A TRAINING FRAMEWORK AND ARCHITECTURAL DESIGN FOR DISTRIBUTED DEEP LEARNING

WEI WANG

(*B.Eng., Renmin University of China*)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2016

Supervisors:

Professor Beng Chin Ooi

Examiners:

Professor Wynne Hsu

Associate Professor Roger Zimmermann

Professor Ihab Francis Ilyas, University of Waterloo

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Wang Wei

Wei Wang

24 July 2016

Acknowledgments

First and foremost, I would like to express my special thanks of gratitude to my supervisor, Prof. Ooi Beng Chin for his continuous support of my PhD study and research. He has not only taught me the knowledge and skills of doing research, but also provided me with the valuable opportunity of leading the SINGA project. I have learnt and grown tremendously from his honest criticism, creative advice, great enthusiasm for system research and his positive attitude towards work and exercise.

I greatly appreciate the help from all members in the database group and my collaborators. When I first joined this group, the senior students guided me to start research. They are Xiaoyan Yang, Dongxiang Zhang, Xuan Liu, Wei Lu, Sai Wu, etc. Since the SINGA project was set up, almost all students in the group have contributed immensely to it. They are Sheng Wang, Anh, Jinyang Gao, Zhongle Xie, Kaiping Zheng, Zhaojing Luo, Wenfeng Wu, Chonho Lee, Ji Wang, Xingrui Cai, Xin Ji, Qingchao Cai, Moaz, Qian Lin, Fei He, Haibo Chen, Hao Zhang, and other anonymous contributors. I am also sincerely grateful to all collaborators. They are Prof. Kian-Lee Tan, Prof. Gang Chen, Prof. H.V.Jagadish, Prof. Anthony, Prof. Meihui Zhang, Prof. Yueting Zhang, Prof. Fei Wu, etc.

Also, I have been fortunate to be surrounded with many good friends, including Shangxuan Tian, Haojie Zhang, Peichu Xie, Xiangnan He, Manman Chen, Yifang Yin, Qing Chen, Tao Tang, Meng Zhang, etc. They brought me much happy time during the 5 years.

Lastly, I would like to thank my parents and brother for their unconditional love. Embarking on the PhD journey is not easy and smooth sailing, but their encouragement and love have always been the strongest pillars of support for me. In addition, I am lucky to meet my girlfriend, Qian Zhang, during my study in Singapore. Many thanks to her accompany and editorial help on this thesis.

Contents

List of Figures viii

List of Tables xi

1 Introduction **1**

1.1 Deep Learning and Its Applications 1

1.2 Challenges of Using Deep Learning 2

1.3 Goals and Contributions 3

1.4 Outline 4

2 Background **5**

2.1 Deep Learning 5

2.1.1 Historical Review 5

2.1.2 Training and Inference 7

2.1.3 Sample Models 10

2.1.4 Existing Systems 13

2.2 Multi-modal Applications 15

2.2.1 Deep Learning Approaches 15

2.2.2 Other Approaches 17

2.3 Summary 18

3 Analysis of Optimization Techniques for Deep Learning Systems **19**

3.1 Optimizations for Stand-alone Training 19

3.1.1 Operation Scheduling 19

3.1.2 Memory Management 20

3.2 Optimizations for Distributed Training 21

3.2.1 Communication and Synchronization 22

3.2.2 Concurrency and Consistency 22

3.2.3 Fault Tolerance and Adaptiveness 23
3.3 Optimization Techniques Used in Existing Systems 24
3.4 Summary 25

4 SINGA: A Distributed Deep Learning System 26
4.1 Introduction 26
4.2 System Overview 29
4.3 Programming Model 30
4.3.1 Programming Abstractions 31
4.3.2 Examples 34
4.4 Distributed Training 37
4.4.1 System Architecture 37
4.4.2 Training Frameworks 39
4.4.3 Neural Network Partitioning 41
4.4.4 Optimizations of Communication 43
4.5 Experimental Study 47
4.5.1 Applications of SINGA 47
4.5.2 Training Performance Evaluation on CPU 49
4.5.3 Training Performance Evaluation on GPU 54
4.6 Development Progress 58
4.6.1 Version 0.1 59
4.6.2 Version 0.2 59
4.6.3 Version 0.3 59
4.6.4 Version 1 60
4.7 Summary 65

5 Deep Learning based Approaches for Multi-modal Retrieval 66
5.1 Introduction 66
5.2 Preliminary 69
5.3 General Training Objective 70
5.4 Unsupervised Approach – MSAE 73

5.4.1 Realization of the Learning Objective 73

5.4.2 Training 76

5.5 Supervised Approach–MDNN 78

5.5.1 Realization of the Learning Objective 78

5.5.2 Training 80

5.6 Query Processing 81

5.7 Experimental Study 82

5.7.1 Evaluation Metrics 83

5.7.2 Experimental Study of MSAE 83

5.7.3 Experimental Study of MDNN 100

5.8 Summary 106

6 Conclusion and Future Work **108**

6.1 Summary 108

6.2 Future Work 109

6.2.1 Deep learning system optimization 109

6.2.2 Multi-modal data analysis with deep learning 110

References **111**

List of Figures

1.1 Sample applications of deep learning. 1

2.1 Categories of popular deep learning models. 7

2.2 Illustration of stochastic gradient descent (SGD). 9

2.3 Data flow of Back-Propagation. 9

2.4 Auto-encoder. 11

2.5 Fine-tune stacked auto-encoders. 11

3.1 Sample operations from a deep learning model. 20

4.1 SINGA overview. 29

4.2 Running example using an MLP. 30

4.3 Layer abstraction. 32

4.4 Structure of MDNN. 34

4.5 Structure of RBM and deep auto-encoders. 36

4.6 Structure of 2-stacked Char-RNN (left before unrolling; right after unrolling). 37

4.7 Logical architecture of SINGA. 38

4.8 Training frameworks in SINGA. 39

4.9 Partition the hidden layer in Figure 4.2a. 41

4.10 Distributed computing for fully connected layers. 44

4.11 Parallelize computation and communication for a GPU worker. 46

4.12 Multi-Modal Retrieval. Top 5 similar text documents (one line per document) and images are displayed. 48

4.13 Visualization of the weight matrix in the bottom RBM and top layer features in the deep auto-encoder. 49

4.14 Training accuracy and loss of Char-RNN. 49

4.15 Synchronous training. 51

4.16 Asynchronous training. 52

4.17 Effect of optimization techniques. 55

4.18 Performance comparison of open source systems. 57

4.19 The Device API. 60

4.20 The Tensor API. 61

4.21 The Layer API. 63

4.22 The FeedForwardNet API. 63

5.1 Flowchart of multi-modal retrieval framework. Step 1 is offline model training that learns mapping functions. Step 2 is offline indexing that maps source objects into latent features and creates proper indexes. Step 3 is online multi-modal kNN query processing. 71

5.2 Flowchart of training. Relevant images (or text) are associated with the same shape (e.g., ■). In single-modal training, objects of same shape and modality are moving close to each other. In multi-modal training, objects of same shape from all modalities are moving close to each other. 72

5.3 Model of MSAE, which consists of one SAE for each modality. The trained SAE maps input data into latent features. 73

5.4 Distribution of image (5.4a) and text (5.4b) features extracted from NUS-WIDE training dataset (See Section 5.7). Each figure is generated by averaging the units for each feature vector, and then plot the histogram for all data. 74

5.5 Model of MDNN, which consists of one DCNN for image modality, and one Skip-Gram + MLP for text modality. The trained DCNN (or Skip-Gram + MLP) maps input data into latent features. 77

5.6 Illustration of query processing. 81

5.7 Visualization of latent features after projecting them into 2D space (Blue points are image latent features; White points are text latent features. Relevant image-tex pairs are connected using red lines). 87

5.8 Adjusting image SAE with different β_I and text SAE fixed (a-d show the positions of features of image-text pairs in 2D space). 88
5.9 Adjusting text SAE with different β_T and image SAE fixed (a-d show the positions of features of image-text pairs in 2D space). 90
5.10 Precision-Recall (P-R) and Recall-Candidates ratio on NUS-WIDE dataset. 94
5.11 Training cost comparison on Flickr1M dataset. 99
5.12 Querying time comparison using real-valued and binary latent features. 100
5.13 Visualization of training on NUS-WIDE-a. 101
5.14 Visualization of training on NUS-WIDE-b. 103
5.15 Visualization of latent features learned by MDNN for the test dataset of NUSWDIE-a (features represented by the same shapes and colors are annotated with the same label). 104
5.16 Training cost comparison on NUSWIDE-a dataset. 105

