
        class2_files = np.asarray(class2_files)
    else:
        all_files = class1_files + class2_files
        total_batches = len(all_files)/FLAGS.batch_size
        shuffle(all_files)
        print(" all_files ", len(all_files))

    print("Total_batches", total_batches)

    for epoch in range(FLAGS.epoch):
        for bn in range(0, int(total_batches)):

            if is_class_balance:
                idex = get_random_int(min=0, max=len(class1_files)-1, number=int(FLAGS.batch_size/2)) #/2)
                batch_files = class1_files[idex]
                idex = get_random_int(min=0, max=len(class2_files)-1, number=int(FLAGS.batch_size/2)) #/2)
                batch_files = np.concatenate((batch_files, class2_files[idex]))
            else:
                batch_files = all_files[bn*FLAGS.batch_size : (bn + 1) * FLAGS.batch_size]

            batch_z = np.random.normal(loc=0.0, scale=1.0, size=(FLAGS.sample_size, z_dim)).astype(np.float32)
```

```
batch_images = threading_data(batch_files, fn=get_image_fn)
batch_images = threading_data(batch_images, fn=distort_fn)

if "svhn" in FLAGS.dataset:
    batch_images[:int(FLAGS.batch_size/2)] = threading_data(batch_images[:int(FLAGS.batch_size/2)],
        fn=add_noise_fn, keep=0.8)
    batch_z_classes = [0]*int(FLAGS.batch_size/2) + [1]*int(FLAGS.batch_size/2)
else:
    batch_z_classes = [0 if class_flag[file_name] == True else 1 for file_name in batch_files]

errD, _ = sess.run([d_loss, d_optim], feed_dict={
    z_noise: batch_z,
    z_classes : batch_z_classes,
    real_images: batch_images
})

for _ in range(2):
    errG, _ = sess.run([g_loss, g_optim], feed_dict={
        z_noise: batch_z,
        z_classes : batch_z_classes,
    })

print("d_loss={}\t g_loss={}\t epoch={}\t batch_no={}\t total_batches={}".format(errD, errG, epoch,
    bn, total_batches))

if bn % FLAGS.save_step == 0:
    print("[*] Saving Models...")

    tl.files.save_npz(net_g.all_params, name=net_g_name, sess=sess)
    tl.files.save_npz(net_d.all_params, name=net_d_name, sess=sess)
    tl.files.save_npz(net_z_classes.all_params, name=net_e_name, sess=sess)

    # Saving after each iteration
    tl.files.save_npz(net_g.all_params, name=net_g_name + "_" + str(epoch), sess=sess)
    tl.files.save_npz(net_d.all_params, name=net_d_name + "_" + str(epoch), sess=sess)
    tl.files.save_npz(net_z_classes.all_params, name=net_e_name + "_" + str(epoch), sess=sess)

    print("[*] Models saved")
```

```
                generated_samples = sess.run([net_g2.outputs], feed_dict={
                    z_noise: batch_z,
                    z_classes : batch_z_classes,
                })[0]

                generated_samples_other_class = sess.run([net_g2.outputs], feed_dict={
                    z_noise: batch_z,
                    z_classes : [0 if batch_z_classes[i] == 1 else 1 for i in range(len(batch_z_classes))],
                })[0]

                combine_and_save_image_sets( [batch_images, generated_samples, generated_samples_other_class],
                    FLAGS.sample_dir+'/step1')

def train_imageEncoder():
    z_dim = FLAGS.z_dim
    z_noise = tf.placeholder(tf.float32, [FLAGS.batch_size, z_dim], name='z_noise')
    z_classes = tf.placeholder(tf.int64, shape=[FLAGS.batch_size, ], name='z_classes')

    if FLAGS.class_embedding_size != None:
        net_z_classes = EmbeddingInputlayer(inputs = z_classes, vocabulary_size = 2, embedding_size =
            FLAGS.class_embedding_size, name ='classes_embedding')
    else:
        net_z_classes = InputLayer(inputs = tf.one_hot(z_classes, 2), name ='classes_embedding')

    net_g, _ = generator(tf.concat(1, [z_noise, net_z_classes.outputs]), is_train=False, reuse=False)
    net_p = imageEncoder(net_g.outputs, is_train=True)
    net_g2, _ = generator(tf.concat(1, [net_p.outputs, net_z_classes.outputs]), is_train=False, reuse=True)

    t_vars = tf.trainable_variables()
    p_vars = [var for var in t_vars if 'imageEncoder' in var.name]
    p_loss = tf.reduce_mean( tf.square( tf.sub( net_p.outputs, z_noise) ))
    p_optim = tf.train.AdamOptimizer(FLAGS.learning_rate/2, beta1=FLAGS.beta1) \
                .minimize(p_loss, var_list=p_vars)

    sess = tf.Session()
    tl.ops.set_gpu_fraction(sess=sess, gpu_fraction=0.5)
    tl.layers.initialize_global_variables(sess)

    # restore the trained ACGAN
```

```
net_g_name = os.path.join(FLAGS.checkpoint_dir, '{}_net_g.npz'.format(FLAGS.dataset))
net_e_name = os.path.join(FLAGS.checkpoint_dir, '{}_net_e.npz'.format(FLAGS.dataset))

if not (os.path.exists(net_g_name) and os.path.exists(net_e_name)):
    print(" [!] Loading checkpoints failed!")
    return
else:
    net_g_loaded_params = tl.files.load_npz(name=net_g_name)
    net_e_loaded_params = tl.files.load_npz(name=net_e_name)
    tl.files.assign_params(sess, net_g_loaded_params, net_g2)
    tl.files.assign_params(sess, net_e_loaded_params, net_z_classes)
    print(" [*] Loading checkpoints SUCCESS!")

net_p_name = os.path.join(FLAGS.checkpoint_dir, '{}_net_p.npz'.format(FLAGS.dataset))
if not FLAGS.retrain:
    net_p_loaded_params = tl.files.load_npz(name=net_p_name)
    tl.files.assign_params(sess, net_p_loaded_params, net_p)
    print(" [*] Loaded Pretrained Image Encoder!")
else:
    print(" [*] Retraining ImageEncoder")

model_no = 0
for step in range(0, FLAGS.imageEncoder_steps):
    batch_z_classes = [0 if random.random() > 0.5 else 1 for i in range(FLAGS.batch_size)]
    batch_z = np.random.normal(loc=0.0, scale=1.0, size=(FLAGS.sample_size, z_dim)).astype(np.float32)

    batch_images, gen_images, _, errP = sess.run([net_g.outputs, net_g2.outputs, p_optim, p_loss], feed_dict={
        z_noise : batch_z,
        z_classes : batch_z_classes,
    })

    print("p_loss={}\t step_no={}\t total_steps={}".format(errP, step, FLAGS.imageEncoder_steps))

    if step % FLAGS.sample_step == 0:
        print(" [*] Sampling images")
        combine_and_save_image_sets([batch_images, gen_images], FLAGS.sample_dir+'/step2')

    if step % 2000 == 0:
```

```
            model_no += 1

        if step % FLAGS.save_step == 0:
            print(" [*] Saving Model")
            tl.files.save_npz(net_p.all_params, name=net_p_name, sess=sess)
            tl.files.save_npz(net_p.all_params, name=net_p_name + "_" + str(model_no), sess=sess)
            print(" [*] Model p(encoder) saved")

def main():
    parser = argparse.ArgumentParser()

    parser.add_argument('--train_step', type=str, default="ac_gan",
                        help='Step of the training : ac_gan, imageEncoder')

    parser.add_argument('--retrain', type=int, default=0,
                        help='Set 0 for using pre-trained model, 1 for retraining the model')

    args = parser.parse_args()

    if not os.path.exists(FLAGS.checkpoint_dir):
        os.makedirs(FLAGS.checkpoint_dir)
    if not os.path.exists(FLAGS.sample_dir):
        os.makedirs(FLAGS.sample_dir)

    if args.train_step == "ac_gan":
        train_ac_gan()

    elif args.train_step == "imageEncoder":
        train_imageEncoder()

if __name__ == '__main__':
    main()
```

A.3 Supplementary Information for Chapter 5

We illustrate more success and failed 64×64 results on the Oxford-102 flower and Caltech-200 bird datasets, and discuss the possible reasons for the failure. The summary of failure can help to under-

stand our limitation.

A.3.1 Additional results

We illustrate more flower images with diverse shapes in Figure A.1. When the text description contains shape information *e.g.* "skinny yellow petals", the proposed method successfully maintains the outline of flowers and change only the related part *i.e.* the petals. Compared with bird dataset, in flower dataset, the performance of background reconstruction is poorer, except some simple backgrounds *e.g.* the fourth column. The reason is that comparing with the backgrounds from bird dataset such as wire, branch and sky, the backgrounds of flower dataset mainly contain leaves with arbitrary and diverse shapes, which is more challenging.

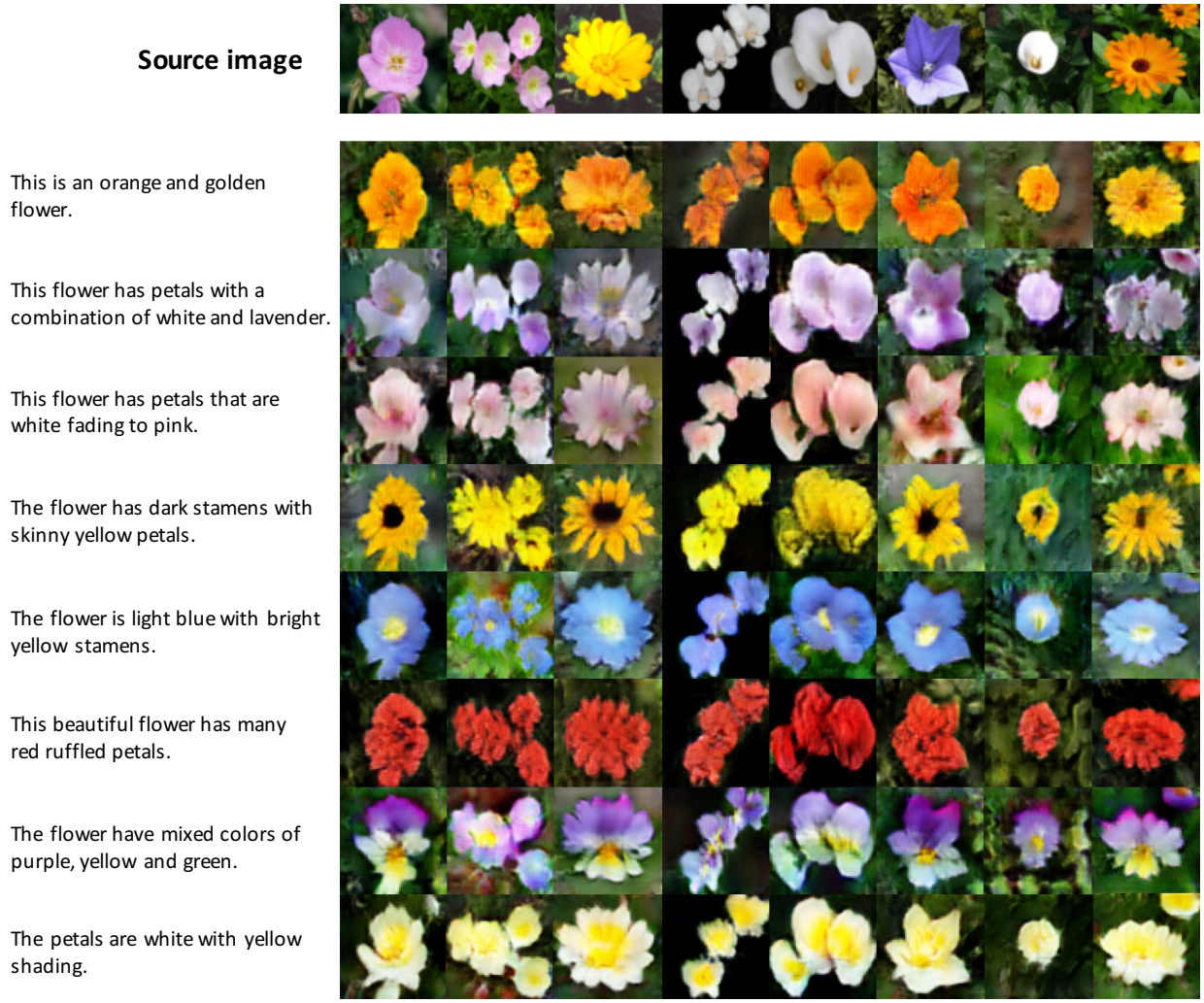

Figure A.1: Additional zero-shot 64 × 64 results of semantic image synthesis without pre-trained VGG encoder on the Oxford-102 flower dataset.

For Caltech-200 bird dataset, first, our method can correctly locate different parts of birds. Second, compared with flower dataset, the background reconstruction of bird dataset is better, it can correctly reconstruct the object in the background with rare colour *e.g.* the red objects in second and sixth columns of Figure A.2. The third column demonstrated that even the background object is small or thin, our method still able to reconstruct it.

Figure A.2: Additional zero-shot 64 × 64 results of semantic image synthesis without pre-trained VGG encoder on the Caltech-200 bird dataset.

A.3.2 Failure cases

Figure A.3 and A.4 show the failure cases of both Oxford-102 flower and Caltech-200 bird datasets. The main reasons for failure cases are:

- **Complex and unrecognised background information:** Take the second example of the bottom row of Figure A.3 for example, as the blue background is rare in the flower dataset, the synthesised image fails to maintain the background and generate the green background which is

common in the flower dataset. For bird failure cases in Figure A.4, the car park background in the top-left example is also rare in the bird dataset, resulting the red car become a part of the bird in the synthesised image.

- **Diverse and complicated shape of objects:** In Figure A.4, as the second example of bottom row shows, even the background is simple, the bird with an uncommon size will lead to a failed result. Similarly, the top-right flower in Figure A.3 has complex petals, the synthesised flower can maintain the general shape, but if we look closely, we can found that the shape details totally changed.

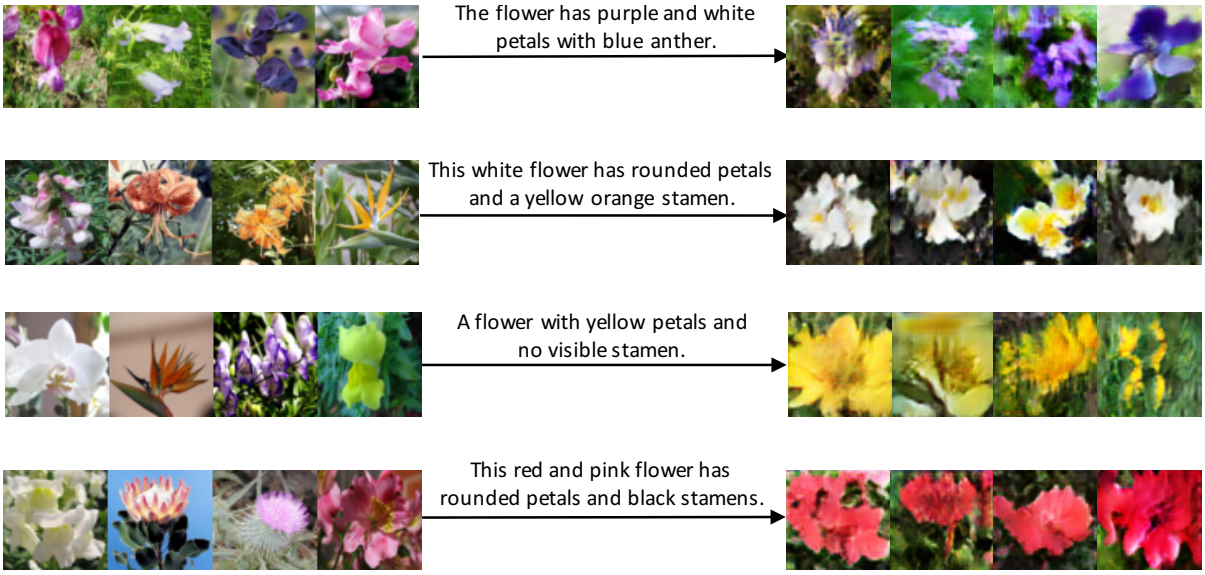

Figure A.3: Failure cases of zero-shot 64×64 results of semantic image synthesis without pre-trained VGG encoder on the Oxford-102 flower dataset.

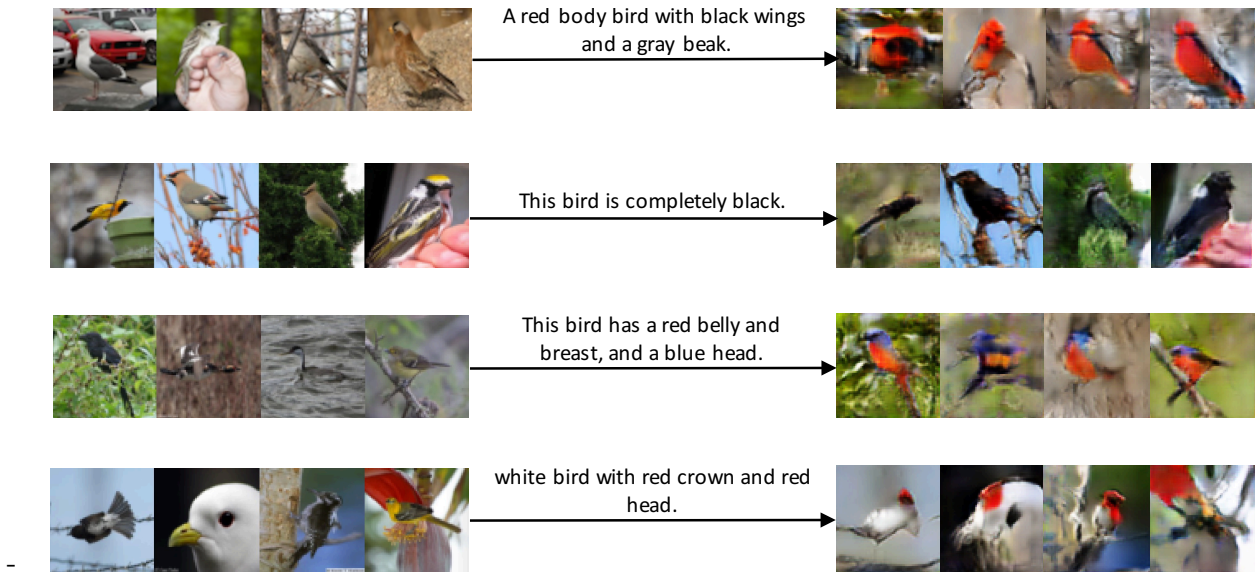

Figure A.4: Failure cases of zero-shot 64×64 results of semantic image synthesis without pre-trained VGG encoder on the Caltech-200 bird dataset.

A.3.3 Implementation

The network architectures for synthesising both 64 × 64 and 256 × 256 images.

```
import tensorflow as tf
import tensorlayer as tl
from tensorlayer.layers import *
import os
# hyper-parameters for text-to-image mapping
t_dim = 128
rnn_hidden_size = t_dim
vocab_size = 8000
word_embedding_size = 256

# the text encoder
def rnn_embed(input_seqs, is_train=True, reuse=False):
    w_init = tf.random_normal_initializer(stddev=0.02)
    LSTMCell = tf.contrib.rnn.BasicLSTMCell
    with tf.variable_scope("rnnftxt", reuse=reuse):
        tl.layers.set_name_reuse(reuse)
        net = EmbeddingInputlayer(
                    inputs = input_seqs,
                    vocabulary_size = vocab_size,
                    embedding_size = word_embedding_size,
```

```
                E_init = w_init,
                name = 'rnn/wordembed')
        net = DynamicRNNLayer(net,
                cell_fn = LSTMCell,
                cell_init_args = {'state_is_tuple' : True, 'reuse': reuse},
                n_hidden = rnn_hidden_size,
                dropout = None,#(keep_prob if is_train else None),
                initializer = w_init,
                sequence_length = tl.layers.retrieve_seq_length_op2(input_seqs),
                return_last = True,
                name = 'rnn/dynamic')
        return net

# the image encoder for training the text encoder
def cnn_encoder(inputs, is_train=True, reuse=False, name='cnnftxt'):
    w_init = tf.random_normal_initializer(stddev=0.02)
    gamma_init = tf.random_normal_initializer(1., 0.02)
    df_dim = 64

    with tf.variable_scope(name, reuse=reuse):
        tl.layers.set_name_reuse(True)

        net_in = InputLayer(inputs, name='/in')
        net_h0 = Conv2d(net_in, df_dim, (4, 4), (2, 2), act=lambda x: tl.act.lrelu(x, 0.2),
                padding='SAME', W_init=w_init, name='cnnf/h0/conv2d')

        net_h1 = Conv2d(net_h0, df_dim*2, (4, 4), (2, 2), act=None,
                padding='SAME', W_init=w_init, b_init=None, name='cnnf/h1/conv2d')
        net_h1 = BatchNormLayer(net_h1, act=lambda x: tl.act.lrelu(x, 0.2),
                is_train=is_train, gamma_init=gamma_init, name='cnnf/h1/batch_norm')

        net_h2 = Conv2d(net_h1, df_dim*4, (4, 4), (2, 2), act=None,
                padding='SAME', W_init=w_init, b_init=None, name='cnnf/h2/conv2d')
        net_h2 = BatchNormLayer(net_h2, act=lambda x: tl.act.lrelu(x, 0.2),
                is_train=is_train, gamma_init=gamma_init, name='cnnf/h2/batch_norm')

        net_h3 = Conv2d(net_h2, df_dim*8, (4, 4), (2, 2), act=None,
                padding='SAME', W_init=w_init, b_init=None, name='cnnf/h3/conv2d')
        net_h3 = BatchNormLayer(net_h3, act=lambda x: tl.act.lrelu(x, 0.2),
```

```
                is_train =is_train , gamma_init=gamma_init, name='cnnf/h3/batch_norm')

        net_h4 = FlattenLayer(net_h3, name='cnnf/h4/flatten')
        net_h4 = DenseLayer(net_h4, n_units=t_dim,
                W_init = w_init, b_init = None, name='cnnf/h4/embed')
        return net_h4

# the generator without VGG for 64x64 images
def encode_generator(inputs, t_txt=None, is_train=True, reuse=False, batch_size=None):
    gf_dim = 128
    w_init = tf.random_normal_initializer(stddev=0.02)
    b_init = None
    gamma_init=tf.random_normal_initializer(1., 0.02)

    with tf.variable_scope("encode_generator", reuse=reuse):
        tl.layers.set_name_reuse(reuse)
        net_in = InputLayer(inputs, name='g/images')
        ## downsampling
        net_h0 = Conv2d(net_in, gf_dim, (3, 3), (1, 1), act=tf.nn.relu,
                padding='SAME', W_init=w_init, name='g_h0/conv2d')
        net_h1 = Conv2d(net_h0, gf_dim*2, (4, 4), (2, 2), act=None,
                padding='SAME', W_init=w_init, b_init=b_init, name='g_h1/conv2d')
        net_h1 = BatchNormLayer(net_h1, act=tf.nn.relu,
                is_train =is_train , gamma_init=gamma_init, name='g_h1/batchnorm')
        net_h2 = Conv2d(net_h1, gf_dim*4, (4, 4), (2, 2), act=None,
                padding='SAME', W_init=w_init, b_init=b_init, name='g_h2/conv2d')
        net_h2 = BatchNormLayer(net_h2, act=tf.nn.relu,
                is_train =is_train , gamma_init=gamma_init, name='g_h2/batchnorm')

        ## join image and text representations
        if t_txt is not None:
            net_txt = InputLayer(t_txt, name='g_join/input_text')
            net_txt = DenseLayer(net_txt, n_units=t_dim * 2,
                    act=lambda x: tl.act.lrelu(x, 0.2),
                    W_init=w_init, b_init=None, name='g_join_reduce_txt/dense')

            def KL_loss(mu, log_sigma):
                with tf.name_scope("KL_divergence"):
                    loss = -log_sigma + .5 * (-1 + tf.exp(2. * log_sigma) + tf.square(mu))
```

```
            loss = tf.reduce_mean(loss)
            return loss

        mean = net_txt.outputs[:, :t_dim]
        log_sigma = net_txt.outputs[:, t_dim:]
        epsilon = tf.truncated_normal(tf.shape(mean))
        stddev = tf.exp(log_sigma)
        c = mean + stddev * epsilon

        kl_loss = KL_loss(mean, log_sigma)

        net_txt_c = InputLayer(c, name='c')
        net_txt_c.all_params.extend(net_txt.all_params)

        net_txt = ExpandDimsLayer(net_txt_c, 1, name='g_join_reduce_txt/expanddim1')
        # net_txt = ExpandDimsLayer(net_txt, 1, name='g_join_reduce_txt/expanddim1')
        net_txt = ExpandDimsLayer(net_txt, 1, name='g_join_reduce_txt/expanddim2')
        net_txt = TileLayer(net_txt, [1, 16, 16, 1], name='g_join_reduce_txt/tile')
        net_h2_concat = ConcatLayer([net_h2, net_txt], concat_dim=3, name='g_join_reduce_txt/concat')
        net_h2 = Conv2d(net_h2_concat, gf_dim*4, (3, 3), (1, 1),
                padding='SAME', W_init=w_init, b_init=b_init, name='g_join/conv2d')
        net_h2 = BatchNormLayer(net_h2, act=tf.nn.relu,
                is_train=is_train, gamma_init=gamma_init, name='g_join/batch_norm')
    else:
        raise Exception("missing text embedding")

    for i in range(4):
        net = Conv2d(net_h2, gf_dim*4, (3, 3), (1, 1),
                padding='SAME', W_init=w_init, b_init=b_init, name='g_residual{}/conv2d_1'.format(i))
        net = BatchNormLayer(net, act=tf.nn.relu,
                is_train=is_train, gamma_init=gamma_init, name='g_residual{}/batch_norm_1'.format(i))
        net = Conv2d(net, gf_dim*4, (3, 3), (1, 1),
                padding='SAME', W_init=w_init, b_init=b_init, name='g_residual{}/conv2d_2'.format(i))
        net = BatchNormLayer(net,
                is_train=is_train, gamma_init=gamma_init, name='g_residual{}/batch_norm_2'.format(i))
        net_h2 = ElementwiseLayer(layer=[net_h2, net], combine_fn=tf.add, name='g_residual{}/add'.format(i))
        net_h2.outputs = tf.nn.relu(net_h2.outputs)

    net_h3 = UpSampling2dLayer(net_h2, size=[32, 32], is_scale=False, method=1, align_corners=False,
```

```
            name='g_up/upsample2d_1')
        net_h3 = Conv2d(net_h3, gf_dim*2, (3, 3), (1, 1),
                padding='SAME', W_init=w_init, b_init=b_init, name='g_up/conv2d_1')
        net_h3 = BatchNormLayer(net_h3, act=tf.nn.relu,
                is_train=is_train, gamma_init=gamma_init, name='g_up/batch_norm_1')

        net_h4 = UpSampling2dLayer(net_h3, size=[64, 64], is_scale=False, method=1, align_corners=False,
            name='g_up/upsample2d_2')
        net_h4 = Conv2d(net_h4, gf_dim, (3, 3), (1, 1),
                padding='SAME', W_init=w_init, b_init=b_init, name='g_up/conv2d_2')
        net_h4 = BatchNormLayer(net_h4, act=tf.nn.relu,
                is_train=is_train, gamma_init=gamma_init, name='g_up/batch_norm_2')

        ## down to 3 channels for RGB
        network = Conv2d(net_h4, 3, (3, 3), (1, 1),
                padding='SAME', W_init=w_init, name='g_out/conv2d')

        logits = network.outputs
        network.outputs = tf.nn.tanh(network.outputs)
    return network, logits, kl_loss

# the generator with VGG for 64x64 images
def encode_generator_244to64(inputs, t_txt=None, is_train=True, reuse=False, batch_size=None):
    gf_dim = 128
    w_init = tf.random_normal_initializer(stddev=0.02)
    b_init = None
    gamma_init=tf.random_normal_initializer(1., 0.02)

    with tf.variable_scope("generator", reuse=reuse):
        tl.layers.set_name_reuse(reuse)

        inputs = (inputs + 1) * 127.5
        mean = tf.constant([123.68, 116.779, 103.939], dtype=tf.float32, shape=[1, 1, 1, 3], name='img_mean')
        net_in = InputLayer(inputs - mean, name='vgg_input_im')
        """ conv1 """
        network = tl.layers.Conv2dLayer(net_in,
                        act = tf.nn.relu,
                        shape = [3, 3, 3, 64],
                        strides = [1, 1, 1, 1],
```

```
                padding='SAME',
                name ='vgg_conv1_1')
    network = tl.layers.Conv2dLayer(network,
                act = tf.nn.relu,
                shape = [3, 3, 64, 64],
                strides = [1, 1, 1, 1],
                padding='SAME',
                name ='vgg_conv1_2')
    network = tl.layers.PoolLayer(network,
                ksize=[1, 2, 2, 1],
                strides=[1, 2, 2, 1],
                padding='SAME',
                pool = tf.nn.max_pool,
                name ='vgg_pool1')
    """ conv2 """
    network = tl.layers.Conv2dLayer(network,
                act = tf.nn.relu,
                shape = [3, 3, 64, 128],
                strides = [1, 1, 1, 1],
                padding='SAME',
                name ='vgg_conv2_1')
    network = tl.layers.Conv2dLayer(network,
                act = tf.nn.relu,
                shape = [3, 3, 128, 128],
                strides = [1, 1, 1, 1],
                padding='SAME',
                name ='vgg_conv2_2')
    network = tl.layers.PoolLayer(network,
                ksize=[1, 2, 2, 1],
                strides=[1, 2, 2, 1],
                padding='SAME',
                pool = tf.nn.max_pool,
                name ='vgg_pool2')
    """ conv3 """
    network = tl.layers.Conv2dLayer(network,
                act = tf.nn.relu,
                shape = [3, 3, 128, 256],
                strides = [1, 1, 1, 1],
                padding='SAME',
```

```
                    name ='vgg_conv3_1')
    network = tl.layers.Conv2dLayer(network,
                    act = tf.nn.relu,
                    shape = [3, 3, 256, 256],
                    strides = [1, 1, 1, 1],
                    padding='SAME',
                    name ='vgg_conv3_2')
    network = tl.layers.Conv2dLayer(network,
                    act = tf.nn.relu,
                    shape = [3, 3, 256, 256],
                    strides = [1, 1, 1, 1],
                    padding='SAME',
                    name ='vgg_conv3_3')
    network = tl.layers.PoolLayer(network,
                    ksize=[1, 2, 2, 1],
                    strides=[1, 2, 2, 1],
                    padding='SAME',
                    pool = tf.nn.max_pool,
                    name ='vgg_pool3')
    """ conv4 """
    network = tl.layers.Conv2dLayer(network,
                    act = tf.nn.relu,
                    shape = [3, 3, 256, 512],
                    strides = [1, 1, 1, 1],
                    padding='SAME',
                    name ='vgg_conv4_1')
    network = tl.layers.Conv2dLayer(network,
                    act = tf.nn.relu,
                    shape = [3, 3, 512, 512],
                    strides = [1, 1, 1, 1],
                    padding='SAME',
                    name ='vgg_conv4_2')
    network = tl.layers.Conv2dLayer(network,
                    act = tf.nn.relu,
                    shape = [3, 3, 512, 512],
                    strides = [1, 1, 1, 1],
                    padding='SAME',
                    name ='vgg_conv4_3')
    net_h2 = tl.layers.PoolLayer(network,
```