List of Tables

2.1 A brief summary of existing systems. 14

3.1 Summary of optimization techniques used in existing systems as of July 2016. 25

4.1 Layer categories. 33

4.2 Specs of hardware and software. 54

5.1 Statistics of Datasets for Unsupervised Training. 84

5.2 Mean average precision on NUS-WIDE dataset. 92

5.3 Mean average precision on NUS-WIDE dataset (using Binary Latent Features). 95

5.4 Mean average precision on Wiki dataset. 96

5.5 Mean average precision on Flickr1M dataset. 98

5.6 Statistics of datasets for supervised training. 101

5.7 Mean average precision using real-valued latent feature. 103

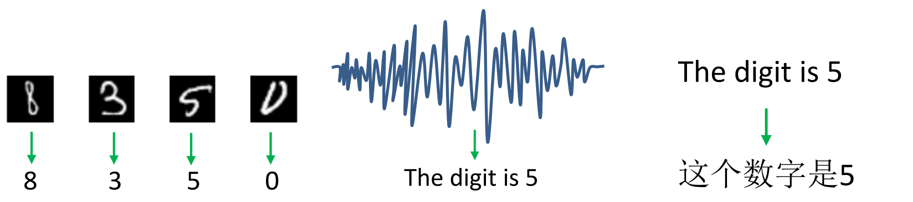
Chapter 1

Introduction

Deep learning (LeCun, Bengio, and Hinton, 2015) refers to a branch of machine learning models, which perform exceptionally well in learning rich representations of data. A deep learning model typically consists of multiple feature transformation layers for extracting high-level features from raw input data. By tuning the model parameters and structures (e.g. layer size and connections), the extracted features could improve the accuracy significantly for tasks of interest.

1.1 Deep Learning and Its Applications

Deep learning is regarded as a re-branding of neural networks developed twenty years ago, as it inherits many key neural networks techniques and algorithms. Its recent resurgence is mainly fueled by its excellent performance for a wide range of tasks. For example, the deep convolutional neural network (Krizhevsky, Sutskever, and Hinton, 2012) has made break-through progress for computer vision tasks, including image classification (Figure 1.1a) and retrieval (Wan et al.,



(a) Image classification. (b) Speech recognition. (c) Machine translation.

Figure 1.1: Sample applications of deep learning.

2014). Deep learning also made big improvement in acoustic modeling (Abdel-rahman Mohamed, Dahl, and Hinton, 2012) (Figure 1.1b) via the combination of a deep multi-layer perceptron model and a hidden Markov model. Recently, deep learning has become popular for textual data applications (Goldberg, 2015; Collobert et al., 2011), e.g. machine translation (Figure 1.1c) based on recurrent neural networks (Sutskever, Vinyals, and Le, 2014). With the ability of learning rich representations of different types of mono-modal data, deep learning has the potential to learn adaptive representations of multi-modal data. Multi-modal data is emerging in online social media and e-commerce platforms, including temporally synchronized data (e.g. video clips and audio transcripts), spatially related data (e.g. point of interest and user travel history), or semantically connected data (e.g. images and tags). One challenge is that different modalities have different properties, e.g. distributions and raw representations. The deep learning models should be able to bridge the gap between different modalities.

1.2 Challenges of Using Deep Learning

Before deploying a deep learning model in an application, we need to train the model. The training procedure tunes parameters involved in the model to optimize an objective function, e.g. a function that measures the error between the prediction and the ground truth. Models with complex structures and a large amount of parameters take a long time to train (Szegedy et al., 2014; Simonyan and Zisserman, 2014), and are prone to local optimal solutions or overfitting[1]. With the advancements of high performance computing devices such Graphic Processing Units (GPU) and large labeled datasets like ImageNet (Deng et al., 2009), we are able to train a big model to capture rich data semantics. However, there are still two major challenges in training large deep learning models, namely efficiency and usability.

[1]which would result in a model with good performance for the training data but poor performance for the test data

First, it takes long time (e.g. several weeks) to train some large models (Krizhevsky, Sutskever, and Hinton, 2012; Szegedy et al., 2014) even using GPUs. Moreover, the training programs for large models consume a vast amount of memory, which restricts the model complexity and size since GPU devices have limited memory. Optimizations in terms of memory efficiency and space efficiency are crucial. Distributed training has been introduced to accelerate the training process. However, there are many types of overhead that may affect the system scalability, including communication and synchronization. A flexible system architecture would be necessary for studying the various effects to optimize the training.

Second, deep learning models have complex structures (e.g. more than 20 layers in (Szegedy et al., 2014)), especially for models from multi-modal applications, which makes it difficult for non-experts to implement these models. Training in a large cluster with big datasets is more challenging. It is significant to provide good programming models and user interfaces to let non-experts train different models with little effort.

1.3 Goals and Contributions

The goal of this dissertation is to propose a deep learning system that is easy to use, efficient and extensible for complex applications. In particular, the contributions are as follows:

- We give a comprehensive investigation and analysis on the optimization techniques for deep learning systems. Many issues will be considered, including efficiency, memory footprint, communication, fault-tolerance, etc. We would also discuss some optimization techniques from database systems that could be adapted for deep learning systems including operation scheduling and memory management.

- Following the investigation, we describe the design and implementation of our deep learning system named SINGA for distributed training. SINGA provides an intuitive programming model base on the common abstraction (i.e., *layer*) of deep learning models. The training is conducted over a dataflow graph consisting of layers. Distributed training is enabled almost transparently to users by partitioning the graph among workers and run each worker over its own sub-graph. Flexible system architecture enables users to exploit different distributed training frameworks to minimize the training time. Our experience with developing and training deep learning models using SINGA shows that the platform is both usable and scalable. The SINGA system has been accepted as an Apache incubator project.

- Besides the system research, we also present our approaches for multi-modal retrieval using deep learning techniques. This application would be used to verify SINGA's capability of handling large complex models. Multi-modal retrieval enables users to query information from different modalities of data. For example, it allows users to take a snap of product to search relevant descriptions and reviews. The challenge is to learn effective mapping functions to extract common representations for data from different modalities. Two approaches will be introduced: (1) an unsupervised approach that uses stacked auto-encoders (SAEs) and requires minimum prior knowledge on the training data, and (2) a supervised approach using deep convolutional neural network (DCNN) and neural language model (NLM). Experimental results on three real datasets demonstrate that our methods achieve significant improvement in search accuracy over the state-of-the-art solutions.

1.4 Outline

The rest of this dissertation is organized as follows: Chapter 2 reviews deep learning systems and the multi-modal retrieval application; the SINGA system is introduced in Chapter 4; Chapter 5 describes our approaches for the multi-modal retrieval application; This dissertation is concluded in Chapter 6.

Chapter 2

Background

This chapter will first give a brief review on the progress of deep learning techniques, including models, training algorithms and applications. After that, there is a discussion on the challenges and opportunities of developing deep learning systems, which provides some guides for the development of our SINGA system. Finally, related works on the multi-modal data retrieval problem will be presented, which is an example application of the SINGA system.

2.1 Deep Learning

2.1.1 Historical Review

Deep learning refers to a set of machine learning models which attempt to learn high-level abstractions of raw data through multiple feature transformation layers. Features extracted from high-level layers typically have better performance than hand-crafted features in many tasks, e.g. image classification and acoustic modeling. Deep learning models with multiple layers were called artificial neural nets (abbr. networks) in history dating back to the 1950s (Selfridge, 1958; Rosenblatt, 1957). Modern neural net architectures were developed around the 1980s (LeCun, 1985; Rumelhart, Hinton, and Williams, 1986), when the back-propagation algorithm was applied to train multiple layer neural nets. Various neural nets were then proposed for different prediction tasks, e.g. speech recognition (Waibel et al., 1989) and document reading (LeCun et al., 1998).

However, these neural nets were found to be hard to train, because it was easy to get trapped in local optima with little training data. Moreover, it was time consuming to train even middle sized neural nets because the hardware (CPU) was slow.