```
                ksize=[1, 2, 2, 1],
                strides=[1, 2, 2, 1],
                padding='SAME',
                pool = tf.nn.max_pool,
                name ='vgg_pool4_4')

with tf.variable_scope("decoder", reuse=reuse):
    # join image and text representation
    if t_txt is not None:
        net_txt = InputLayer(t_txt, name='input_text')
        net_txt = DenseLayer(net_txt, n_units=t_dim * 2,
                act=lambda x: tl.act.lrelu(x, 0.2),
                W_init=w_init, b_init=None, name='dense')

        def KL_loss(mu, log_sigma):
            with tf.name_scope("KL_divergence"):
                loss = -log_sigma + .5 * (-1 + tf.exp(2. * log_sigma) + tf.square(mu))
                loss = tf.reduce_mean(loss)
                return loss

        mean = net_txt.outputs[:, :t_dim]
        log_sigma = net_txt.outputs[:, t_dim:]
        epsilon = tf.truncated_normal(tf.shape(mean))
        stddev = tf.exp(log_sigma)
        c = mean + stddev * epsilon
        kl_loss = KL_loss(mean, log_sigma)

        net_txt_c = InputLayer(c, name='c')
        net_txt_c.all_params.extend(net_txt.all_params)
        net_txt = ExpandDimsLayer(net_txt_c, 1, name='expanddim1')
        net_txt = ExpandDimsLayer(net_txt, 1, name='expanddim2')
        net_txt = TileLayer(net_txt, [1, 16, 16, 1], name='tile')
        net_h2_concat = ConcatLayer([net_h2, net_txt], concat_dim=3, name='concat') # (64, 4, 4, 640)
        net_h2 = Conv2d(net_h2_concat, gf_dim*4, (3, 3), (1, 1),
                padding='SAME', W_init=w_init, b_init=b_init, name='conv2d')
        net_h2 = BatchNormLayer(net_h2, act=tf.nn.relu,
                is_train=is_train, gamma_init=gamma_init, name='batch_norm')
    else:
        raise Exception('missing text embedding')
```

```
        ## residual block x 4(for 64--256)
        for i in range(4):
            net = Conv2d(net_h2, gf_dim*4, (3, 3), (1, 1),
                    padding='SAME', W_init=w_init, b_init=b_init, name='res{}/c1'.format(i))
            net = BatchNormLayer(net, act=tf.nn.relu,
                    is_train=is_train, gamma_init=gamma_init, name='res{}/bn1'.format(i))
            net = Conv2d(net, gf_dim*4, (3, 3), (1, 1),
                    padding='SAME', W_init=w_init, b_init=b_init, name='res{}/c2'.format(i))
            net = BatchNormLayer(net,
                    is_train=is_train, gamma_init=gamma_init, name='res{}/bn2'.format(i))
            net_h2 = ElementwiseLayer(layer=[net_h2, net], combine_fn=tf.add, name='res{}/add'.format(i))
            net_h2.outputs = tf.nn.relu(net_h2.outputs)

        net_h3 = UpSampling2dLayer(net_h2, size=[32, 32], is_scale=False, method=1, align_corners=False,
             name='up1')
        net_h3 = Conv2d(net_h3, gf_dim*2, (3, 3), (1, 1),
                padding='SAME', W_init=w_init, b_init=b_init, name='c1')
        net_h3 = BatchNormLayer(net_h3, act=tf.nn.relu,
                is_train=is_train, gamma_init=gamma_init, name='bn1')

        net_h4 = UpSampling2dLayer(net_h3, size=[64, 64], is_scale=False, method=1, align_corners=False,
             name='up2')
        net_h4 = Conv2d(net_h4, gf_dim, (3, 3), (1, 1),
                padding='SAME', W_init=w_init, b_init=b_init, name='c2')
        net_h4 = BatchNormLayer(net_h4, act=tf.nn.relu,
                is_train=is_train, gamma_init=gamma_init, name='bn2')

        ## down to 3 channels
        network = Conv2d(net_h4, 3, (3, 3), (1, 1), act=tf.nn.tanh,
                padding='SAME', W_init=w_init, name='encode_generator_244_out/conv2d')
    return network, kl_loss

# the generator for 256x256 images
def cyclegan_generator_resnet(image, t_txt=None, is_train=True, reuse=False, batch_size=None):
    b_init = None
    w_init = tf.truncated_normal_initializer(stddev=0.02)
    g_init = tf.random_normal_initializer(1., 0.02)
    _, nx, ny, nz = image.get_shape().as_list()
```

```
with tf.variable_scope('generator', reuse=reuse):
    tl.layers.set_name_reuse(reuse)
    gf_dim = 32

    net_in = InputLayer(image, name='in')

    net_c1 = Conv2d(net_in, gf_dim, (7, 7), (1, 1), act=None,
            padding='SAME', W_init=w_init, b_init=b_init, name='c7s1-32')
    net_c1 = BatchNormLayer(net_c1, act=tf.nn.relu,
             is_train=is_train, gamma_init=g_init, name='bn1')

    net_c2 = Conv2d(net_c1, gf_dim * 2, (3, 3), (2, 2), act=None,
            padding='SAME', W_init=w_init, b_init=b_init, name='d64')
    net_c2 = BatchNormLayer(net_c2, act=tf.nn.relu,
            is_train=is_train, gamma_init=g_init, name='bn2')

    net_c3 = Conv2d(net_c2, gf_dim * 4, (3, 3), (2, 2), act=None,
           padding='SAME', W_init=w_init, b_init=b_init, name='d128')
    net_c3 = BatchNormLayer(net_c3, act=tf.nn.relu,
            is_train=is_train, gamma_init=g_init, name='bn3')

    ## join image and text representation
    if t_txt is not None:
        net_txt = InputLayer(t_txt, name='txt/in')
        net_txt = DenseLayer(net_txt, n_units=t_dim,
                act=lambda x: tl.act.lrelu(x, 0.2),
                W_init=w_init, b_init=None, name='txt/den')

        net_txt_c = InputLayer(mean, name='txt/mean')
        net_txt_c.all_params.extend(net_txt.all_params)

        net_txt = ExpandDimsLayer(net_txt_c, 1, name='txt/expand1')
        net_txt = ExpandDimsLayer(net_txt, 1, name='txt/expand2')
        # net_txt = TileLayer(net_txt, [1, 64, 64, 1], name='txttile')
        net_txt = TileLayer(net_txt, [1, int(nx/4), int(ny/4), 1], name='txt/tile')
        net_c3_concat = ConcatLayer([net_c3, net_txt], concat_dim=3, name='txt/concat')
        net_c3 = Conv2d(net_c3_concat, gf_dim*4, (3, 3), (1, 1),
                padding='SAME', W_init=w_init, b_init=b_init, name='txt/c')
        net_c3 = BatchNormLayer(net_c3, act=tf.nn.relu,
```

```
                is_train=is_train, gamma_init=g_init, name='txt/bn')
        else:
            raise Exception('missing text embedding')

        n = net_c3
        for i in range(16):
            nn = Conv2d(n, gf_dim * 4, (3, 3), (1, 1), act=None,
                padding='SAME', W_init=w_init, b_init=b_init, name='res/c1/%s' % i)
            nn = BatchNormLayer(nn, act=tf.nn.relu,
                is_train=is_train, gamma_init=g_init, name='res/bn/%s_1' % i)
            nn = Conv2d(nn, gf_dim * 4, (3, 3), (1, 1), act=None,
                padding='SAME', W_init=w_init, b_init=b_init, name='res/c2/%s' % i)
            nn = BatchNormLayer(nn,
                is_train=is_train, gamma_init=g_init, name='res/bn/%s_2' % i)
            nn = ElementwiseLayer([n, nn], tf.add, 'res/add/%s' % i)
            n = nn

        net_r9 = n
        net_d1 = DeConv2d(net_r9, gf_dim * 2, (3, 3), out_size=(128,128),
            strides=(2, 2), padding='SAME', batch_size=batch_size, act=None, W_init=w_init, b_init=b_init,
                name='u64')
        net_d1 = BatchNormLayer(net_d1, act=tf.nn.relu,
            is_train=is_train, gamma_init=g_init, name='bn/d1')

        net_d2 = DeConv2d(net_d1, gf_dim, (3, 3), out_size=(256,256),
            strides=(2, 2), padding='SAME',batch_size=batch_size, act=None, W_init=w_init, b_init=b_init,
                name='u32')
        net_d2 = BatchNormLayer(net_d2, act=tf.nn.relu,
            is_train=is_train, gamma_init=g_init, name='bn/d2')

        net_c4 = Conv2d(net_d2, 3, (7, 7), (1, 1), act=tf.nn.tanh, W_init=w_init,
            padding='SAME', name='c7s1-3')
    return net_c4, None

# the discriminator for 64x64 images
def discriminator_txt2img_64(input_images, t_txt=None, is_train=True, reuse=False):
    w_init = tf.random_normal_initializer(stddev=0.02)
    gamma_init=tf.random_normal_initializer(1., 0.02)
    df_dim = 64
```

```
s = 64
s2, s4, s8, s16 = int(s/2), int(s/4), int(s/8), int(s/16)

with tf.variable_scope("discriminator", reuse=reuse):
    tl.layers.set_name_reuse(reuse)
    net_in = InputLayer(input_images, name='d_input/images')
    net_h0 = Conv2d(net_in, df_dim, (4, 4), (2, 2), act=lambda x: tl.act.lrelu(x, 0.2),
            padding='SAME', W_init=w_init, name='d_h0/conv2d')

    net_h1 = Conv2d(net_h0, df_dim*2, (4, 4), (2, 2), act=None,
            padding='SAME', W_init=w_init, b_init=None, name='d_h1/conv2d')
    net_h1 = BatchNormLayer(net_h1, act=lambda x: tl.act.lrelu(x, 0.2),
            is_train=is_train, gamma_init=gamma_init, name='d_h1/batchnorm')
    net_h2 = Conv2d(net_h1, df_dim*4, (4, 4), (2, 2), act=None,
            padding='SAME', W_init=w_init, b_init=None, name='d_h2/conv2d')
    net_h2 = BatchNormLayer(net_h2, act=lambda x: tl.act.lrelu(x, 0.2),
            is_train=is_train, gamma_init=gamma_init, name='d_h2/batchnorm')
    net_h3 = Conv2d(net_h2, df_dim*8, (4, 4), (2, 2), act=None,
            padding='SAME', W_init=w_init, b_init=None, name='d_h3/conv2d')
    net_h3 = BatchNormLayer(net_h3,
            is_train=is_train, gamma_init=gamma_init, name='d_h3/batchnorm')

    net = Conv2d(net_h3, df_dim*2, (1, 1), (1, 1), act=None,
            padding='VALID', W_init=w_init, b_init=None, name='d_h4_res/conv2d')
    net = BatchNormLayer(net, act=lambda x: tl.act.lrelu(x, 0.2),
            is_train=is_train, gamma_init=gamma_init, name='d_h4_res/batchnorm')
    net = Conv2d(net, df_dim*2, (3, 3), (1, 1), act=None,
            padding='SAME', W_init=w_init, b_init=None, name='d_h4_res/conv2d2')
    net = BatchNormLayer(net, act=lambda x: tl.act.lrelu(x, 0.2),
            is_train=is_train, gamma_init=gamma_init, name='d_h4_res/batchnorm2')
    net = Conv2d(net, df_dim*8, (3, 3), (1, 1), act=None,
            padding='SAME', W_init=w_init, b_init=None, name='d_h4_res/conv2d3')
    net = BatchNormLayer(net,
            is_train=is_train, gamma_init=gamma_init, name='d_h4_res/batchnorm3')
    net_h4 = ElementwiseLayer(layer=[net_h3, net], combine_fn=tf.add, name='d_h4/add')
    net_h4.outputs = tl.act.lrelu(net_h4.outputs, 0.2)

    net_txt = InputLayer(t_txt, name='d_input_txt')
    net_txt = DenseLayer(net_txt, n_units=t_dim,
```

```
            act=lambda x: tl.act.lrelu(x, 0.2),
            W_init=w_init, name='d_reduce_txt/dense')
        net_txt = ExpandDimsLayer(net_txt, 1, name='d_txt/expanddim1')
        net_txt = ExpandDimsLayer(net_txt, 1, name='d_txt/expanddim2')
        net_txt = TileLayer(net_txt, [1, 4, 4, 1], name='d_txt/tile')
        net_h4_concat = ConcatLayer([net_h4, net_txt], concat_dim=3, name='d_h3_concat')
        net_h4 = Conv2d(net_h4_concat, df_dim*8, (1, 1), (1, 1),
                padding='VALID', W_init=w_init, b_init=None, name='d_h3/conv2d_2')
        net_h4 = BatchNormLayer(net_h4, act=lambda x: tl.act.lrelu(x, 0.2),
                is_train=is_train, gamma_init=gamma_init, name='d_h3/batch_norm_2')

        net_ho = Conv2d(net_h4, 1, (s16, s16), (s16, s16), padding='VALID', W_init=w_init, name='d_ho/conv2d')
        logits = net_ho.outputs
        net_ho.outputs = tf.nn.sigmoid(net_ho.outputs)
    return net_ho, logits


# the discriminator for 256x256 images
def discriminator_txt2img_256(input_images, t_txt=None, is_train=True, reuse=False):
    w_init = tf.random_normal_initializer(stddev=0.02)
    b_init = None
    gamma_init = tf.random_normal_initializer(1., 0.02)
    df_dim = 32
    s2, s4, s8, s16 = int(df_dim/2), int(df_dim/4), int(df_dim/8), int(df_dim/16)

    with tf.variable_scope("discriminator", reuse=reuse):
        tl.layers.set_name_reuse(reuse)

        net_in = InputLayer(input_images, name='d_input/images')

        net_h0 = Conv2d(net_in, df_dim, (3, 3), (2, 2), act=lambda x: tl.act.lrelu(x, 0.2),
                padding='SAME', W_init=w_init, name='d_h0/conv2d')
        net_h1 = Conv2d(net_h0, df_dim*2, (3, 3), (2, 2), act=None,
                padding='SAME', W_init=w_init, b_init=b_init, name='d_h1/conv2d')
        net_h1 = BatchNormLayer(net_h1, act=lambda x: tl.act.lrelu(x, 0.2),
                is_train=is_train, gamma_init=gamma_init, name='d_h1/batchnorm')
        net_h2 = Conv2d(net_h1, df_dim*2, (3, 3), (2, 2), act=None,
                padding='SAME', W_init=w_init, b_init=b_init, name='d_h2/conv2d')
        net_h2 = BatchNormLayer(net_h2, act=lambda x: tl.act.lrelu(x, 0.2),
```