In 2006, Hinton et al. (Hinton, Osindero, and Teh, 2006) proposed a new method to train one type of deep neural nets, called Deep Belief Networks (DBN). This method uses Restricted Boltzmann Machine (RBM) to initialize layers one by one. RBM can exploit a large amount of unlabeled data to initialize the model parameters into sensible values; therefore it helps to train a good model. RBM and DBN are in the energy model category in Figure 2.1. DBN made a big improvement for speech recognition (Dahl et al., 2012; Abdel-rahman Mohamed, Dahl, and Hinton, 2012) with the help of Graphic Processing Units (GPU), which is 10 to 20 times faster than CPU. Since then, deep learning has been referred to as deep neural nets.

With GPUs and a large labeled training dataset, i.e. ImageNet (Deng et al., 2009), Alex et al. (Krizhevsky, Sutskever, and Hinton, 2012) proposed a Deep Convolutional Neural Net (DCNN) in 2012, also called AlexNet, which made a breakthrough improvement for image classification. AlexNet shares a similar architecture as the LeNet (LeCun et al., 1998) developed in the 1990s, but has a deeper and larger structure, which increases its capability of capturing richer data representations. Since then, many variants of DCNN have been proposed (Simonyan and Zisserman, 2014; Szegedy et al., 2014) with much deeper structures and larger sizes of model parameters. They dominate almost all computer vision tasks. However, the training still takes a long time, e.g. several weeks.

Another set of popular neural networks is called Recurrent Neural Networks (RNN). Modern RNNs were developed in the 1990s for modeling sequential data, including sentences and time series data. However, the training of RNN is much harder than other neural nets due to the gradient vanishing and exploding

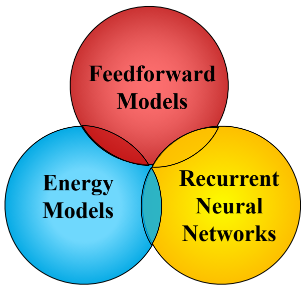


Figure 2.1: Categories of popular deep learning models.

problems (Hochreiter, 1991; Bengio, Simard, and Frasconi, 1994). Many variants were proposed to address this issue, including LSTM (Hochreiter and Schmidhuber, 1997) and GRU (Cho et al., 2014). Together with the advancements of training algorithms, it was shown recently that RNN is practical for a broad range of applications such as image caption generation (Mao et al., 2014) and machine translation (Sutskever, Vinyals, and Le, 2014).

DBN, DCNN and RNN are the three most popular deep learning models as shown in Figure 2.1. They have achieved state-of-the-art performance for various tasks. However, these models have different structures and training algorithms. It is tedious and may not be even possible for a deep learning user or researcher to implement the training and inference algorithms 2.1.2 for these models from scratch. The case is more complex for applications with multi-modal data, where one deep learning model is applied for one modality. It is essential to have a training system that is efficient and extensible for wide range of applications.

2.1.2 Training and Inference

There are two tasks of a deep learning system, namely training and inference. Before deploying a deep learning model, the model parameters involved in the transformation layers need to be trained. The training turns out to be a numeric optimization procedure to find parameter values that minimize the discrepancy (loss function) between the expected output and the real output. Stochastic

Gradient Descent (SGD) is the most widely used training algorithm. As shown in Figure 2.2, SGD initializes the parameters with random values, and then iteratively refines them based on the computed gradients with respect to the loss function. There are three commonly used algorithms for gradient computation corresponding to the three model categories above: Back Propagation (BP), Contrastive Divergence (CD) and Back Propagation Through Time (BPTT). By regarding the layers of a neural net as nodes of a graph, these algorithms can be evaluated by traversing the graph in certain sequences. For instance, the BP algorithm is illustrated in Figure 2.3, where a simple feedforward model is trained by forward propagating along the solid arrows to compute the data (feature) of each layer, and backward propagating along the dashed arrows to compute the gradient of each layer and each parameter. Equation 2.1-2.3 are for the forward propagation, where $\boldsymbol{x}$ is the input feature vector and l is the squared Euclidean loss. Equation 2.4- 2.7 are for the backward propagation, where W' and $\boldsymbol{b'}$ are gradients. Equation 2.8 and 2.9 update the parameters along the direction of decreasing the objective loss value, where α controls the updating step. For distributed training, multiple workers (e.g., machines or GPU cards) run SGD synchronously or asynchronously. For instance, the Hogwild!(Recht et al., 2011) training framework uses multiple threads (on a single node with memory shared) to compute the gradients and update the parameters independently (i.e., asynchronously). More details on the synchronous and asynchronous training are discussed in Section 4.4.

Inference is the procedure of extracting representations for new data, which propagates the raw input through all layers of a neural net without changing the model parameters. For example, the inference procedure of the simple neural net shown in Figure 2.3 would compute Equation 2.1-2.2 in order. Given that the inference is much simper than training, we will focus on the training procedure for the rest of this dissertation.

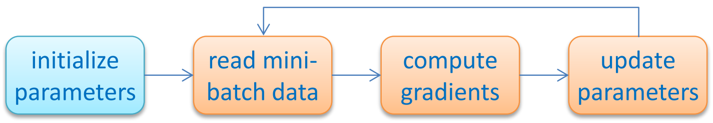


Figure 2.2: Illustration of stochastic gradient descent (SGD).

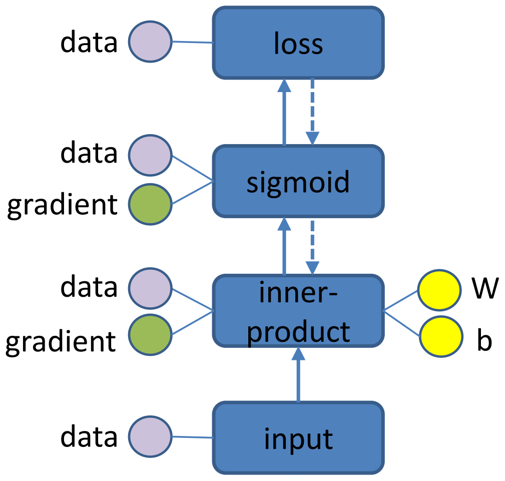


Figure 2.3: Data flow of Back-Propagation.

$$\boldsymbol{a} = W\boldsymbol{x} + \boldsymbol{b} \tag{2.1}$$

$$\boldsymbol{h} = \sigma(\boldsymbol{a}) = \frac{1}{1+e^{-\boldsymbol{x}}} \tag{2.2}$$

$$l = ||\boldsymbol{h} - \boldsymbol{y}||_2^2 \tag{2.3}$$

$$\boldsymbol{h}' = \boldsymbol{h} - \boldsymbol{y} \tag{2.4}$$

$$\boldsymbol{a}' = \boldsymbol{h}' * \boldsymbol{h} * (\boldsymbol{1} - \boldsymbol{h}) \tag{2.5}$$

$$W' = \boldsymbol{a}' \cdot \boldsymbol{x}^T \tag{2.6}$$

$$\boldsymbol{b}' = \boldsymbol{a}' \tag{2.7}$$

$$W = W - \alpha * W' \tag{2.8}$$

$$b = b - \alpha * b' \tag{2.9}$$

2.1.3 Sample Models

Three specific deep learning models are introduced below, which would be used in Chapter 5 for multi-modal retrieval.

Auto-encoder

Auto-encoder has been widely used in unsupervised feature learning and classification tasks (Rifai et al., 2011; Vincent et al., 2008; Goroshin and LeCun, 2013; Socher et al., 2011). It can be seen as a special neural network with three layers – the input layer, the latent layer, and the reconstruction layer. As shown in Figure 2.4, the raw input feature $x_0 \in \mathcal{R}^{d_0}$ in the input layer is **encoded** into latent feature $x_1 \in \mathcal{R}^{d_1}$ via a deterministic mapping f_e:

$$x_1 = f_e(x_0) = s_e(W_1^T x_0 + b_1) \tag{2.10}$$

where s_e is the activation function of the encoder, $W_1 \in \mathbb{R}^{d_0 \times d_1}$ is a weight matrix and $b_1 \in \mathcal{R}^{d_1}$ is a bias vector. The latent feature x_1 is then **decoded** back to $x_2 \in \mathcal{R}^{d_0}$ via another mapping function f_d:

$$x_2 = f_d(x_1) = s_d(W_2^T x_1 + b_2) \tag{2.11}$$

Similarly, s_d is the activation function of the decoder with parameters $\{W_2, b_2\}$, $W_2 \in \mathbb{R}^{d_1 \times d_0}$, $b_2 \in \mathcal{R}^{d_0}$. Sigmoid function or Tanh function is typically used as the activation functions s_e and s_d. The parameters $\{W_1, W_2, b_1, b_2\}$ of the auto-encoder are learned with the objective of minimizing the difference (called reconstruction error) between the raw input x_0 and the reconstruction output x_2. Squared Euclidean distance, negative log likelihood and cross-entropy are often used to measure the reconstruction error. By minimizing the reconstruction error, we can use the latent feature to reconstruct the original input with minimum information loss. In this way, the latent feature preserves regularities (or semantics) of the input data.