```
        is_train =is_train , gamma_init=gamma_init, name='d_h2/batchnorm')
net_h3 = Conv2d(net_h2, df_dim*4, (3, 3), (2, 2), act=None,
        padding='SAME', W_init=w_init, b_init=b_init, name='d_h3/conv2d')
net_h3 = BatchNormLayer(net_h3, act=lambda x: tl.act.lrelu(x, 0.2),
        is_train =is_train , gamma_init=gamma_init, name='d_h3/batchnorm')
net_h4 = Conv2d(net_h3, df_dim*4, (3, 3), (2, 2), act=None,
        padding='SAME', W_init=w_init, b_init=b_init, name='d_h4/conv2d')
net_h4 = BatchNormLayer(net_h4, act=lambda x: tl.act.lrelu(x, 0.2),
        is_train =is_train , gamma_init=gamma_init, name='d_h4/batchnorm')
net_h5 = Conv2d(net_h4, df_dim*4, (3, 3), (2, 2), act=None,
        padding='SAME', W_init=w_init, b_init=b_init, name='d_h5/conv2d')
net_h5 = BatchNormLayer(net_h5, act=lambda x: tl.act.lrelu(x, 0.2),
        is_train =is_train , gamma_init=gamma_init, name='d_h5/batchnorm')
net_h6 = Conv2d(net_h5, df_dim*16, (1, 1), (1, 1), act=None,
        padding='SAME', W_init=w_init, b_init=b_init, name='d_h6/conv2d')
net_h6 = BatchNormLayer(net_h6, act=lambda x: tl.act.lrelu(x, 0.2),
        is_train =is_train , gamma_init=gamma_init, name='d_h6/batchnorm')
net_h7 = Conv2d(net_h6, df_dim*8, (1, 1), (1, 1), act=None,
        padding='SAME', W_init=w_init, b_init=b_init, name='d_h7/conv2d')
net_h7 = BatchNormLayer(net_h7,
        is_train =is_train , gamma_init=gamma_init, name='d_h7/batchnorm')
net_h8 = net_h7

net_txt = InputLayer(t_txt, name='d_t_txt')
net_txt = DenseLayer(net_txt, n_units=t_dim,
        act=lambda x: tl.act. lrelu (x, 0.2) ,
        W_init=w_init, b_init=None, name='d_reduce_txt/dense')
net_txt = ExpandDimsLayer(net_txt, 1, name='d_reduce_txt/expanddim1')
net_txt = ExpandDimsLayer(net_txt, 1, name='d_reduce_txt/expanddim2')
net_txt = TileLayer(net_txt, [1, 4, 4, 1], name='d_reduce_txt/tile')

net_h8_concat = ConcatLayer([net_h8, net_txt], concat_dim=3, name='d_txt_concat')
net_h8 = Conv2d(net_h8_concat, df_dim*4, (1, 1), (1, 1),
        padding='SAME', W_init=w_init, b_init=b_init, name='d_txt/conv2d_2')
net_h8 = BatchNormLayer(net_h8, act=lambda x: tl.act.lrelu(x, 0.2),
        is_train =is_train , gamma_init=gamma_init, name='d_txt/batch_norm_2')

logits = net_ho.outputs
net_ho.outputs = tf.nn.sigmoid(net_ho.outputs)
```

```
    return net_ho, logits
```

The training script for text encoder is as follow:

```
import tensorflow as tf
import tensorlayer as tl
from tensorlayer.layers import *
import numpy as np
from scipy.io import loadmat
import time, os, re, nltk, scipy
import model

# load data
print("Loading data from pickle ...")
import pickle
with open("data/_vocab.pickle", 'rb') as f:
    vocab = pickle.load(f)
with open("data/_image_train.pickle", 'rb') as f:
    images_train = pickle.load(f)
with open("data/_image_test.pickle", 'rb') as f:
    images_test = pickle.load(f)
with open("data/_n.pickle", 'rb') as f:
    n_captions_train, n_captions_test, n_captions_per_image, n_images_train, n_images_test = pickle.load(f)
with open("data/_caption.pickle", 'rb') as f:
    captions_ids_train, captions_ids_test = pickle.load(f)

images_train = np.array(images_train)
images_test = np.array(images_test)
print(images_train.shape)

save_dir = "samples"
tl.files.exists_or_mkdir(save_dir)
checkpoint_dir = "checkpoint"
tl.files.exists_or_mkdir(checkpoint_dir)

save_dir_cnn = os.path.join(checkpoint_dir, 'rnn_encoder_cnn.npz')
save_dir_rnn = os.path.join(checkpoint_dir, 'rnn_encoder.npz')
```

```
def to64_fn(x):
    x = tl.prepro. flip_axis (x, axis=1, is_random=True)
    x = tl.prepro.rotation(x, rg=16, is_random=True, fill_mode='nearest')
    x = tl.prepro.imresize(x, size=[64+15, 64+15], interp='bicubic', mode=None)
    x = tl.prepro.crop(x, wrg=64, hrg=64, is_random=True)
    x = x / 127.5 - 1
    return x

def main_train():
    batch_size = 64
    ni = int(np. ceil (np.sqrt( batch_size )))
    t_real_image = tf.placeholder('float32', [batch_size, 64, 64, 3], name = 'matching_image')
    t_wrong_image = tf.placeholder('float32', [batch_size, 64, 64, 3], name = 'mismatching_image')
    t_real_caption = tf.placeholder(dtype=tf.int64, shape=[batch_size, None], name='matching_text')
    t_wrong_caption = tf.placeholder(dtype=tf.int64, shape=[batch_size, None], name='mismatching_text')

    # matching image
    net_cnn = model.cnn_encoder(t_real_image, is_train=True, reuse=False)
    x = net_cnn.outputs
    # matching text
    net_rnn = model.rnn_embed(t_real_caption, is_train=True, reuse=False)
    v = net_rnn.outputs
    # mismatching image
    x_w = model.cnn_encoder(t_wrong_image, is_train=True, reuse=True).outputs
    # mismatching text
    v_w = model.rnn_embed(t_wrong_caption, is_train=True, reuse=True).outputs

    alpha = 0.2 # margin alpha
    rnn_loss = tf.reduce_mean(tf.maximum(0., alpha - tl.cost.cosine_similarity(x, v)
            + tl.cost. cosine_similarity (x, v_w))) + tf.reduce_mean(tf.maximum(0., alpha -
                tl.cost.cosine_similarity(x, v) + tl.cost. cosine_similarity (x_w, v)))

    lr = 0.0002
    lr_decay = 0.5
    decay_every = 50
    beta1 = 0.5

    cnn_vars = tl.layers.get_variables_with_name('cnnftxt', True, True)
    rnn_vars = tl.layers.get_variables_with_name('rnnftxt', True, True)
```

```
with tf.variable_scope('learning_rate'):
    lr_v = tf.Variable(lr, trainable=False)

optimizer = tf.train.AdamOptimizer(lr_v, beta1=beta1)
grads, _ = tf.clip_by_global_norm(tf.gradients(rnn_loss, rnn_vars + cnn_vars), 10)
rnn_optim = optimizer.apply_gradients(zip(grads, rnn_vars + cnn_vars))

sess = tf.InteractiveSession()
tl.layers.initialize_global_variables(sess)

n_epoch = 50
print_freq = 1
n_batch_epoch = int(n_images_train / batch_size)

for epoch in range(0, n_epoch+1):
    start_time = time.time()

    if epoch !=0 and (epoch % decay_every == 0):
        new_lr_decay = lr_decay ** (epoch // decay_every)
        sess.run(tf.assign(lr_v, lr * new_lr_decay))
        log = " ** new learning rate: %f" % (lr * new_lr_decay)
        print(log)
    elif epoch == 0:
        log = " ** init lr: %f decay_every_epoch: %d, lr_decay: %f" % (lr, decay_every, lr_decay)
        print(log)

    for step in range(n_batch_epoch):
        step_time = time.time()
        ## get matching text
        idexs = tl.utils.get_random_int(min=0, max=n_captions_train-1, number=batch_size)
        b_real_caption = captions_ids_train[idexs]
        b_real_caption = tl.prepro.pad_sequences(b_real_caption, padding='post')

        ## get matching image
        b_real_images =
            images_train[np.floor(np.asarray(idexs).astype('float')/n_captions_per_image).astype('int')]

        ## get mismatching caption
```

```
            idexs = tl.utils.get_random_int(min=0, max=n_captions_train-1, number=batch_size)
            b_wrong_caption = captions_ids_train[idexs]
            b_wrong_caption = tl.prepro.pad_sequences(b_wrong_caption, padding='post')

            ## get mismatching image
            idexs2 = tl.utils.get_random_int(min=0, max=n_images_train-1, number=batch_size)
            b_wrong_images = images_train[idexs2]

            b_real_images = tl.prepro.threading_data(b_real_images, to64_fn)
            b_wrong_images = tl.prepro.threading_data(b_wrong_images, to64_fn)

            ## updates text-to-image mapping
            errRNN, _ = sess.run([rnn_loss, rnn_optim], feed_dict={
                                        t_real_image : b_real_images,
                                        t_wrong_image : b_wrong_images,
                                        t_real_caption : b_real_caption,
                                        t_wrong_caption : b_wrong_caption})

            print("Epoch: [%2d/%2d] [%4d/%4d] time: %4.4fs, rnn_loss: %.8f" \
                        % (epoch, n_epoch, step, n_batch_epoch, time.time() - step_time, errRNN))

        ## save model
        if (epoch != 0) and (epoch % 10) == 0:
            tl.files.save_npz(net_cnn.all_params, name=save_dir_cnn, sess=sess)
            tl.files.save_npz(net_rnn.all_params, name=save_dir_rnn, sess=sess)
            print("[*] Save checkpoints SUCCESS!")

if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser()

    parser.add_argument('--mode', type=str, default="train",
                        help='train, train_encoder, translation')

    args = parser.parse_args()

    main_train()
```

The training script for the generator is as follow:

```
import tensorflow as tf
import tensorlayer as tl
from tensorlayer.layers import *
from tensorlayer.prepro import *
from tensorlayer.cost import *
import numpy as np
import scipy
from scipy.io import loadmat
import time, os, re, nltk

from utils import *
from model import *
import model

t_real_image = tf.placeholder('float32', [batch_size, image_size, image_size, 3], name = 'real_image')
t_real_caption = tf.placeholder(dtype=tf.int64, shape=[batch_size, None], name='match_caption_input')
t_wrong_caption = tf.placeholder(dtype=tf.int64, shape=[batch_size, None], name='mismatch_caption_input')
t_relate_caption = tf.placeholder(dtype=tf.int64, shape=[batch_size, None], name='relate_caption_input')

discriminator_txt2img = model.discriminator_txt2img_64

net_rnn = rnn_embed(t_real_caption, is_train=False, reuse=False)
net_rnn_w = rnn_embed(t_wrong_caption, is_train=False, reuse=True)
net_rnn_a = rnn_embed(t_relate_caption, is_train=False, reuse=True)

net_fake_image, _, kl_loss = encode_generator(t_real_image,
                    net_rnn_a.outputs,
                    is_train=True, reuse=False)
net_d, disc_fake_image_logits = discriminator_txt2img(
                    net_fake_image.outputs,
                    net_rnn_a.outputs,
                    is_train=True, reuse=False)

_, disc_real_image_logits = discriminator_txt2img(
                    t_real_image, net_rnn.outputs, is_train=True, reuse=True)

_, disc_mismatch_logits = discriminator_txt2img(
                    t_real_image, net_rnn_w.outputs, is_train=True, reuse=True)
```

```
net_fake_image2, _, _ = encode_generator(t_real_image,
                    net_rnn.outputs, is_train =False, reuse=True)

# loss function
d_loss1 = tl.cost.sigmoid_cross_entropy( disc_real_image_logits , tf.ones_like ( disc_real_image_logits ), name='d1')
d_loss2 = tl.cost.sigmoid_cross_entropy(disc_mismatch_logits, tf.zeros_like (disc_mismatch_logits), name='d2')
d_loss3 = tl.cost.sigmoid_cross_entropy( disc_fake_image_logits , tf.zeros_like ( disc_fake_image_logits ), name='d3')
d_loss = d_loss1 + d_loss2 * 0.5 + d_loss3 * 0.5
g_loss2 = tl.cost.sigmoid_cross_entropy( disc_fake_image_logits , tf.ones_like ( disc_fake_image_logits ),
    name='g_any')
g_loss = g_loss2 + kl_loss * 2

lr = 0.0002
lr_decay = 0.5
decay_every = 50
beta1 = 0.5

d_vars = tl.layers.get_variables_with_name('discriminator', True, True)
encode_generator_vars = tl.layers.get_variables_with_name('encode_generator', True, True)

with tf.variable_scope('learning_rate'):
    lr_v = tf.Variable(lr, trainable=False)
d_optim = tf.train.AdamOptimizer(lr_v, beta1=beta1).minimize(d_loss, var_list=d_vars )
g_optim = tf.train.AdamOptimizer(lr_v, beta1=beta1).minimize(g_loss, var_list=encode_generator_vars)

sess = tf.InteractiveSession ()
tl.layers. initialize_global_variables (sess)

# load the latest checkpoints
net_rnn_name = os.path.join(save_dir, 'net_rnn.npz')
net_txtim2im_name = os.path.join(save_dir, 'net_txtim2im_g.npz')
net_d_name = os.path.join(save_dir, 'net_txtim2im_d.npz')

if load_and_assign_npz(sess=sess, name=net_rnn_name, model=net_rnn) is False:
    raise Exception("missing rnn model")

# seed for generation
sample_size = batch_size
idexs = get_random_int(min=0, max=n_captions_train-1, number=sample_size, seed=100)
```

```
sample_sentence = captions_ids_train[idexs]
sample_sentence = tl.prepro.pad_sequences(sample_sentence, padding='post')
sample_image = images_train[np.floor(np.asarray(idexs).astype('float')/n_captions_per_image).astype('int')]
sample_image = threading_data(sample_image, prepro_img, mode='test')

color_ids = [vocab.word_to_id(w) for w in ["red", "green", "yellow", "blue", "white", "pink", "purple", "orange",
    "black", "brown", "lavender"]]
sample_sentence_change = change_id(sample_sentence, color_ids, vocab.word_to_id("red"))

for i, sentence in enumerate(sample_sentence):
    print(i, [vocab.id_to_word(w) for w in sentence])
save_images(sample_image, [ni, ni], 'samples/txtim2im/_sample_images.png')

n_epoch = 600
n_batch_epoch = int(n_images_train / batch_size)

for epoch in range(0, n_epoch+1):
    start_time = time.time()

    if epoch !=0 and (epoch % decay_every == 0):
        new_lr_decay = lr_decay ** (epoch // decay_every)
        sess.run(tf.assign(lr_v, lr * new_lr_decay))
        log = " ** new learning rate: %f" % (lr * new_lr_decay)
        print(log)
    elif epoch == 0:
        log = " ** init lr: %f decay_every_epoch: %d, lr_decay: %f" % (lr, decay_every, lr_decay)
        print(log)

    for step in range(n_batch_epoch):
        step_time = time.time()
        # get matched text
        idexs = get_random_int(min=0, max=n_captions_train-1, number=batch_size)
        b_real_caption = captions_ids_train[idexs]
        b_real_caption = tl.prepro.pad_sequences(b_real_caption, padding='post')
        # get match image
        b_real_images = images_train[np.floor(np.asarray(idexs).astype('float')/n_captions_per_image).astype('int')]
        # get mismatching caption
        b_wrong_caption = np.append(b_real_caption[1:], [b_real_caption[0]], axis=0)
        # get related caption
```

```
        idexs = get_random_int(min=0, max=n_captions_train-1, number=batch_size)
        b_relate_caption = captions_ids_train[idexs]
        b_relate_caption = tl.prepro.pad_sequences(b_relate_caption, padding='post')
        b_real_images = threading_data(b_real_images, prepro_img, mode='train_strong')

        ## updates D
        errD, errD1, errD2, errD3, _ = sess.run([d_loss, d_loss1, d_loss2, d_loss3, d_optim], {
                        t_real_image : b_real_images,
                        t_wrong_caption : b_wrong_caption,
                        t_real_caption : b_real_caption,
                        t_relate_caption : b_relate_caption,
                        })
        ## updates G
        errG, _ = sess.run([g_loss, g_optim], {
                        t_real_image : b_real_images,
                        t_relate_caption : b_relate_caption,
                        })

        print("Epoch: [%2d/%2d] [%4d/%4d] time: %4.4fs, d_loss: %.8f (%.8f %.8f %.8f), g_loss: %.8f" \
                    % (epoch, n_epoch, step, n_batch_epoch, time.time() - step_time, errD, errD1, errD2, errD3,
                        errG))

    print(" ** Epoch %d took %fs" % (epoch, time.time()-start_time))
    img_gen, rnn_out = sess.run([net_fake_image2.outputs, net_rnn.outputs], {
                                    t_real_caption : sample_sentence,
                                    t_real_image : sample_image,
                                    })
    save_images(img_gen, [ni, ni], 'samples/txtim2im/train_{:02d}.png'.format(epoch))

    img_gen, rnn_out = sess.run([net_fake_image2.outputs, net_rnn.outputs], {
                                    t_real_caption : sample_sentence_change,
                                    t_real_image : sample_image,
                                    })
    save_images(img_gen, [ni, ni], 'samples/txtim2im/train_{:02d}c.png'.format(epoch))

    # save model
    if (epoch != 0) and (epoch % 5) == 0:
        tl.files.save_npz(net_d.all_params, name=net_d_name, sess=sess)
        tl.files.save_npz(net_fake_image.all_params, name=net_txtim2im_name, sess=sess)
```

```
    if (epoch != 0) and (epoch % 100) == 0:
        tl.files.save_npz(net_d.all_params, name=net_d_name+str(epoch), sess=sess)
        tl.files.save_npz(net_fake_image.all_params, name=net_txtim2im_name+str(epoch), sess=sess)
```

For 256 × 256 images, the loss functions are defined as follow:

```
encode_generator = model.cyclegan_generator_resnet
discriminator_txt2img = model.discriminator_txt2img_256

net_g, _ = encode_generator(
                t_real_image,
                net_rnn_relevant.outputs, is_train=True, reuse=False)

net_d, disc_fake_image_logits = discriminator_txt2img(
                net_g.outputs,
                net_rnn_relevant.outputs, is_train=True, reuse=False)

net_g_cycle, _= encode_generator(
                net_g.outputs,
                net_rnn_match.outputs, is_train=True, reuse=True)

_, disc_match_logits = discriminator_txt2img(
                t_real_image,
                net_rnn_match.outputs, is_train=True, reuse=True)

_, disc_mismatch_logits = discriminator_txt2img(
                t_real_image,
                net_rnn_mismatch.outputs, is_train=True, reuse=True)

## loss
d_loss1 = tl.cost.mean_squared_error(disc_match_logits, tf.ones_like(disc_match_logits), is_mean=True)
d_loss2 = tl.cost.mean_squared_error(disc_mismatch_logits, tf.zeros_like(disc_mismatch_logits), is_mean=True)
d_loss3 = tl.cost.mean_squared_error(disc_fake_image_logits, tf.zeros_like(disc_fake_image_logits), is_mean=True)
d_loss = d_loss1 + d_loss2 * 0.5 + d_loss3 * 0.5

cycle_loss = tl.cost.absolute_difference_error(net_g_cycle.outputs, t_real_image, is_mean=True)

g_loss1 = tl.cost.mean_squared_error(disc_fake_image_logits, tf.ones_like(disc_fake_image_logits), is_mean=True)
```

```
g_cycle_loss = cycle_loss * 0.1
g_loss = g_loss1 + g_cycle_loss
```

A.4 Supplementary Information for Chapter 6

A.4.1 Model performance

CIFAR-10

The following code is the implementation of the CIFAR-10 experiment.