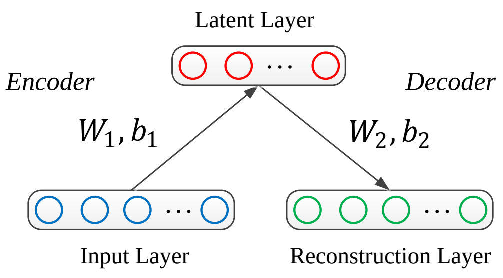


Figure 2.4: Auto-encoder.

Stacked Auto-encoder

Stacked Auto-encoders (SAE) are constructed by stacking multiple (e.g., h) auto-encoders. The input feature vector x_0 is fed to the bottom auto-encoder. After training the bottom auto-encoder, the latent representation x_1 is propagated to the higher auto-encoder. The same procedure is repeated until all the auto-encoders are trained. The latent representation x_h from the top (i.e., h-th) auto-encoder, is the output of the stacked auto-encoders, which can be further fed into other applications, such as SVM for classification. The stacked auto-encoders can be fine-tuned by minimizing the reconstruction error between the input feature x_0 and the reconstruction feature x_{2h} which is computed by forwarding the x_0 through all encoders and then through all decoders as shown in Figure 2.5. In this way, the output feature x_h can reconstruct the input feature with minimal information loss. In other words, x_h preserves regularities (or semantics) of the input data x_0.

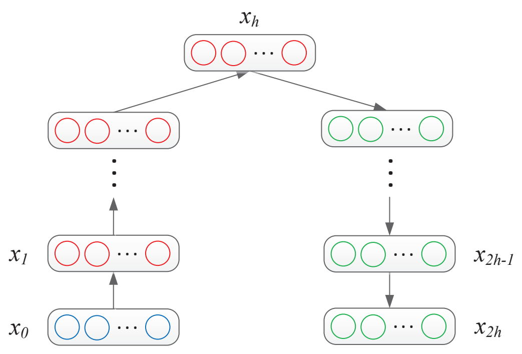


Figure 2.5: Fine-tune stacked auto-encoders.

Deep Convolutional Neural Network

Deep Convolutional Neural Network (DCNN) has shown great success in computer vision tasks (Donahue et al., 2013; Girshick et al., 2014) since the first DCNN (called AlexNet) was proposed by Alex (Krizhevsky, Sutskever, and Hinton, 2012). It has specialized connectivity structure, which usually consists of multiple convolutional layers followed by fully connected layers. These layers form stacked, multiple-staged feature extractors, with higher layers generating more abstract features from lower ones. On top of the feature extractor layers, there is a classification layer.

The input to DCNN contains raw image pixels such as a vector of RGB values, which is forwarded through all feature extractor layers to generate a feature vector that is a high-level abstraction of the input data. The training data of DCNN consists of image-label pairs. Let x denote the image raw feature and $f_I(x)$ the feature vector extracted from DCNN. t is the binary label vector of x. If x is associated with the i-th label l_i, t_i is set to 1 and all other elements are set to 0. $f_I(x)$ is forwarded to the classification layer to predict the final output $p(x)$, where $p_i(x)$ is the probability of x being labeled with l_i. Given x and $f_I(x)$, $p_i(x)$ is defined as:

$$p_i(x) = \frac{e^{f_I(x)_i}}{\sum_j e^{f_I(x)_j}} \tag{2.12}$$

which is a Softmax function. Based on Equation 2.12, the prediction error is defined as the negative log likelihood:

$$\mathcal{L}_I(x,t) = -\sum_i t_i \log p_i(x) \tag{2.13}$$

Neural Language Model

Neural Language Model (NLM), first introduced in (Bengio et al., 2003), learn a dense feature vector for each word or phrase, called a distributed representation or a *word embedding*. Among them, the Skip-Gram model (SGM) (Mikolov et al., 2013) proposed by Mikolov *et al.* is the state-of-the-art. Given a word a and context b that co-occur, SGM models the conditional probability $p(a|b)$ using Softmax:

$$p(a|b) = \frac{e^{v_a \cdot v_b}}{\sum_{\tilde{a}} e^{v_{\tilde{a}} \cdot v_b}} \tag{2.14}$$

where v_a and v_b are vector representations of word a and context b respectively. The denominator $\sum_{\tilde{a}} e^{v_{\tilde{a}} \cdot v_b}$ is expensive to calculate given a large vocabulary, where $\tilde{a}$ is any word in the vocabulary. Thus, approximations were proposed to estimate it (Mikolov et al., 2013). Given a corpus of sentences, SGM is trained to learn vector representations v by maximizing Equation 2.14 over all co-occurring pairs.

The learned dense vectors can be used to construct a dense vector for one sentence or document (e.g., by averaging), or to calculate the similarity of two words, e.g., using the cosine similarity function.

2.1.4 Existing Systems

Before we started developing our system, there were a couple of deep learning systems, namely Caffe (Jia et al., 2014a), Torch (Collobert, Kavukcuoglu, and Farabet, 2011), Theano (Bastien et al., 2012), Google Brain (Dean et al., 2012a) and the Adam project (Chilimbi et al., 2014) from Microsoft. However, they were either closed source or lack of distributed training support. Hence, we started developing our open source system (SINGA), aiming to accelerate the training speed via distributed computing. More systems were released recently, including TensorFlow (Abadi et al., 2015), CNTK (Yu et al., 2014) and MxNet (Chen et al., 2015). Table 2.1 gives a brief summary of these systems. Google Brain and the Adam project are closed-source, rendering them unusable by common users. Optimization techniques used in these systems would be discussed in Section 3.3.

Two major programming styles are used in the existing systems, namely im-

Table 2.1: A brief summary of existing systems.

System /Library	**License**	**Start Time**	**Distributed**	**Programming Style**	**Hardware**
Theano	BSD	2008	N	Declarative	GPU/CPU
Torch	BSD	2012	N	Imperative	GPU/CPU
Caffe	BSD	2013	N	Imperative	GPU/CPU
SINGA	Apache V2	2014	Y	Imperative	GPU/CPU
MxNet	Apache V2	2015	Y	Mixed	GPU/CPU
TensorFlow	Apache V2	2015	Y	Declarative	GPU/CPU
Google Brain	closed	2012	Y	-	CPU
Adam	closed	2013	Y	-	CPU

'L' for Linux; 'M' for Mac OS; 'W' for Windows. '-': unknown.

perative programming and declarative programming. Caffe (Jia et al., 2014a) and Torch (Collobert, Kavukcuoglu, and Farabet, 2011) use imperative programming, which is easy to get started and debug. TensorFlow (Abadi et al., 2015), Theano (Bastien et al., 2012) and CNTK (Yu et al., 2014) follow the declarative programming model, where users simply declare the learning objective and then the system would create a computation graph (dataflow graph) for automatically optimizing the learning objective. The computation graph provides opportunities for speed and memory optimization (Chen et al., 2015), but is not easy to debug and requires some effort to get started.

Layer is an inherent abstraction of neural networks. Almost all systems provide the layer abstraction (may use different names). Caffe uses Layer as the lowest computation unit, which was designed for feed-forward neural networks, and extended to support RNN, but has no support for energy models. Other systems provide Tensor abstractions for algebra operations, which are more flexible to implement general machine learning algorithms.