```
import tensorflow as tf
import tensorlayer as tl
from tensorlayer.layers import *
import numpy as np
import time
from PIL import Image
import os
import io

model_file_name = "model_cifar10_tfrecord.ckpt"

## Download data, and convert it to TFRecord format
X_train, y_train, X_test, y_test = tl.files.load_cifar10_dataset(
                                    shape=(-1, 32, 32, 3), plotable=False)

def data_to_tfrecord(images, labels, filename):
    """ Save data into TFRecord format """
    if os.path.isfile(filename):
        print("%s exists" % filename)
        return
    print("Converting data into %s ..." % filename)
    cwd = os.getcwd()
    writer = tf.python_io.TFRecordWriter(filename)
    for index, img in enumerate(images):
        img_raw = img.tobytes()
        label = int(labels[index])
```

```
        example = tf.train.Example(features=tf.train.Features(feature={
            "label": tf.train.Feature(int64_list=tf.train.Int64List(value=[label])),
            'img_raw': tf.train.Feature(bytes_list=tf.train.BytesList(value=[img_raw])),
        }))
        writer.write(example.SerializeToString())
    writer.close()

def read_and_decode(filename, is_train=None):
    """ Return tensor to read from TFRecord """
    filename_queue = tf.train.string_input_producer([filename])
    reader = tf.TFRecordReader()
    _, serialized_example = reader.read(filename_queue)
    features = tf.parse_single_example(serialized_example,
                                       features={
                                           'label': tf.FixedLenFeature([], tf.int64),
                                           'img_raw' : tf.FixedLenFeature([], tf.string),
                                       })
    # You can do more image distortion here for training data
    img = tf.decode_raw(features['img_raw'], tf.float32)
    img = tf.reshape(img, [32, 32, 3])
    # img = tf.cast(img, tf.float32) #* (1. / 255) - 0.5
    if is_train == True:
        # 1. Randomly crop a [height, width] section of the image.
        img = tf.random_crop(img, [24, 24, 3])
        # 2. Randomly flip the image horizontally.
        img = tf.image.random_flip_left_right(img)
        # 3. Randomly change brightness.
        img = tf.image.random_brightness(img, max_delta=63)
        # 4. Randomly change contrast.
        img = tf.image.random_contrast(img, lower=0.2, upper=1.8)
        # 5. Subtract off the mean and divide by the variance of the pixels.
        try: # TF 0.12+
            img = tf.image.per_image_standardization(img)
        except: # earlier TF versions
            img = tf.image.per_image_whitening(img)

    elif is_train == False:
        # 1. Crop the central [height, width] of the image.
        img = tf.image.resize_image_with_crop_or_pad(img, 24, 24)
```

```
        # 2. Subtract off the mean and divide by the variance of the pixels.
        try: # TF 0.12+
            img = tf.image.per_image_standardization(img)
        except: # earlier TF versions
            img = tf.image.per_image_whitening(img)
    elif is_train == None:
        img = img

    label = tf.cast(features['label'], tf.int32)
    return img, label

## Save data into TFRecord files
data_to_tfrecord(images=X_train, labels=y_train, filename="train.cifar10")
data_to_tfrecord(images=X_test, labels=y_test, filename="test.cifar10")

batch_size = 128

sess = tf.Session(config=tf.ConfigProto(allow_soft_placement=True))
# prepare data in cpu
x_train_, y_train_ = read_and_decode("train.cifar10", True)
x_test_, y_test_ = read_and_decode("test.cifar10", False)

x_train_batch, y_train_batch = tf.train.shuffle_batch([x_train_, y_train_],
    batch_size=batch_size, capacity=2000, min_after_dequeue=1000, num_threads=32)
x_test_batch, y_test_batch = tf.train.batch([x_test_, y_test_],
    batch_size=batch_size, capacity=50000, num_threads=32)

def model(x_crop, y_, reuse):
    W_init = tf.truncated_normal_initializer(stddev=5e-2)
    W_init2 = tf.truncated_normal_initializer(stddev=0.04)
    b_init2 = tf.constant_initializer(value=0.1)
    with tf.variable_scope("model", reuse=reuse):
        tl.layers.set_name_reuse(reuse)
        net = InputLayer(x_crop, name='input')
        net = Conv2d(net, 64, (5, 5), (1, 1), padding='SAME',
                    W_init=W_init, name='cnn1')
        net = MaxPool2d(net, (3, 3), (2, 2), padding='SAME',name='pool1')
        net = LocalResponseNormLayer(net, depth_radius=4, bias=1.0,
                    alpha=0.001 / 9.0, beta=0.75, name='norm1')
```

```
        net = Conv2d(net, 64, (5, 5), (1, 1), padding='SAME',
                W_init=W_init, name='cnn2')
        net = LocalResponseNormLayer(net, depth_radius=4, bias=1.0,
                alpha=0.001 / 9.0, beta=0.75, name='norm2')
        net = MaxPool2d(net, (3, 3), (2, 2), padding='SAME',name='pool2')

        net = FlattenLayer(net, name='flatten')
        net = DenseLayer(net, n_units=384, act=tf.nn.relu,
                W_init=W_init2, b_init=b_init2, name='relu1')
        net = DenseLayer(net, n_units=192, act=tf.nn.relu,
                W_init=W_init2, b_init=b_init2, name='relu2')
        net = DenseLayer(net, n_units=10, act=tf.identity,
                W_init=tf.truncated_normal_initializer(stddev=1/192.0),
                name='output')
        y = net.outputs

        ce = tl.cost.cross_entropy(y, y_, name='cost')
        # L2 for the MLP, without this, the accuracy will be reduced by 15%.
        L2 = tf.contrib.layers.l2_regularizer(0.004)(net.all_params[4]) + \
             tf.contrib.layers.l2_regularizer(0.004)(net.all_params[6])
        cost = ce + L2

        correct_prediction = tf.equal(tf.cast(tf.argmax(y, 1), tf.int32), y_)
        acc = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

        return net, cost, acc

network, cost, acc, = model(x_train_batch, y_train_batch, False)
_, cost_test, acc_test = model(x_test_batch, y_test_batch, True)

## train
n_epoch = 50000
learning_rate = 0.0001
print_freq = 1
n_step_epoch = int(len(y_train)/batch_size)
n_step = n_epoch * n_step_epoch

with tf.device('/gpu:0'):
```

```
train_op = tf.train.AdamOptimizer(learning_rate, beta1=0.9, beta2=0.999,
    epsilon=1e-08, use_locking=False).minimize(cost)

tl.layers.initialize_global_variables(sess)
network.print_params(False)
network.print_layers()

print('    learning_rate: %f' % learning_rate)
print('    batch_size: %d' % batch_size)
print('    n_epoch: %d, step in an epoch: %d, total n_step: %d' % (n_epoch, n_step_epoch, n_step))

coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(sess=sess, coord=coord)
step = 0
for epoch in range(n_epoch):
    start_time = time.time()
    train_loss, train_acc, n_batch = 0, 0, 0
    for s in range(n_step_epoch):
        err, ac, _ = sess.run([cost, acc, train_op])
        step += 1; train_loss += err; train_acc += ac; n_batch += 1

    if epoch + 1 == 1 or (epoch + 1) % print_freq == 0:
        print("Epoch %d : Step %d-%d of %d took %fs" % (epoch, step, step + n_step_epoch, n_step,
            time.time() - start_time))
        print("    train loss: %f" % (train_loss/ n_batch))
        print("    train acc: %f" % (train_acc/ n_batch))

        test_loss, test_acc, n_batch = 0, 0, 0
        for _ in range(int(len(y_test)/batch_size)):
            err, ac = sess.run([cost_test, acc_test])
            test_loss += err; test_acc += ac; n_batch += 1
        print("    test loss: %f" % (test_loss/ n_batch))
        print("    test acc: %f" % (test_acc/ n_batch))

    if (epoch + 1) % (print_freq * 50) == 0:
        print("Save model " + "!"*10)
        saver = tf.train.Saver()
        save_path = saver.save(sess, model_file_name)
```

```
    coord.request_stop()
    coord.join(threads)
    sess.close()
```

PTB LSTM

The following code is the implementation of the PTB LSTM experiment.

```
import sys
import time
import numpy as np
import tensorflow as tf
import tensorlayer as tl

def main(_):
    # hyper-parameter setting
    init_scale = 0.1
    learning_rate = 1.0
    max_grad_norm = 5
    num_steps = 20
    hidden_size = 200
    max_epoch = 4
    max_max_epoch = 13
    keep_prob = 1.0
    lr_decay = 0.5
    batch_size = 20
    vocab_size = 10000

    # load PTB dataset
    train_data, valid_data, test_data, vocab_size = tl.files.load_ptb_dataset()
    sess = tf.InteractiveSession()

    # define models
    input_data = tf.placeholder(tf.int32, [batch_size, num_steps])
    targets = tf.placeholder(tf.int32, [batch_size, num_steps])
    input_data_test = tf.placeholder(tf.int32, [1, 1])
    targets_test = tf.placeholder(tf.int32, [1, 1])
```

```
def inference(x, is_training, num_steps, reuse=None):
    init = tf.random_uniform_initializer(-init_scale, init_scale)
    with tf.variable_scope("model", reuse=reuse):
        net = tl.layers.EmbeddingInputlayer(x, vocab_size, hidden_size, init, name='embedding')
        net = tl.layers.DropoutLayer(net, keep=keep_prob, is_fix=True, is_train=is_training, name='drop1')
        net = tl.layers.RNNLayer(
            net,
            cell_fn=tf.contrib.rnn.BasicLSTMCell,
            cell_init_args={'forget_bias': 0.0},
            n_hidden=hidden_size,
            initializer=init,
            n_steps=num_steps,
            return_last=False,
            name='basic_lstm_layer1'
        )
        lstm1 = net
        net = tl.layers.DropoutLayer(net, keep=keep_prob, is_fix=True, is_train=is_training, name='drop2')
        net = tl.layers.RNNLayer(
            net,
            cell_fn=tf.contrib.rnn.BasicLSTMCell,
            cell_init_args={'forget_bias': 0.0},
            n_hidden=hidden_size,
            initializer=init,
            n_steps=num_steps,
            return_last=False,
            return_seq_2d=True,
            name='basic_lstm_layer2'
        )
        lstm2 = net
        net = tl.layers.DropoutLayer(net, keep=keep_prob, is_fix=True, is_train=is_training, name='drop3')
        net = tl.layers.DenseLayer(net, vocab_size, W_init=init, b_init=init, act=None, name='output')
    return net, lstm1, lstm2

# inference for training
net, lstm1, lstm2 = inference(input_data, is_training=True, num_steps=num_steps, reuse=None)
# inference for validating
net_val, lstm1_val, lstm2_val = inference(input_data, is_training=False, num_steps=num_steps, reuse=True)
# inference for testing (evaluation)
net_test, lstm1_test, lstm2_test = inference(input_data_test, is_training=False, num_steps=1, reuse=True)
```

```
sess.run(tf.global_variables_initializer())

def loss_fn(outputs, targets):
    loss = tf.contrib.legacy_seq2seq.sequence_loss_by_example(
        [outputs], [tf.reshape(targets, [-1])], [tf.ones_like(tf.reshape(targets, [-1]), dtype=tf.float32)]
    )
    cost = tf.reduce_sum(loss) / batch_size
    return cost

# cost for training
cost = loss_fn(net.outputs, targets)
# cost for validating
cost_val = loss_fn(net_val.outputs, targets)
# cost for testing (evaluation)
cost_test = loss_fn(net_test.outputs, targets_test)

# truncated backpropagation for training
with tf.variable_scope('learning_rate'):
    lr = tf.Variable(0.0, trainable=False)
tvars = tf.trainable_variables()
grads, _ = tf.clip_by_global_norm(tf.gradients(cost, tvars), max_grad_norm)
optimizer = tf.train.GradientDescentOptimizer(lr)
train_op = optimizer.apply_gradients(zip(grads, tvars))

sess.run(tf.global_variables_initializer())

net.print_params()
net.print_layers()
tl.layers.print_all_variables()

print("Start learning a language model by using PTB dataset")
for i in range(max_max_epoch):
    # decreases the initial learning rate after several epoachs
    new_lr_decay = lr_decay**max(i - max_epoch, 0.0)
    sess.run(tf.assign(lr, learning_rate * new_lr_decay))

    # training
    print("Epoch: %d/%d Learning rate: %.3f" % (i + 1, max_max_epoch, sess.run(lr)))
```

```
epoch_size = ((len(train_data) // batch_size) - 1) // num_steps
start_time = time.time()
costs = 0.0
iters = 0
state1 = tl.layers.initialize_rnn_state(lstm1.initial_state)
state2 = tl.layers.initialize_rnn_state(lstm2.initial_state)
for step, (x, y) in enumerate(tl.iterate.ptb_iterator(train_data, batch_size, num_steps)):
    feed_dict = {
        input_data: x,
        targets: y,
        lstm1.initial_state: state1,
        lstm2.initial_state: state2,
    }

    feed_dict.update(net.all_drop)
    _cost, state1, state2, _ = sess.run(
        [cost, lstm1.final_state, lstm2.final_state, train_op], feed_dict=feed_dict
    )
    costs += _cost
    iters += num_steps

    if step % (epoch_size // 10) == 10:
        print(
            "%.3f perplexity: %.3f speed: %.0f wps" %
            (step * 1.0 / epoch_size, np.exp(costs / iters), iters * batch_size / (time.time() -
                start_time))
        )
train_perplexity = np.exp(costs / iters)
print("Epoch: %d/%d Train Perplexity: %.3f" % (i + 1, max_max_epoch, train_perplexity))

# validing
start_time = time.time()
costs = 0.0
iters = 0
state1 = tl.layers.initialize_rnn_state(lstm1_val.initial_state)
state2 = tl.layers.initialize_rnn_state(lstm2_val.initial_state)
for step, (x, y) in enumerate(tl.iterate.ptb_iterator(valid_data, batch_size, num_steps)):
    feed_dict = {
        input_data: x,
```

```
                targets: y,
                lstm1_val.initial_state: state1,
                lstm2_val.initial_state: state2,
            }
            _cost, state1, state2, _ = sess.run(
                [cost_val, lstm1_val.final_state, lstm2_val.final_state,
                 tf.no_op()], feed_dict=feed_dict
            )
            costs += _cost
            iters += num_steps
        valid_perplexity = np.exp(costs / iters)
        print("Epoch: %d/%d Valid Perplexity: %.3f" % (i + 1, max_max_epoch, valid_perplexity))

    print("Evaluation")
    start_time = time.time()
    costs = 0.0
    iters = 0
    state1 = tl.layers.initialize_rnn_state(lstm1_test.initial_state)
    state2 = tl.layers.initialize_rnn_state(lstm2_test.initial_state)
    for step, (x, y) in enumerate(tl.iterate.ptb_iterator(test_data, batch_size=1, num_steps=1)):
        feed_dict = {
            input_data_test: x,
            targets_test: y,
            lstm1_test.initial_state: state1,
            lstm2_test.initial_state: state2,
        }
        _cost, state1, state2 = sess.run(
            [cost_test, lstm1_test.final_state, lstm2_test.final_state], feed_dict=feed_dict
        )
        costs += _cost
        iters += 1
    test_perplexity = np.exp(costs / iters)
    print("Test Perplexity: %.3f took %.2fs" % (test_perplexity, time.time() - start_time))

if __name__ == "__main__":
    tf.app.run()
```

Word2Vec

The following code is the implementation of the Word2Vec experiment.