Caffe, Torch and Theano focused on training in a single node using CPU or a single GPU. Hence, they lack optimization for distributed training. TensorFlow, CNTK and MxNet have more optimization techniques for distributed training. A couple of papers on the use of Multi-GPUs to train DCNNs have been published (Krizhevsky, 2014; Paine et al., 2013; Yadan et al., 2013). However, these are model specific and do not generalize well to other models. In contrast, the well-known general data processing systems like MapReduce (Dean and Ghemawat, 2004) and Spark (Zaharia et al., 2012) lack optimizations specific to deep learning models.

2.2 Multi-modal Applications

Multi-modal retrieval is emerging as a new search paradigm that enables seamless information retrieval from various types of media. For example, users can simply snap a movie poster to search for relevant reviews and trailers. We study multi-modal applications with two motivations. Firstly, considering that deep learning is good at feature learning for mono-modal data, e.g. image, text and audio data, it has the potential to learn adaptive representations to bridge the gap between different modalities. Secondly, the models for multi-modal retrieval would be more complex than those for mono-modal data applications, and thus are good applications for verifying the usability, efficiency and extensibility of SINGA.

The key problem of multi-modal retrieval is to find an effective mapping mechanism, which maps data from different modalities onto a common latent space. An effective mapping mechanism would preserve both intra-modal semantics and inter-modal semantics well in the latent space, and thus generates good retrieval performance.

2.2.1 Deep Learning Approaches

Multi-modal deep learning (Ngiam et al., 2011; Srivastava and Salakhutdinov, 2012) could be the first work of extending deep learning to multi-modal scenario. (Srivastava and Salakhutdinov, 2012) connects two deep Boltzmann machines

(DBM) (one for image, one for text) by a common latent layer to construct a Multi-modal DBM. The representation of the common latent layer is a fused feature for images and text, which can be extracted using data from either a single modality or both modalities. Pairs of semantically relevant image and text documents are fed into the model for training, which updates the model parameters to maximize the probability of the training pairs. (Ngiam et al., 2011) constructs a Bimodal deep auto-encoder with two deep auto-encoders (one for audio, one for video). The two paths join at a hidden layer that fuses the features, and then depart to reconstruct the input image-text pair. The objective is to minimize the reconstruction errors to capture the regularities of the input data in the fused feature. Both two models aim to improve the classification accuracy of objects with features from multiple modalities. They combined different features to learn a good (high dimensional) latent feature. In this dissertation, we will propose approaches for learning low-dimensional latent features to enable effective and efficient multi-modal retrieval.

DeViSE (Frome et al., 2013) from Google shares the similar idea with one of our approach. It embeds image features into text space directly, which are then used to retrieve similar text features for zero-shot learning. In particular, it extracts the image feature from a pre-trained DCNN model and then transforms the feature via linear projection into the same space as the text feature, which is extracted from the Skip-Gram model (Mikolov et al., 2013). Our approach embeds both image features (extractd from DCNN) and text features (extracted from Skip-Gram model and MLP) into the same latent space. Notice that the text features used in DeViSE to learn the embedding function are generated from high-quality labels. However, in multi-modal retrieval, queries usually do not come with labels and text features are generated from noisy tags. This makes DeViSE less effective in learning robust latent features against noisy input compare to our approach.

2.2.2 Other Approaches

Early works like (Zhuang, Yang, and Wu, 2008) and (Rasiwasia et al., 2010) try to exploit the correlations of data via correlation graph from different modalities to find the latent space. Recently, linear projection has been studied to solve this problem (Kumar and Udupa, 2011; Song et al., 2013; Zhu et al., 2013). The main idea is to find a linear projection matrix for each modality that maps semantic relevant data into similar latent vectors. However, when the distribution of the original data is non-linear, it would be hard to find a set of effective projection matrices. CVH (Kumar and Udupa, 2011) extends the Spectral Hashing (Weiss, Torralba, and Fergus, 2008) for multi-modal data by finding a linear projection for each modality that minimizes the Euclidean distance of relevant data in the latent space. Similarity matrices for both inter-modal data and intra-modal data are required to learn a set of good mapping functions. IMH (Song et al., 2013) learns the latent features of all training data first, and then it finds a hash function to map the input data and output latent features. It could be very expensive in terms of memory and computation to learn the representations of all data together. LCMH (Zhu et al., 2013) exploits the intra-modal correlations by representing data from each modality using its distance to cluster centroids of the training data. Projection matrices are then learned to minimize the distance of relevant data (e.g., image and tags) from different modalities.

Besides linear projection, another kind of approach is based on Latent Dirichlet Allocation (LDA) (Blei and Jordan, 2003; Putthividhya, Attias, and Nagarajan, 2010). They try to represent each image or text with a latent topic vector. LDA works well for text modality, but directly applying it for image modality may not perform very well (Wang and Grimson, 2007).

Other recent works include CMSSH (Bronstein et al., 2010), MLBE (Zhen and Yeung, 2012) and LSCMR (Lu et al., 2013). CMSSH uses a boosting method to learn the projection function for each dimension of the latent space. However, it requires prior knowledge such as both semantic relevant and irrelevant pairs.

MLBE explores correlations of data (both inter-modal and intra-modal similarity matrices) to learn latent features of training data using a probabilistic graphic model. Given a query, it is converted into the latent space based on its correlation with the training data. Such correlation is decided by labels associated with the query. However, labels of a query are usually not available in practice, which makes it hard to obtain its correlation with the training data. LSCMR (Lu et al., 2013) learns the mapping functions with the objective to optimize the ranking criteria (e.g., MAP) directly. Ranking examples (a ranking example is a query and its ranking list) are needed for training. In Chapter 5, we will present an approach which uses simple relevant pairs (e.g., image and its tags) as training input. No prior knowledge such as irrelevant pairs, similarity matrix, ranking examples and labels of queries, is needed.

2.3 Summary

In this chapter, we described the progress of deep learning, which was fueled mainly by three factors, immense computing power, big training dataset, and advancements of neural net structures. Deep learning has become crucial for many applications across computer vision, speech, and NLP. However, the training programs are non-trivial to implement from scratch, and is slow and memory-hungry. We introduced some existing deep learning systems, which would be analyzed in Chapter 3 in terms of their optimization techniques. In addition, related works for multi-modal applications were reviewed, for which we will propose deep learning based approaches in Chapter 5.

CHAPTER 3

Analysis of Optimization Techniques for Deep Learning Systems

This chapter discusses challenges and opportunities of optimizing the deep learning training procedure from the system perspective. We leave the theory perspective optimization as a future work, including convergence analysis of asynchronous SGD like Hogwild! (Recht et al., 2011).

3.1 Optimizations for Stand-alone Training

Currently, the most effective approach for improving the training speed of deep learning models is to use Nvidia GPU with the cuDNN library[1]. Researchers are also working on other hardware, e.g. FPGA[2] (Lacey, Taylor, and Areibi, 2016). Besides exploiting advancements in hardware technology, operation scheduling and memory management are two important components to consider.

3.1.1 Operation Scheduling

Training algorithms of deep learning models typically involve expensive linear algebra operations as shown in Figure 3.1, where the matrix $W1$ and $W2$ could be larger than $4096*4096$. Operation scheduling is to first detect the data dependency

[1]https://developer.nvidia.com/cudnn

[2]short for field-programmable gate array.

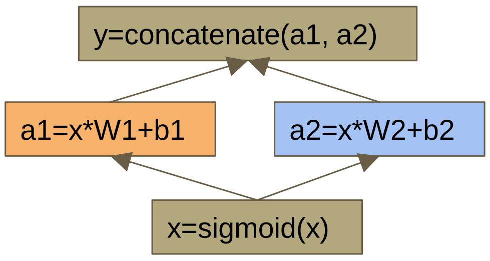


Figure 3.1: Sample operations from a deep learning model.

of operations and then place the operations without dependencies onto executors, e.g., CUDA (Compute Unified Device Architecture) streams and CPU threads. Take the operations in Figure 3.1 as an example, $\boldsymbol{a1}$ and $\boldsymbol{a2}$ in Figure 3.1 could be computed in parallel because they have no dependencies. The first step could be done statically based on the dataflow graph or dynamically (Chen et al., 2015) by analyzing the orders of read and write operations. Databases also have this kind of problems in optimizing transaction execution (Yao et al., 2016) and query plans. Those solutions should be considered for deep learning systems. For instance, databases use cost models to estimate query plans. For deep learning, we may also create a cost model to find an optimal operation placing strategy for the second step of operation scheduling given a fixed computing resources including executors and memory.

3.1.2 Memory Management

Deep learning models are becoming larger and larger, and they are already occupying a huge amount of memory space. For example, the VGG model (Simonyan and Zisserman, 2014) cannot be trained on normal GPU cards due to memory size constraints. Many approaches have been proposed towards reducing the memory consumption. Shorter data representation, e.g. 16-bit float (Courbariaux, Bengio, and David, 2014) is now supported by CUDA. Memory sharing is an effective approach for memory saving (Chen et al., 2015). Take Figure 3.1 as an example, the input and output of the *sigmoid* function share the same variable and thus the same memory space. Such operations are called 'in-place' operations. Another memory sharing case is illustrated in Figure 2.3. In each iteration, the data variable of the *inner-product* layer will not be used after finishing the forward propagation of the *sigmoid* layer, hence its memory space can be reused by the gradient of the *sigmoid* layer afterwards. The dataflow graph is necessary for finding such variables (Chen et al., 2015). Recently, two approaches were proposed to trade-off computation time for memory. Swapping memory between GPU and CPU resolves the problem of small GPU memory and large model size by swapping variables out to CPU and then swapping back manually(Cui et al., 2016). Another approach drops some variables to free memory and recomputes them when necessary based on the static dataflow graph(Chen et al., 2016).

Memory management is a hot topic in the database community with extensive research done towards in-memory databases (Tan et al., 2015; Zhang et al., 2015), including locality, paging and cache optimization. To elaborate more, the paging strategies could be useful for deciding when and which variable to swap. In addition, failure recovery in databases is similar to the idea of dropping and recomputing variables, hence the logging techniques in databases could be considered. If all operations (and execution time) are logged, we can then do runtime analysis without the static dataflow graph. Other techniques, including garbage collection and memory pool, would also be useful for deep learning systems, especially for GPU memory management.

3.2 Optimizations for Distributed Training

Distributed training is a natural solution for accelerating the training speed of deep learning models. The parameter server architecture (Dean et al., 2012b) is typically used, in which the workers compute parameter gradients and the servers update the parameter values after receiving gradients from workers. There are two basic parallelism schemes for distributed training, namely, data parallelism and model parallelism. In data parallelism, each worker is assigned a data partition and a model replica; while for model parallelism, each worker is assigned a partition of the model and the whole dataset.

We investigated some common issues of distributed computing, which are discussed below.

3.2.1 Communication and Synchronization

Given that deep learning models have a large set of parameters, the communication overhead between workers and servers is likely to be the bottleneck of a training system. It becomes worse when the workers are running on GPUs, which decrease the computation time and thus increase the communication time relatively. In addition, for large clusters, the synchronization between workers can be significant. Consequently, it is important to investigate efficient communication protocols for both single-node multiple GPU training and training over a large cluster. Possible research directions include: a) compressing the parameters and gradients for transmission (Seide et al., 2014); b) organizing servers in an optimized topology to reduce the communication burden of each single node, e.g., tree structure (Gupta, Zhang, and Milthorpe, 2015) and AllReduce structure (Wu et al., 2015) (all-to-all connection); c) using more efficient networking hardware like RDMA (Coates et al., 2013).

3.2.2 Concurrency and Consistency

Currently, both declarative programming (e.g., Theano and TenforFlow) and imperative programming (e.g., Caffe and SINGA) have been adopted in existing systems for concurrency implementation. Most deep learning systems use threads and locks directly. Other concurrency implementation methods like actor model (good at failure recovery), co-routine and communicating sequential processes have not been explored. Co-routine would be useful for asynchronous BP and parameter updating. To be specific, as shown in Algorithm 3.1, in Line 2, we create a future object which serves as a channel that receives pairs of parameter

Algorithm 3.1: Use co-routine in BP

```
1 ...
2 future=net.backward() // run in another thread
3 foreach $(W, W')$ in future do
4     $W = W - \alpha * W'$ ; // or transfer $W'$ to servers
5 end
```

values and gradients computed by the backward propagation procedure in another thread. In this way, we can easily run the backward propagation and parameter updating (in Line 4) in parallel.

Sequential consistency (from synchronous training) and eventual consistency (from asynchronous training) are typically used for distributed deep learning. Both approaches have scalability issues (Wang et al., 2015). Recently, there are studies for training convex models (deep learning models are non-linear and non-convex) using a value bounded consistency model (Wei et al., 2015). Researchers are starting to investigate the influence of consistency models on distributed training (Gupta, Zhang, and Milthorpe, 2015; Hadjis et al., 2016; Chen et al., 2016). There remains much research to be done on how to provide flexible consistency models for distributed training, and how each consistency model affects the scalability of the system, including communication overhead.

3.2.3 Fault Tolerance and Adaptiveness

Current deep learning systems recover the training from crashes mainly based on checkpointing files (Abadi et al., 2015). However, frequent checkpointing would incur vast overhead. The SGD algorithm used by deep learning training systems can tolerate a certain degree of inconsistency. The way to exploit the SGD properties and system architectures to implement fault tolerance efficiently remains an interesting problem. Considering that distributed training would replicate the model status, it is thus possible to recover from a replica instead of checkpointing files.

There is a trend to train large deep learning models on cloud platforms due to the high demand on hardware resources. For such cases, it is necessary to make the system adaptive to the available resources. Existing system simply assume that the training environment (including the number of workers and specs of each node) would remain the same throughout the whole training procedure, which is in fact not true for cloud platforms, e.g., Amazon EC2 Spot Instances.

3.3 Optimization Techniques Used in Existing Systems

There are some open source deep learning systems under active development. A summary of these systems in terms of the above mentioned optimization aspects is listed in Table 3.1. Many researchers have done ad hoc optimization using Caffe (Jia et al., 2014a), including memory swapping and communication optimization. However, the official version is not well optimized. Similarly, Torch (Collobert, Kavukcuoglu, and Farabet, 2011) itself provides limited support for distributed training. Mxnet (Chen et al., 2015) has optimization for both memory and operations scheduling. Theano (Bastien et al., 2012) is typically used for stand-alone training. TensorFlow (Abadi et al., 2015) has the potential for the aforementioned static optimization based on the dataflow graph.

In this dissertation (Chapter 4), we will propose a distributed training system with a flexible architecture for different parallelism frameworks. Other optimization techniques will be explored as our future work.

Table 3.1: Summary of optimization techniques used in existing systems as of July 2016.

	Caffe	Mxnet	TensorFlow	Theano	Torch	SINGA
1. operation scheduling	x	✓	-	-	x	x
2. memory management	i	d+s	p	p	-	p
3. parallelism	d	d + m	d + m	-	d + m	d+m
4. consistency	s/a	s+a+h	s+a+h	-	s	s+a+h

-: unknown

1. x: not available: ✓: available

2. d: dynamic; a: swap; p: memory pool; i: in-place operation; s: static;

3. d: data parallelism; m: model parallelism;

4. s: synchronous; a: asynchronous; h:hybrid;

3.4 Summary

In this chapter, we investigated a wide range of optimization techniques for deep learning systems for both stand-alone training and distributed training. Some of these techniques are exploited for developing our SINGA system (Chapter 4). The analysis is included in the following vision paper, which discusses the challenges and opportunities of optimizing deep learning systems from databases perspective, and the database applications that may benefit from deep learning models.

- *Wei Wang, Meihui Zhang, Gang Chen, H.V. Jagadish, Beng Chin Ooi, Kian-Lee Tan:Database Meets Deep Learning: Challenges and Opportunities. ACM SIGMOD Record, Volume 45 Issue 2, Pages 17-22, 2016.*

Chapter 4

SINGA: A Distributed Deep Learning System

In this chapter, we present the SINGA system for training large deep learning models on big datasets via distributed computing.

4.1 Introduction

There are two challenges in bringing deep learning to wide adoption in real-life applications. The first challenge is *usability*, meaning the implementation of different models and training algorithms must be done by non-experts with little effort. The user must be able to choose among many existing deep learning models, as different multimedia applications may benefit from different models. For instance, the deep convolutional neural network (DCNN) is suitable for image classification (Krizhevsky, Sutskever, and Hinton, 2012), recurrent neural network (RNN) for language modelling (Mikolov et al., 2011), and deep auto-encoders for multi-modal data analysis (Wang et al., 2014; Feng, Wang, and Li, 2014; Zhang et al., 2014). Furthermore, the user must not be required to implement most of these models and training algorithms from scratch, for they are too complex and costly. An example of complex models is the GoogleLeNet (Szegedy et al., 2014) which comprises 22 layers of 10 different types. Training algorithms are intricate in details. For instance the Back-Propagation (LeCun et al., 1996) algorithm is notoriously difficult to debug.