```
import sys
import time
import numpy as np
import tensorflow as tf
import tensorlayer as tl
from six.moves import xrange

def main_word2vec_basic():
    tf.logging.set_verbosity(tf.logging.DEBUG)
    tl.logging.set_verbosity(tl.logging.DEBUG)
    sess = tf.Session(config=tf.ConfigProto(allow_soft_placement=True))

    # Step 1: Download the data, read the context into a list of strings and set hyper parameters
    words = tl.files.load_matt_mahoney_text8_dataset()
    data_size = len(words)

    _UNK = "_UNK"
    vocabulary_size = 50000
    batch_size = 128
    embedding_size = 128
    skip_window = 1
    num_skips = 2
    num_sampled = 64
    learning_rate = 1.0
    n_epoch = 20
    model_file_name = "model_word2vec_50k_128"
    num_steps = int((data_size / batch_size) * n_epoch)

    print('%d Steps in a Epoch, total Epochs %d' % (int(data_size / batch_size), n_epoch))
    print('    learning_rate: %f' % learning_rate)
    print('    batch_size: %d' % batch_size)

    # Step 2: Build the dictionary and replace rare words with 'UNK' token.
    data, count, dictionary, reverse_dictionary = tl.nlp.build_words_dataset(words, vocabulary_size, True, _UNK)

    print('Most 5 common words (+UNK)',
```

```
        count[:5])  # [['UNK', 418391], (b'the', 1061396), (b'of', 593677), (b'and', 416629), (b'one', 411764)]
print('Sample data', data[:10], [reverse_dictionary[i] for i in data[:10]])

# Step 3: Function to generate a training batch for the Skip-Gram model.
batch, labels, data_index = tl.nlp.generate_skip_gram_batch(data=data, \
    batch_size=8, num_skips=4, skip_window=2, data_index=0)
for i in range(8):
    print(batch[i], reverse_dictionary[batch[i]], '->', labels[i, 0], reverse_dictionary[labels[i, 0]])

batch, labels, data_index = tl.nlp.generate_skip_gram_batch(data=data, \
    batch_size=8, num_skips=2, skip_window=1, data_index=0)
for i in range(8):
    print(batch[i], reverse_dictionary[batch[i]], '->', labels[i, 0], reverse_dictionary[labels[i, 0]])

# Step 4: Build a Skip-Gram model.
train_inputs = tf.placeholder(tf.int32, shape=[batch_size])
train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
valid_dataset = tf.constant(valid_examples, dtype=tf.int32)

# Look up embeddings for inputs.
emb_net = tl.layers.Word2vecEmbeddingInputlayer(
    inputs=train_inputs,
    train_labels=train_labels,
    vocabulary_size=vocabulary_size,
    embedding_size=embedding_size,
    num_sampled=num_sampled,
    nce_loss_args={},
    E_init=tf.random_uniform_initializer(minval=-1.0, maxval=1.0),
    E_init_args={},
    nce_W_init=tf.truncated_normal_initializer(stddev=float(1.0 / np.sqrt(embedding_size))),
    nce_W_init_args={},
    nce_b_init=tf.constant_initializer(value=0.0),
    nce_b_init_args={},
    name='word2vec_layer',
)

# Construct the optimizer
cost = emb_net.nce_cost
train_params = emb_net.all_params
```

```
train_op = tf.train.AdagradOptimizer(learning_rate, initial_accumulator_value=0.1,
                                     use_locking=False).minimize(cost, var_list=train_params)

# Compute the cosine similarity between minibatch examples and all embeddings.
normalized_embeddings = emb_net.normalized_embeddings
valid_embed = tf.nn.embedding_lookup(normalized_embeddings, valid_dataset)
similarity = tf.matmul(valid_embed, normalized_embeddings, transpose_b=True)

# Step 5: Start training.
sess.run(tf.global_variables_initializer())
print("Load existing model" + "!" * 10)
tl.files.load_and_assign_npz_dict(name=model_file_name + '.npz', sess=sess)

emb_net.print_params(False)
emb_net.print_layers()

tl.nlp.save_vocab(count, name='vocab_text8.txt')

average_loss = 0
step = 0
print_freq = 2000
while step < num_steps:
    start_time = time.time()
    batch_inputs, batch_labels, data_index = tl.nlp.generate_skip_gram_batch(data=data, \
        batch_size=batch_size, num_skips=num_skips, skip_window=skip_window, data_index=data_index)
    feed_dict = {train_inputs: batch_inputs, train_labels: batch_labels}
    # We perform one update step by evaluating the train_op
    _, loss_val = sess.run([train_op, cost], feed_dict=feed_dict)
    average_loss += loss_val

    if step % print_freq == 0:
        if step > 0:
            average_loss /= print_freq
        print("Average loss at step %d/%d. loss: %f took: %fs" % \
            (step, num_steps, average_loss, time.time() - start_time))
        average_loss = 0
    # Prints out nearby words given a list of words.
    if step % (print_freq * 5) == 0:
        sim = similarity.eval(session=sess)
```

```
            for i in xrange(valid_size):
                valid_word = reverse_dictionary[valid_examples[i]]
                top_k = 8
                nearest = (-sim[i, :]).argsort()[1:top_k + 1]
                log_str = "Nearest to %s:" % valid_word
                for k in xrange(top_k):
                    close_word = reverse_dictionary[nearest[k]]
                    log_str = "%s %s," % (log_str, close_word)
                print(log_str)

        if (step % (print_freq * 20) == 0) and (step != 0):
            print("Save model, data and dictionaries" + "!" * 10)
            tl.files.save_npz_dict(emb_net.all_params, name=model_file_name + '.npz', sess=sess)
            tl.files.save_any_to_npy(
                save_dict={
                    'data': data,
                    'count': count,
                    'dictionary': dictionary,
                    'reverse_dictionary': reverse_dictionary
                }, name=model_file_name + '.npy'
            )
        step += 1

# Step 6: Visualize the normalized embedding matrix by t-SNE.
final_embeddings = sess.run(normalized_embeddings) #.eval()
tl.visualize.tsne_embedding(final_embeddings, reverse_dictionary, plot_only=500, \
    second=5, saveable=False, name='word2vec_basic')

# Step 7: Evaluate by analogy questions. see tensorflow/models/embedding/word2vec_optimized.py
analogy_questions = tl.nlp.read_analogies_file(eval_file='questions-words.txt', word2id=dictionary)
# The eval feeds three vectors of word ids for a, b, c, each of
# which is of size N, where N is the number of analogies we want to
# evaluate in one batch.
analogy_a = tf.placeholder(dtype=tf.int32)
analogy_b = tf.placeholder(dtype=tf.int32)
analogy_c = tf.placeholder(dtype=tf.int32)
# Each row of a_emb, b_emb, c_emb is a word's embedding vector.
# They all have the shape [N, emb_dim]
a_emb = tf.gather(normalized_embeddings, analogy_a)
```

```
b_emb = tf.gather(normalized_embeddings, analogy_b)
c_emb = tf.gather(normalized_embeddings, analogy_c)
target = c_emb + (b_emb - a_emb)
# Compute cosine distance between each pair of target and vocabulary.
dist = tf.matmul(target, normalized_embeddings, transpose_b=True)
# For each question (row in dist), find the top 'n_answer' words.
n_answer = 4
_, pred_idx = tf.nn.top_k(dist, n_answer)

def predict(analogy):
    """Predict the top 4 answers for analogy questions."""
    idx, = sess.run([pred_idx], {analogy_a: analogy[:, 0], analogy_b: analogy[:, 1], analogy_c: analogy[:, 2]})
    return idx

# Evaluate analogy questions and reports accuracy.
correct = 0
total = analogy_questions.shape[0]
start = 0
while start < total:
    limit = start + 2500
    sub = analogy_questions[start:limit, :]
    idx = predict(sub)  # 4 answers for each question
    start = limit
    for question in xrange(sub.shape[0]):
        for j in xrange(n_answer):
            if idx[question, j] == sub[question, 3]:
                print(
                    j + 1, tl.nlp.word_ids_to_words([idx[question, j]], reverse_dictionary), ':',
                    tl.nlp.word_ids_to_words(sub[question, :], reverse_dictionary)
                )
                correct += 1
                break
            elif idx[question, j] in sub[question, :3]:
                # We need to skip words already in the question.
                continue
            else:
                # The correct label is not the precision@1
                break
print("Eval %4d/%d accuracy = %4.1f%%" % (correct, total, correct * 100.0 / total))
```

```
if __name__ == '__main__':
    main_word2vec_basic()
```

A.4.2 Hyper parameter selection

Task runner

The following code is the implementation the task runner for the hyper parameter selection experiment.

```
import time
import tensorlayer as tl

# connect to database
db = tl.db.TensorHub(ip='localhost', port=27017, dbname='temp', project_name='tutorial')

# monitor the database and pull tasks to run
while True:
    print("waiting task from distributor")
    db.run_top_task(task_name='mnist', sort=[("time", -1)])
    time.sleep(1)
```

Task dispatcher

The following code is the implementation of the task dispatcher that create 100 tasks for the hyper parameter selection experiment.

```
import time
import tensorlayer as tl
import tensorflow as tf

tl.logging.set_verbosity(tl.logging.DEBUG)

# connect to database
db = tl.db.TensorHub(ip='localhost', port=27017, dbname='temp', project_name='tutorial')

# delete existing tasks, models and datasets in this project
```

```
db.delete_tasks()
db.delete_model()
db.delete_datasets()

# save dataset into database, then allow other servers to use it
X_train, y_train, X_val, y_val, X_test, y_test = tl.files.load_mnist_dataset(shape=(-1, 784))
db.save_dataset((X_train, y_train, X_val, y_val, X_test, y_test), 'mnist', description='handwriting digit')

# push tasks into database, then allow other servers pull tasks to run
for drop1 in range(1, 11):
    for drop2 in range(1, 11):
        db.create_task(
            task_name='mnist', script='task_script.py', hyper_parameters=dict(drop1=drop1/10., drop2=drop2/10.),
            saved_result_keys=['test_accuracy'], description='100tasks-{}-{}'.format(drop1, drop2)
        )

# wait all tasks to be finished
while db.check_unfinished_task(task_name='mnist'):
    print("waiting runners to finish the tasks")
    time.sleep(1)

# get the best model
print("all tasks finished")
sess = tf.InteractiveSession()
net = db.find_top_model(sess=sess, model_name='mlp', sort=[("test_accuracy", -1)])
print("the best accuracy {} is from model {}".format(net._test_accuracy, net._name))
```

Task script

The following code is the implementation of the training pipeline (*i.e.,* the script field of task record) for training MLP classifier on MNIST.

```
import tensorflow as tf
import tensorlayer as tl

tf.logging.set_verbosity(tf.logging.DEBUG)
tl.logging.set_verbosity(tl.logging.DEBUG)
```

```
sess = tf.InteractiveSession()

# connect to database
db = tl.db.TensorHub(ip='localhost', port=27017, dbname='temp', project_name='tutorial')

# load dataset from database
X_train, y_train, X_val, y_val, X_test, y_test = db.find_top_dataset('mnist')

# define placeholder
x = tf.placeholder(tf.float32, shape=[None, 784], name='x')
y_ = tf.placeholder(tf.int64, shape=[None], name='y_')

# define the network
def mlp(x, is_train=True, reuse=False):
    with tf.variable_scope("MLP", reuse=reuse):
        net = tl.layers.InputLayer(x, name='input')
        net = tl.layers.DenseLayer(net, n_units=800, act=tf.nn.relu, name='relu1')
        net = tl.layers.DropoutLayer(net, keep=drop1, is_fix=True, is_train=is_train, name='drop1')
        net = tl.layers.DenseLayer(net, n_units=800, act=tf.nn.relu, name='relu2')
        net = tl.layers.DropoutLayer(net, keep=drop2, is_fix=True, is_train=is_train, name='drop2')
        net = tl.layers.DenseLayer(net, n_units=10, act=None, name='output')
    return net

# define inferences
net_train = mlp(x, is_train=True, reuse=False)
net_test = mlp(x, is_train=False, reuse=True)

# cost for training
y = net_train.outputs
cost = tl.cost.cross_entropy(y, y_, name='xentropy')
correct_prediction = tf.equal(tf.argmax(y, 1), y_)
acc = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# cost and accuracy for evalution
y2 = net_test.outputs
cost_test = tl.cost.cross_entropy(y2, y_, name='xentropy2')
correct_prediction = tf.equal(tf.argmax(y2, 1), y_)
acc_test = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

```
# define the optimizer
train_params = tl.layers.get_variables_with_name('MLP', True, False)
train_op = tf.train.AdamOptimizer(learning_rate=0.0001).minimize(cost, var_list=train_params)

# initialize all variables in the session
tl.layers.initialize_global_variables(sess)

# train the network
tl.utils.fit(
    sess, net_train, train_op, cost, X_train, y_train, x, y_, acc=acc, batch_size=500, n_epoch=100, print_freq=5,
    X_val=X_val, y_val=y_val, eval_train=False
)

# evaluation and save result that match the result_key
test_accuracy = tl.utils.test(sess, net_test, acc_test, X_test, y_test, x, y_, batch_size=None, cost=cost_test)
test_accuracy = float(test_accuracy)

# save model into database
db.save_model(net_train, model_name='mlp', name=str(n_units1) + '-' + str(n_units2), test_accuracy=test_accuracy)
```

A.4.3 Deep reinforcement learning

Model trainer

The following code is the implementation of the model trainer for the deep reinforcement learning experiment.

```
import tensorflow as tf
import tensorlayer as tl
import gym
import numpy as np
import time

# hyper parameters
image_size = 80
D = image_size * image_size
H = 200
batch_size = 10
```

```
learning_rate = 1e-4
gamma = 0.99
decay_rate = 0.99
np.set_printoptions(threshold=np.nan)

from tensorlayer.db import TensorHub

# This is to initialize the connection to your MondonDB server
db = TensorHub(ip='IP_ADDRESS', port=27017, db_name='DATABASE_NAME', user_name=None,
    password=None, project_name='drl')

states_batch_pl = tf.placeholder(tf.float32, shape=[None, D])
# policy network
net = tl.layers.InputLayer(states_batch_pl, name='input')
net = tl.layers.DenseLayer(net, n_units=H, act=tf.nn.relu, name='relu1')
net = tl.layers.DenseLayer(net, n_units=3, act=tf.identity, name='output')
probs = net.outputs
sampling_prob = tf.nn.softmax(probs)

actions_batch_pl = tf.placeholder(tf.int32, shape=[None])
discount_rewards_batch_pl = tf.placeholder(tf.float32, shape=[None])
loss = tl.rein.cross_entropy_reward_loss(probs, actions_batch_pl,
                                         discount_rewards_batch_pl)
train_op = tf.train.RMSPropOptimizer(learning_rate, decay_rate).minimize(loss)

with tf.Session() as sess:
    tl.layers.initialize_global_variables(sess)
    net.print_params()
    net.print_layers()

    start_time = time.time()
    game_number = 0
    n = 0
    total_n_examples = 0
    while True:
        is_found = False
        while is_found is False:
            data, f_id = db.find_one_params(args={'type': 'train_data'}, lz4_decomp=True)
            if (data is not False):
```

```
            epx, epy, epr = data
            db.del_params(args={'type': 'train_data', 'f_id': f_id})
            is_found = True
        else:
            time.sleep(0.5)
    disR = tl.rein.discount_episode_rewards(epr, gamma)
    disR -= np.mean(disR)
    disR /= np.std(disR)

    sess.run(train_op,{
            states_batch_pl: epx,
            actions_batch_pl: epy,
            discount_rewards_batch_pl: disR
        })
    n_examples = epx.shape[0]
    total_n_examples += n_examples
    print("[*] Update {}: n_examples: {} / total averaged speed: {} examples/second".format(n, n_examples,
                                round(total_n_examples/(time.time() - start_time), 2)))
    n += 1

    if n % 10 == 0:
        db.del_params(args={'type': 'network_parameters'})
        db.save_params(sess.run(net.all_params), args={'type': 'network_parameters'}, lz4_comp=True)
```

Data generator

The following code is the implementation of the data generator for the deep reinforcement learning experiment.

```
import tensorflow as tf
import tensorlayer as tl
import gym
import numpy as np
import time, os
import argparse
from bson.objectid import ObjectId

os.environ["CUDA_VISIBLE_DEVICES"]=""
```

```
from tensorlayer.db import TensorHub

db = TensorHub(ip='IP_ADDRESS', port=27017, db_name='DATABASE_NAME', user_name=None,
     password=None, project_name='drl')

def main(args):
    # hyper parameters
    image_size = 80
    D = image_size * image_size
    H = 200
    batch_size = 10
    gamma = 0.99
    np.set_printoptions(threshold=np.nan)

    def prepro(I):
        """ preprocess 210x160x3 uint8 frame into 6400 (80x80) 1D float vector """
        I = I[35:195]
        I = I[::2, ::2, 0]
        I[I == 144] = 0
        I[I == 109] = 0
        I[I != 0] = 1
        return I.astype(np.float).ravel()

    env = gym.make("Pong-v0")
    observation = env.reset()
    prev_x = None
    running_reward = None
    reward_sum = 0
    episode_number = 0

    xs, ys, rs = [], [], []
    # observation for training and inference
    states_batch_pl = tf.placeholder(tf.float32, shape=[None, D])
    # policy network
    net = tl.layers.InputLayer(states_batch_pl, name='input')
    net = tl.layers.DenseLayer(net, n_units=H, act=tf.nn.relu, name='relu1')
    net = tl.layers.DenseLayer(net, n_units=3, act=tf.identity, name='output')
    probs = net.outputs
```

```
sampling_prob = tf.nn.softmax(probs)

with tf.Session() as sess:
    tl.layers.initialize_global_variables(sess)
    net.print_params()
    net.print_layers()

    start_time = time.time()
    game_number = 0
    while True:
        task = db.get_task()

        cur_x = prepro(observation)
        x = cur_x - prev_x if prev_x is not None else np.zeros(D)
        x = x.reshape(1, D)
        prev_x = cur_x

        prob = sess.run(sampling_prob, feed_dict={states_batch_pl: x})
        # action. 1: STOP 2: UP 3: DOWN
        action = np.random.choice([1, 2, 3], p=prob.flatten())

        observation, reward, done, _ = env.step(action)
        reward_sum += reward
        xs.append(x)
        ys.append(action - 1)
        rs.append(reward)
        if done:
            episode_number += 1
            game_number = 0

            if episode_number % batch_size == 0:
                print('batch over ..... saving training data ..... ')
                epx = np.vstack(xs)
                epy = np.asarray(ys)
                epr = np.asarray(rs)
                disR = tl.rein.discount_episode_rewards(epr, gamma)
                disR -= np.mean(disR)
                disR /= np.std(disR)
```

```
                xs, ys, rs = [], [], []

                print("[*] Generated {} examples".format(epx.shape[0]))
                f_id = db.save_params([epx, epy, epr], args={'type': 'train_data'}, lz4_comp=True)

            running_reward = reward_sum if running_reward is None else running_reward * 0.99 + reward_sum *
                0.01
            print('resetting env. episode reward total was %f. running mean: %f' % (reward_sum,
                running_reward))
            reward_sum = 0
            observation = env.reset()
            prev_x = None

        if reward != 0:
            print(('episode %d: game %d took %.5fs, reward: %f' %
                (episode_number, game_number,
                    time.time() - start_time, reward)),
                ('' if reward == -1 else '   !!!!!!!! '))
            start_time = time.time()

            if (episode_number % 20 == 0) and (game_number == 0):
                try:
                    params, f = db.find_one_params(args={'type': 'network_parameters'}, lz4_decomp=True)
                    if (params is not False):
                        tl.files.assign_params(sess, params, net)
                        print("[*] Update Model")
                except:
                    continue

            game_number += 1

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--task_id", type=str, required=True, help="Task ID.")
    args = parser.parse_args()
    main(args)
```

Bibliography

[1] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: Lessons learned from the 2015 mscoco image captioning challenge," *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 2016.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the Neural Information Processing Systems (NIPS)*, 2012, pp. 1097–1105.

[4] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, and M. Bernstein, "Imagenet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[5] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[6] F. Milletari, N. Navab, and S.-A. Ahmadi, "V-net: Fully convolutional neural networks for volumetric medical image segmentation," in *Proceedings of the International Conference on 3D Vision (3DV)*. IEEE, 2016, pp. 565–571.

[7] J. Li, X. Liang, Y. Wei, T. Xu, J. Feng, and S. Yan, "Perceptual generative adversarial networks for small object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 1222–1230.

[8] C. Fan, Z. Zhai, Y. Ming, and L. Tian, "Two-stream siamese network with contrastive-center losses for rgb-d action recognition," *Journal of Electronic Imaging*, vol. 28, no. 2, p. 023004, 2019.

[9] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context." Springer, 2014.

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[11] P. Rajpurkar, J. Irvin, K. Zhu, B. Yang, H. Mehta, T. Duan, D. Ding, A. Bagul, C. Langlotz, K. Shpanskaya, M. P. Lungren, and A. Y. Ng, "Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning," *arXiv preprint arXiv:1711.05225*, 2017.

[12] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[13] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[14] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 2015, pp. 649–657.

[15] I. Saito, J. Suzuki, K. Nishida, K. Sadamitsu, S. Kobashikawa, R. Masumura, Y. Matsumoto, and J. Tomita, "Improving neural text normalization with data augmentation at character-and morphological levels," in *Proceedings of the International Joint Conference on Natural Language Processing*, vol. 2, 2017, pp. 257–262.

[16] A. Odena, C. Olah, and J. Shlens, "Conditional image synthesis with auxiliary classifier gans," in *Processing of the International Conference on Machine Learning (ICML)*.

[17] M. Mirza and S. Osindero, "Conditional Generative Adversarial Nets," *arXiv preprint arXiv:1411.1784*, 2014.

[18] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2014.

[19] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[20] T. Karras, T. Aila, S. Laine, and J. Lehtinen, "Progressive growing of gans for improved quality, stability, and variation," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

[21] H. Miao, A. Li, L. S. Davis, and A. Deshpande, "Modelhub: Deep learning lifecycle management," in *International Conference on Data Engineering (ICDE)*, 2017.

[22] A. Krogh, J. Vedelsby *et al.*, "Neural network ensembles, cross validation, and active learning," in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 1995.

[23] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *Proceedings of the Neural Information Processing Systems (NIPS) Workshop*, 2017.

[24] S. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, and H. Lee, "Generative Adversarial Text to Image Synthesis," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2016.

[25] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks," in *Proceedings of International Conference on Computer Vision (ICCV)*, 2017.

[26] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Nets," in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 2014.

[27] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1440–1448.

[28] J. Johnson, A. Alahi, and L. Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2016.

[29] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi, "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[30] D. Pathak, P. Krahenbuhl, J. Donahue, T. Darrell, and A. A. Efros, "Context encoders: Feature learning by inpainting," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016, pp. 2536–2544.

[31] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological Review*, vol. 65, no. 6, p. 386, 1958.

[32] D. W. Ruck, S. K. Rogers, M. Kabrisky, M. E. Oxley, and B. W. Suter, "The multilayer perceptron as an approximation to a bayes optimal discriminant function," *IEEE Transactions on Neural Networks*, vol. 1, no. 4, pp. 296–298, 1990.

[33] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2011, pp. 315–323.

[34] Z. Cao, Z. Simon, S.-E. Wei, and S.-E. Sheikh, "Realtime multi-person 2d pose estimation using part affinity fields," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[35] H. Noh, S. Hong, and B. Han, "Learning deconvolution network for semantic segmentation," in *Proceedings of the International Conference on Computer Vision (ICCV)*, 2015, pp. 1520–1528.

[36] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," in *Proceedings of the International Conference on Machine Learning (ICML) Workshop*, 2015.

[37] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[38] L. Bottou and O. Bousquet, "The Tradeoffs of Large Scale Learning." in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, vol. 20, 2007, pp. 161–168.

[39] H. Dong, G. Yike, and Y. Guang, *Deep Learning using TensorLayer*, 2018.

[40] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[41] H. Dong, G. Yang, F. Liu, Y. Mo, and Y. Guo, "Automatic brain tumor detection and segmentation using u-net based fully convolutional networks," in *Proceedings of the Annual Conference on Medical Image Understanding and Analysis (MIUA)*. Springer, 2017, pp. 506–517.

[42] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 3431–3440.

[43] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv preprint arXiv:1603.07285*, 2016.

[44] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *Proceedings of the International Conference on Computer Vision (ICCV)*. IEEE, 2017, pp. 2980–2988.

[45] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[46] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *Computing Research Repository (CoRR)*, 2017.

[47] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[48] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[49] A. Mahendran and A. Vedaldi, "Visualizing deep convolutional neural networks using natural pre-images," *International Journal of Computer Vision (IJCV)*, vol. 120, no. 3, pp. 233–255, 2016.

[50] A. Odena and O. Vincent, Dumoulinand Chris, "Rethinking the inception architecture for computer vision," *Distill*, 2016.

[51] Y. Zhang, R. Jin, and Z.-H. Zhou, "Understanding bag-of-words model: a statistical framework," *International Journal of Machine Learning and Cybernetics*, vol. 1, no. 1-4, pp. 43–52, 2010.

[52] R. Kiros, R. Salakhutdinov, and R. S. Zemel, "Unifying visual-semantic embeddings with multimodal neural language models," *Transactions of the Association for Computational Linguistics (TACL)*, 2015.

[53] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *Computing Research Repository (CoRR)*, 2013.

[54] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the Empirical Methods in Natural Language Processing (EMNLP) Conference*, 2014, pp. 1532–1543.

[55] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[56] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 2014, pp. 3104–3112.

[57] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," in *Proceedings of the Empirical Methods in Natural Language Processing (EMNLP) Conference*, 2014.

[58] S. Hochreiter, S. Hochreiter, J. Schmidhuber, and J. Schmidhuber, "Long Short-Term Memory." *Neural Computation*, vol. 9, no. 8, pp. 1735–80, 1997.

[59] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)*, vol. 28, no. 10, pp. 2222–2232, 2017.

[60] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, p. 436, 2015.

[61] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2014.

[62] T. Tieleman and G. Hinton, "Divide the gradient by a running average of its recent magnitude. coursera: Neural networks for machine learning," Technical Report, Tech. Rep., 2017.

[63] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research (JMLR)*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[64] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[65] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research (JMLR)*, vol. 15, no. 1, pp. 1929–1958, 2014.

[66] K. Hara, D. Saitoh, and H. Shouno, "Analysis of dropout learning regarded as ensemble learning," in *Proceedings of the International Conference on Artificial Neural Networks (ICANN)*. Springer, 2016, pp. 72–79.

[67] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2016, pp. 1050–1059.

[68] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2015.

[69] T. Ko, V. Peddinti, D. Povey, and S. Khudanpur, "Audio augmentation for speech recognition," in *Annual Conference of the International Speech Communication Association*, 2015.

[70] J. West, D. Ventura, and S. Warnick, "Spring research presentation: A theoretical foundation for inductive transfer," *Brigham Young University, College of Physical and Mathematical Sciences*, vol. 1, 2007.

[71] L. Y. Pratt, "Discriminability-based transfer between neural networks," in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 1993, pp. 204–211.

[72] M. Andriluka, L. Pishchulin, P. Gehler, and B. Schiele, "2d human pose estimation: New benchmark and state of the art analysis," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[73] X. Rong, "Word2vec parameter learning explained," *arXiv preprint arXiv:1411.2738*, 2014.

[74] A. Supratak, S. Schneider, H. Dong, L. Li, and Y. Guo, "Towards desynchronization detection in biosignals," in *Proceedings of the Neural Information Processing Systems (NIPS) Workspace*, 2017.

[75] A. Dosovitskiy, J. Tobias Springenberg, and T. Brox, "Learning to Generate Chairs with Convolutional Neural Networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[76] J. Yang, S. Reed, M.-H. Yang, and H. Lee, "Weakly-supervised Disentangling with Recurrent Transformations for 3D View Synthesis," in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 2015.

[77] S. Reed, Y. Zhang, Y. Zhang, and H. Lee, "Deep Visual Analogy-Making," in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 2015.

[78] D. J. Rezende, S. Mohamed, and D. Wierstra, "Stochastic Backpropagation and Approximate Inference in Deep Generative Models," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2014.

[79] K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra, "Stochastic Backpropagation and Approximate Inference in Deep Generative Models," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2015.

[80] A. V. d. Oord, N. Kalchbrenner, O. Vinyals, L. Espeholt, A. Graves, and K. Kavukcuoglu, "Conditional Image Generation with PixelCNN Decoders," in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 2016.

[81] A. V. d. Oord, N. Kalchbrenner, and K. Kavukcuoglu, "Pixel Recurrent Neural Networks," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2016.

[82] E. Denton, S. Chintala, A. Szlam, and R. Fergus, "Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks," in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 2015.

[83] H. Zhao, S. Zhang, G. Wu, G. J. Gordon *et al.*, "Multiple source domain adaptation with adversarial learning," in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 2018.

[84] A. Borji, "Pros and cons of gan evaluation measures," *Computer Vision and Image Understanding*, vol. 179, pp. 41–65, 2019.

[85] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "Gans trained by a two time-scale update rule converge to a local nash equilibrium," in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 2017.

[86] H. Zhang, T. Xu, H. Li, S. Zhang, X. Huang, X. Wang, and D. Metaxas, "StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks," in *Proceedings of the International Conference on Computer Vision (ICCV)*, 2017.

[87] J.-Y. Zhu, R. Zhang, D. Pathak, T. Darrell, A. A. Efros, O. Wang, and E. Shechtman, "Toward Multimodal Image-to-Image Translation," in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 2017.

[88] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved Techniques for Training GANs," in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 2016.

[89] X. Mao, Q. Li, H. Xie, R. Y. Lau, Z. Wang, and S. P. Smolley, "Least squares generative adversarial networks," in *Proceedings of the International Conference on Computer Vision (ICCV)*. IEEE, 2017, pp. 2813–2821.