The second challenge is *scalability*, that is the deep learning system must be able to make provision for a huge demand of computing resources for training large models with massive datasets. As larger training datasets and bigger models are being used to improve accuracy (Ciresan et al., 2010; Le et al., 2012; Szegedy et al., 2014), memory requirement for training the model may easily exceed the capacity of a single CPU or GPU. In addition, the computational cost of training may be too high for a single commodity server, which results in unreasonably long training time. For instance, it takes 10 days (Yadan et al., 2013; Paine et al., 2013) to train the DCNN (Krizhevsky, Sutskever, and Hinton, 2012) with 1.2 million training images and 60 million parameters using one GPU [1].

Addressing both usability and scalability challenges requires a distributed training platform that supports various deep learning models, that comes with an intuitive programming model, and that is scalable. General distributed platforms such as MapReduce and Spark achieve good scalability, but they are designed for general data processing. As a result, they lack both the programming model and system optimization specific to deep learning, hindering the overall usability and scalability. There are several specialized distributed platforms (Dean et al., 2012a; Coates et al., 2013; Chilimbi et al., 2014) that exploit deep learning specific optimization and hence are able to achieve high training throughput. However, they forgo usability issues: the platforms are closed-source and no details of their programming models are given, rendering them unusable by multimedia users. There are a couple of distributed deep learning systems under active development, including Caffe (Jia et al., 2014a), TensorFlow (Abadi et al., 2015), CNTK (Yu et al., 2014) and MxNet (Chen et al., 2015).

In this chapter, we present our effort in bringing deep learning to the masses by proposing a distributed training system called SINGA. SINGA provides a simple, intuitive programming model that makes it accessible even to non-experts.

[1]According to the authors, with 2 GPUs, the training still took about 6 days.

SINGA's simplicity is driven by the observation that both the structures and training algorithms of deep learning models can be expressed using a simple abstraction: the neuron layer (or layer). In SINGA, the user defines and connects layers to form the neural network model, and the runtime transparently manages other issues pertaining to the distributed training such as partitioning, synchronization and communication. Particularly, the neural network is represented as a dataflow computation graph with each layer being a node. During distributed training, the graph is partitioned and each sub-graph can be trained on CPUs or on GPUs. SINGA's scalability comes from its flexible system architecture and specific optimization. Both synchronous and asynchronous training frameworks are supported with a range of built-in partitioning strategies, which enables users to readily explore and find an optimal training configuration. Optimization techniques, including minimizing data transferring and overlapping computation and communication, are implemented to reduce the communication overhead from distributed training.

In summary, this chapter makes the following contributions:

1. We present a distributed platform called SINGA which offers a simple and intuitive programming model based on the layer abstraction.

2. We describe SINGA's distributed architecture and optimization for reducing the communication overhead in distributed training.

3. We evaluate SINGA's performance by comparing it with other open-source systems. The results show that SINGA is scalable and outperforms other systems in terms of training time.

The rest of this chapter is organized as follows. An overview of SINGA as a platform follows in Section 4.2. The programming model is discussed in Section 4.3. We discuss SINGA architecture and training optimization in Section 4.4. The experimental study is presented in Section 4.5 before we conclude in Section 4.7.

4.2 System Overview

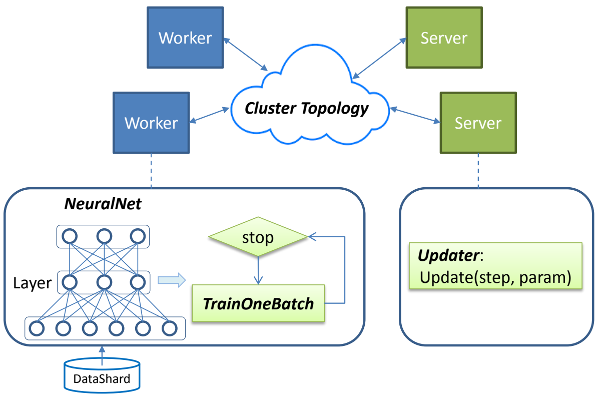


Figure 4.1: SINGA overview.

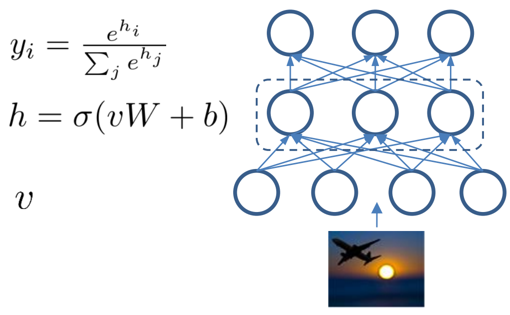
SINGA trains deep learning models using SGD over the worker-server architecture, as shown in Figure 4.1. Workers compute parameter gradients and servers perform parameter updates. To start a training job, the user (or programmer) submits a job configuration specifying the following four components:

- A *NeuralNet* describing the neural network (or neural net) structure with the detailed layers and their connections. SINGA comes with many built-in layers (Section 4.3.1), and users can also implement their own layers for feature transforming or data reading (writing).
- A *TrainOneBatch* algorithm for training the model. SINGA implements different algorithms (Section 4.3.1) for all three popular model categories.
- An *Updater* defining the protocol for updating parameters at the servers.
- A *Cluster Topology* specifying the distributed architecture of workers and servers. SINGA's architecture is flexible to support both different training frameworks, including synchronous and asynchronous training (Section 4.4).

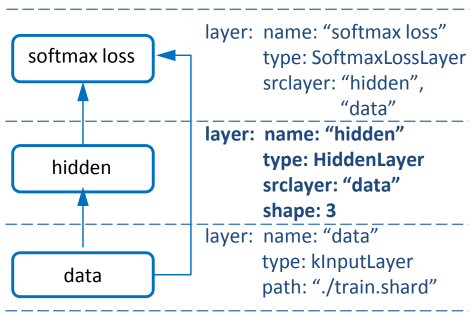
Given a job configuration, SINGA distributes the training tasks over the cluster and coordinates the training. In each iteration, every worker calls *TrainOneBatch* function to compute parameter gradients. *TrainOneBatch* takes a *NeuralNet* object representing the neural net, and visits (part of) the model layers in an order specific to the model category. The computed gradients are sent to the corresponding servers for updating. Workers then fetch the updated parameters at the next iteration.

4.3 Programming Model

This section describes SINGA's programming model. We use a multi-layer perceptron (MLP) model for image classification (Figure 4.2a) as a running example. The model consists of an input layer, a hidden feature transformation layer and a Softmax output layer.



(a) Sample MLP.



```
layer:  name: "softmax loss"
        type: SoftmaxLossLayer
        srclayer: "hidden",
                  "data"
layer:  name: "hidden"
        type: HiddenLayer
        srclayer: "data"
        shape: 3
layer:  name: "data"
        type: kInputLayer
        path: "./train.shard"
```

(b) Net configuration.

```
Blob feature = data[0], gradient=data[1];
Param W, b;

Func  ComputeFeature(flag, srclayers) {
    feature= logistic(dot(srclayers[0].feature, W.data) + b.data);
}
Func  ComputeGradient(flag, srclayers) {
    Blob tmp = feature * (1-feature);
    srclayers[0].gradient = tmp*dot(gradient, W.data.transpose());
    W.gradient = tmp*dot(src[0].data.transpose(), gradient);
    b.gradient = tmp * gradient;
}
```

(c) Hidden layer implementation.

Figure 4.2: Running example using an MLP.

4.3.1 Programming Abstractions

NeuralNet

NeuralNet represents a neural net instance in SINGA. It comprises a set of unidirectionally connected layers. Properties and connections of layers are specified by users. The *NeuralNet* object is passed as an argument to the *TrainOneBatch* function.