[90] H. Dong, P. Neekhara, C. Wu, and Y. Guo, "Unsupervised image-to-image translation with generative adversarial networks," *arXiv preprint arXiv:1701.02676*, 2017.

[91] X. Chen, Y. Duan, R. Houthooft, J. Schulman, I. Sutskever, and P. Abbeel, "InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets," in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 2016.

[92] X. Yan, J. Yang, K. Sohn, and H. Lee, "Attribute2Image: Conditional Image Generation from Visual Attributes," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2016.

[93] Y. Taigman, A. Polyak, and L. Wolf, "Unsupervised Cross-Domain Image Generation," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

[94] D. Yoo, N. Kim, S. Park, A. S. Paek, and I. S. Kweon, "Pixel-Level Domain Transfer," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2016.

[95] M.-Y. Liu, T. Breuel, and J. Kautz, "Unsupervised image-to-image translation networks," in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 2017, pp. 700–708.

[96] J.-Y. Zhu, P. Krähenbühl, E. Shechtman, and A. A. Efros, "Generative Visual Manipulation on the Natural Image Manifold," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2016.

[97] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, "Neural Photo Editing with Introspective Adversarial Networks," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

[98] X. Wang and A. Gupta, "Generative Image Modeling Using Style and Structure Adversarial Networks," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2016.

[99] C. Li and M. Wand, "Precomputed Real-Time Texture Synthesis with Markovian Generative Adversarial Networks," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2016.

[100] C. Donahue, B. Li, and R. Prabhavalkar, "Exploring speech enhancement with generative adversarial networks for robust speech recognition," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2018, pp. 5024–5028.

[101] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein generative adversarial networks," in *Processing of the International Conference on Machine Learning (ICML)*, 2017, pp. 214–223.

[102] C. Wah, S. Branson, P. Welinder, P. Perona, and S. Belongie, "The Caltech-UCSD Birds-200-2011 Dataset," California Institute of Technology, Tech. Rep. CNS-TR-2011-001, 2011.

[103] M.-E. Nilsback and A. Zisserman, "Automated Flower Classification over a Large Number of Classes," in *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing (ICCVGIP)*, 2008.

[104] A. Karpathy and L. Fei-Fei, "Deep visual-semantic alignments for generating image descriptions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[105] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio, "Show, attend and tell: Neural image caption generation with visual attention," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2015, pp. 2048–2057.

[106] G. Kulkarni, V. Premraj, S. Dhar, S. Li, Y. Choi, A. C. Berg, and T. L. Berg, "Baby talk: Understanding and generating image descriptions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Citeseer, 2011.

[107] A. Farhadi, M. Hejrati, M. A. Sadeghi, P. Young, C. Rashtchian, J. Hockenmaier, and D. Forsyth, "Every picture tells a story: Generating sentences from images," in *Proceedings of the European Conference on Computer Vision (ECCV)*. Springer, 2010, pp. 15–29.

[108] K. Gregor, I. Danihelka, A. Graves, and D. Wierstra, "DRAW: A Recurrent Neural Network For Image Generation," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2015.

[109] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," in *Proceedings of the Neural Information Processing Systems (NIPS) Workshop on Deep Learning and Unsupervised Feature Learning*, vol. 2011, no. 2, 2011, p. 5.

[110] E. Mansimov, E. Parisotto, J. L. Ba, and R. Salakhutdinov, "Generating Images from Captions with Attention," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[111] P. Baldi, "Autoencoders, Unsupervised Learning, and Deep Architectures," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2012, pp. 37–50.

[112] H. Dong, A. Supratak, L. Mai, F. Liu, A. Oehmichen, S. Yu, and Y. Guo, "TensorLayer: a versatile library for efficient deep learning development," in *Proceedings of the ACM Multimedia (MM)*, 2017. [Online]. Available: http://tensorlayer.org

[113] M. Abadi, P. Barham, J. Chen, A. Davis, J. Dean, M. Devin, S. Geoffrey, G. Irving, M. Devin, M. Kudlur, J. Manjunath, R. Monga, S. Moore, D. G. Murray, B. Derek, P. Tucker, V. Vasude-

van, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Usenix OSDI*, 2016.

[114] R. A. Fisher and F. Yates, *Statistical tables for biological, agricultural and medical research.* Oliver and Boyd Ltd, London, 1943.

[115] W. Chen, H. Wang, Y. Li, H. Su, Z. Wang, C. Tu, D. Lischinski, D. Cohen-Or, and B. Chen, "Synthesizing training images for boosting human 3d pose estimation," in *Proceedings of the International Conference on 3D Vision (3DV).* IEEE, 2016, pp. 479–488.

[116] X. Peng, B. Usman, N. Kaushik, D. Wang, J. Hoffman, and K. Saenko, "Visda: A synthetic-to-real benchmark for visual domain adaptation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2018, pp. 2021–2026.

[117] M. Frid-Adar, E. Klang, M. Amitai, J. Goldberger, and H. Greenspan, "Synthetic data augmentation using gan for improved liver lesion classification," in *International Symposium on Biomedical Imaging (ISBI).* IEEE, 2018, pp. 289–293.

[118] M. Frid-Adar, I. Diamant, E. Klang, M. Amitai, J. Goldberger, and H. Greenspan, "Gan-based synthetic medical image augmentation for increased cnn performance in liver lesion classification," *Neurocomputing*, vol. 321, pp. 321–331, 2018.

[119] E. Collier, K. Duffy, S. Ganguly, G. Madanguit, S. Kalia, G. Shreekant, R. Nemani, A. Michaelis, S. Li, A. Ganguly *et al.*, "Progressively growing generative adversarial networks for high resolution semantic segmentation of satellite images," in *Proceedings of the International Conference on Data Mining Workshops (ICDMW).* IEEE, 2018, pp. 763–769.

[120] I. Korshunova, W. Shi, J. Dambre, and L. Theis, "Fast face-swap using convolutional neural networks," in *Proceedings of the International Conference on Computer Vision (ICCV)*, 2017.

[121] Y. Shih, S. Paris, F. Durand, and W. T. Freeman, "Data-driven hallucination of different times of day from a single outdoor photo," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 6, p. 200, 2013.

[122] P.-Y. Laffont, Z. Ren, X. Tao, C. Qian, and J. Hays, "Transient attributes for high-level understanding and editing of outdoor scenes," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, p. 149, 2014.

[123] T. Chen, M.-M. Cheng, P. Tan, A. Shamir, and S.-M. Hu, "Sketch2photo: internet image montage," *ACM Transactions on Graphics (TOG)*, vol. 28, no. 5, p. 124, 2009.

[124] Z. Liu, P. Luo, X. Wang, and X. Tang, "Deep learning face attributes in the wild," in *Proceedings of the International Conference on Computer Vision (ICCV)*, 2015.

[125] W. Ting-Chun, L. Ming-Yu, Z. Jun-Yan, T. Andrew, K. Jan, and C. Bryan, "High-resolution image synthesis and semantic manipulation with conditional GANs," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[126] Q. Chen and K. Vladlen, "Photographic image synthesis with cascaded refinement networks," in *Proceedings of International Conference on Computer Vision (ICCV)*, 2017.

[127] R. A. Yeh, C. Chen, T. Yian Lim, A. G. Schwing, M. Hasegawa-Johnson, and M. N. Do, "Semantic image inpainting with deep generative models," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 5485–5493.

[128] F. Liu, C. Shen, and G. Lin, "Deep convolutional neural fields for depth estimation from a single image," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 5162–5170.

[129] C. Barnes, E. Shechtman, A. Finkelstein, and D. B. Goldman, "Patchmatch: A randomized correspondence algorithm for structural image editing," in *ACM Transactions on Graphics (ToG)*, vol. 28, no. 3. ACM, 2009, p. 24.

[130] Z. Wang, A. C. Bovik, H. R. Sheikh, E. P. Simoncelli *et al.*, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing (TIP)*, vol. 13, no. 4, pp. 600–612, 2004.

[131] L. Metz, B. Poole, D. Pfau, and J. Sohl-Dickstein, "Unrolled generative adversarial networks," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

[132] Z. Yi, H. Zhang, P. Tan, and M. Gong, "Dualgan: Unsupervised dual learning for image-to-image translation," in *Proceedings of the International Conference on Computer Vision (ICCV)*, 2017.

[133] T. Kim, M. Cha, H. Kim, J. Lee, and J. Kim, "Learning to discover cross-domain relations with generative adversarial networks," in *Processing of the International Conference on Machine Learning (ICML)*, 2017.

[134] S. Gurumurthy, R. K. Sarvadevabhatla, and V. B. Radhakrishnan, "DeliGAN: Generative adversarial networks for diverse and limited data," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 1, 2017.

[135] S. Reed, Z. Akata, S. Mohan, S. Tenka, B. Schiele, and H. Lee, "Learning What and Where to Draw," in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 2016.

[136] S. Reed, Z. Akata, H. Lee, and B. Schiele, "Learning Deep Representations of Fine-Grained Visual Descriptions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[137] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: A cpu and gpu math compiler in python," in *Proc. 9th Python in Science Conf*, vol. 1, 2010, pp. 3–10.

[138] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu, "Spatial transformer networks," in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 2015, pp. 2017–2025.

[139] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," in *CoRR*, 2016.

[140] J. Andreas, M. Rohrbach, T. Darrell, and D. Klein, "Neural module networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[141] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research (JMLR)*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[142] T.-F. Wu, C.-J. Lin, and R. C. Weng, "Probability estimates for multi-class classification by pairwise coupling," *Journal of Machine Learning Research (JMLR)*, vol. 5, no. Aug, pp. 975–1005, 2004.

[143] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.

[144] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[145] J. A. Hartigan, "Clustering algorithms," *Mathematical Physics and Mathematics*, 1975.

[146] I. Jolliffe, "Principal component analysis," in *International Encyclopedia of Statistical Science*. Springer, 2011, pp. 1094–1096.

[147] R. Rehurek and P. Sojka, "Software framework for topic modelling with large corpora," in *In Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Citeseer, 2010.

[148] S. Bird and E. Loper, "Nltk: the natural language toolkit," in *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*. Association for Computational Linguistics, 2004, p. 31.

[149] R. R. Curtin, J. R. Cline, N. P. Slagle, W. B. March, P. Ram, N. A. Mehta, and A. G. Gray, "MLPACK: A scalable C++ machine learning library," *Journal of Machine Learning Research (JMLR)*, vol. 14, no. Mar, pp. 801–805, 2013.

[150] S. Sonnenburg, S. Henschel, C. Widmer, J. Behr, A. Zien, F. d. Bona, A. Binder, C. Gehl, V. Franc *et al.*, "The shogun machine learning toolbox," *Journal of Machine Learning Research (JMLR)*, vol. 11, no. Jun, pp. 1799–1802, 2010.

[151] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *IEEE symposium on Mass storage systems and technologies (MSST)*. Ieee, 2010, pp. 1–10.

[152] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[153] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "Mllib: Machine learning in apache spark," *Journal of Machine Learning Research (JMLR)*, vol. 17, no. 1, pp. 1235–1241, 2016.

[154] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "SystemML: Declarative machine learning on MapReduce," in *International Conference on Data Engineering (ICDE)*. IEEE, 2011, pp. 231–242.

[155] F. Seide and A. Agarwal, "CNTK: Microsoft's open-source deep-learning toolkit," in *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2016.

[156] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the ACM Multimedia (MM)*, 2014.

[157] R. Collobert, S. Bengio, and J. Mariéthoz, "Torch: a modular machine learning software library," Technical Report IDIAP-RR 02-46, IDIAP, Tech. Rep., 2002.

[158] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," in *Proceedings of the Neural Information Processing Systems (NIPS) Workshop*, 2015.

[159] M. L. Hines, T. Morse, M. Migliore, N. T. Carnevale, and G. M. Shepherd, "ModelDB: a database to support computational neuroscience," *Journal of Computational Neuroscience*, vol. 17, no. 1, pp. 7–11, 2004.

[160] M. Akdere, U. Cetintemel, M. Riondato, E. Upfal, and S. B. Zdonik, "The case for predictive database systems: Opportunities and challenges." in *the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2011, pp. 167–174.

[161] X. Feng, A. Kumar, B. Recht, and C. Ré, "Towards a unified architecture for in-rdbms analytics," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, 2012, pp. 325–336.

[162] M. Vartak, P. Ortiz, K. Siegel, H. Subramanyam, S. Madden, and M. Zaharia, "Supporting fast iteration in model building," in *Proceedings of the Neural Information Processing Systems (NIPS) Workshop of Learning System*, 2015.

[163] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proceedings of the International Conference on Computer Vision (ICCV)*, vol. 2, no. 6, 2017.

[164] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, 2015.

[165] S. Antol, A. Agrawal, J. Lu, M. Mitchell, D. Batra, C. L. Zitnick, and D. Parikh, "VQA: Visual question answering," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, vol. 11-18-Dece, 2016, pp. 2425–2433.

[166] H. Dong, S. Yu, C. Wu, and Y. Guo, "Semantic image synthesis via adversarial learning," in *Proceedings of International Conference on Computer Vision (ICCV)*, 2017.

[167] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[168] C. Ju, X. Su, H. Yang, and H. Ning, "Single-image super-resolution reconstruction via generative adversarial network," in *9th International Symposium on Advanced Optical Manufacturing and Testing Technologies: Optoelectronic Materials and Devices for Sensing and Imaging*, vol. 10843. International Society for Optics and Photonics, 2019, p. 108430J.

[169] L. Mukherjee, A. Keikhosravi, D. Bui, and K. W. Eliceiri, "Convolutional neural networks for whole slide image superresolution," *Biomedical Optics Express*, vol. 9, no. 11, pp. 5368–5386, 2018.

[170] C. Wu, G. Wang, J. Zhu, P. Lertvittayakumjorn, S. Hu, C. Tan, H. Mi, Y. Xu, and J. Xiao, "Exploratory analysis for big social data using deep network," *IEEE Access*, 2019.

[171] H. Dong, J. Zhang, D. McIlwraith, and Y. Guo, "I2t2i: Learning text to image synthesis with textual data augmentation," in *Proceedings of the IEEE International Conference on Image Processing (ICIP)*, 2017.

[172] A. Supratak, H. Dong, C. Wu, and Y. Guo, "Deepsleepnet: A model for automatic sleep stage scoring based on raw single-channel eeg," *IEEE Transactions on Neural Systems and Rehabilitation Engineering (TNSRE)*, vol. 25, no. 11, pp. 1998–2008, 2017.

[173] G. Yang, S. Yu, H. Dong, G. Slabaugh, P. L. Dragotti, X. Ye, F. Liu, S. Arridge, J. Keegan, Y. Guo, and D. Firmin, "DAGAN: Deep de-aliasing generative adversarial networks for fast compressed sensing MRI reconstruction," *IEEE Transactions on Medical Imaging (TMI)*, 2017.

[174] S. Yu, H. Dong, G. Yang, G. Slabaugh, P. L. Dragotti, X. Ye, F. Liu, S. Arridge, J. Keegan, D. Firmin *et al.*, "Deep de-aliasing for fast compressive sensing MRI," *arXiv preprint arXiv:1705.07137*, 2017.

[175] S. Niklaus, L. Mai, and F. Liu, "Video frame interpolation via adaptive separable convolution," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 261–270.

[176] Y. I. Parish and P. Müller, "Procedural modeling of cities," in *Proceedings of the Annual Conference on Computer Graphics and Interactive Techniques*, 2001, pp. 301–308.

[177] H. Zhao, S. Zhang, G. Wu, J. M. Moura, J. P. Costeira, and G. J. Gordon, "Adversarial multiple source domain adaptation," in *Proceedings of the Neural Information Processing Systems (NIPS) Conference*, 2018, pp. 8568–8579.

[178] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server." in *Usenix OSDI*, vol. 14, 2014, pp. 583–598.

[179] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A new platform for distributed machine learning on big data," *IEEE Transactions on Big Data*, vol. 1, no. 2, pp. 49–67, 2015.

[180] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.