Layer connections in *NeuralNet* are not designed explicitly; instead each layer records its own source layers as specified by users (Figure 4.2b). Although different model categories have different types of layer connections, they can be unified using directed edges as follows. For feed-forward models, nothing needs to be done as their connections are already directed. For undirected models, users need to replace each edge with two directed edges, as shown in Figure 4.5. For recurrent models, users can unroll a recurrent layer into directed-connecting sub-layers, as shown in Figure 4.6.

Layer

Layer is a core abstraction in SINGA. Different layer implementations perform different feature transformations to extract high-level features. In every SGD iteration, all layers in the *NeuralNet* are visited by the *TrainOneBatch* function during the process of computing parameter gradients. From the dataflow perspective, we can regard the neural net as a graph where each layer is a node. The training procedure passes data along the connections of layers and invokes functions of layers. Distributed training can be easily conducted by assigning sub-graphs to workers.

Figure 4.3 shows the definition of a base layer. The *data* field records data (blob) associated with a layer. Some layers may require parameters (e.g., a weight matrix) for their feature transformation functions. In this case, these parameters are represented by *Param* objects, each with a *data* field for the parameter

```
Layer:
  vector<Blob> data
  vector<Param> param
  Func ComputeFeature(flag, srclayers);
  Func ComputeGradient(flag, srclayers);

Param:
  Blob data, gradient;
```

Figure 4.3: Layer abstraction.

values and a *gradient* field for the gradients. The *ComputeFeature* function evaluates the feature blob by transforming features from the source layers. The *ComputeGradient* function computes the gradients associated with this layer. These two functions are invoked by the *TrainOneBatch* function during training (Section 4.3.1).

SINGA provides a variety of built-in layers to help users build their models. Table 4.1 lists the layer categories in SINGA. For example, the data layer loads a mini-batch of records via the *ComputeFeature* function in each iteration. Users can also define their own layers for their specific requirements. Figure 4.2c shows an example of implementing the hidden layer h in the MLP. In this example, besides feature blobs there are gradient blobs storing the gradients of the loss with respect to the feature blobs. There are two *Param* objects: the weight matrix W and the bias vector b. The *ComputeFeature* function rotates (multiply W), shifts (plus b) the input features and then applies non-linear (logistic) transformations. The *ComputeGradient* function computes the layer's parameter gradients, as well as the source layer's gradients that will be used for evaluating the source layer's parameter gradients.

TrainOneBatch

The *TrainOneBatch* function determines the sequence of invoking *ComputeFeature* and *ComputeGradient* functions in all layers during each SGD iteration. SINGA

Table 4.1: Layer categories.

Category	**Description**
Input layers	Load records from file, database or HDFS.
Output layers	Dump records to file, database or HDFS.
Neuron layers	Feature transformation, e.g., convolution.
Loss layers	Compute objective loss, e.g., cross-entropy loss.
Connection layers	Connect layers when neural net is partitioned.

Algorithm 4.1: BPTrainOneBatch

```
foreach layer in net.layers do
    Collect(layer.params()) // receive parameters
    layer.ComputeFeature() // forward prop
end
foreach layer in reverse(net.layers) do
    layer.ComputeGradient() //backward prop
    Update(layer.params()) // send gradients
end
```

implements two *TrainOneBatch* algorithms for the three model categories. For feed-forward and recurrent models, the BP algorithm is provided. For undirected modes (e.g., RBM), the CD algorithm is provided. Users simply select the corresponding algorithm in the job configuration. Should there be specific requirements for the training workflow, users can define their own *TrainOneBatch* function following a template shown in Algorithm 4.1. Algorithm 4.1 implements the BP algorithm which takes a *NeuralNet* object as input. The first loop visits each layer and computes their features, and the second loop visits each layer in the reverse order and computes parameter gradients.

Updater

Once the parameter gradients are computed, workers send these values to servers to update the parameters. SINGA implements several parameter updating protocols, such as AdaGrad(Duchi, Hazan, and Singer, 2011). Users can also define their own updating protocols by overriding the *Update* function.

4.3.2 Examples

This section demonstrates the use of SINGA for example applications. We discuss the training of three deep learning models for three different applications: a multi-modal deep neural network (MDNN) for multi-modal retrieval, a RBM for dimensionality reduction, and a RNN for sequence modelling.

MDNN for Multi-modal Retrieval

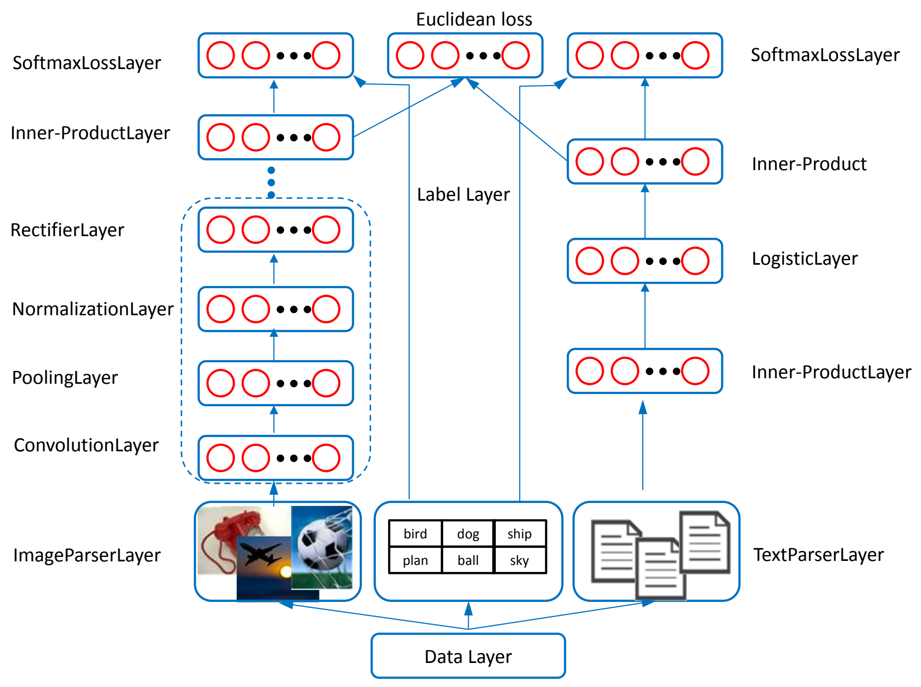


Figure 4.4: Structure of MDNN.

Feed-forward models such as CNN and MLP are widely used to learn high-level features in multimedia applications, especially for image classification (Krizhevsky, Sutskever, and Hinton, 2012). Here, we demonstrate the training of the MDNN (Wang et al., 2015) using SINGA to extract features for the multi-modal retrieval task (Wang et al., 2014; Feng, Wang, and Li, 2014; Shen, Ooi, and Tan, 2000) that searches objects from different modalities. The details of MDNN will be provided in Chapter 5. Generally, in MDNN, there is a CNN (Krizhevsky, Sutskever, and Hinton, 2012) for extracting image features, and a MLP for extracting text features. The training objective is to minimize a weighted sum of: (1) the error of predicting the labels of image and text documents using extracted features; and (2) the distance between features of relevant image and text objects.

Figure 4.4 depicts the neural net of the MDNN model in SINGA. We can see that there are two parallel paths: one for text modality and the other for image modality. The data layer reads in records of semantically relevant image-text pairs. The image layer, text layer and label layer parse the visual feature, text feature (e.g., tags of the image) and labels respectively from the records. The image path consists of layers from DCNN (Krizhevsky, Sutskever, and Hinton, 2012), e.g., the convolutional layer and pooling layer. The text path includes an inner-product (or fully connected) layer, a logistic layer and a loss layer. The Euclidean loss layer measures the distance of the feature vectors extracted from these two paths. All except the parser layers, which are application specific, are SINGA's built-in layers. Since this model is a feed-forward model, the BP algorithm is selected for the *TrainOneBatch* function.

RBM for Dimensionality Reduction

RBM is often employed to pre-train parameters for other models. In this example application, we use RBM to pre-train deep auto-encoders (Hinton and Salakhutdinov, 2006) for dimensionality reduction. Dimensionality reduction techniques, such as Principal Component Analysis (PCA), are commonly applied in the pre-processing step of data analytic applications. Deep auto-encoders are reported (Hinton and Salakhutdinov, 2006) to have better performance than

PCA.

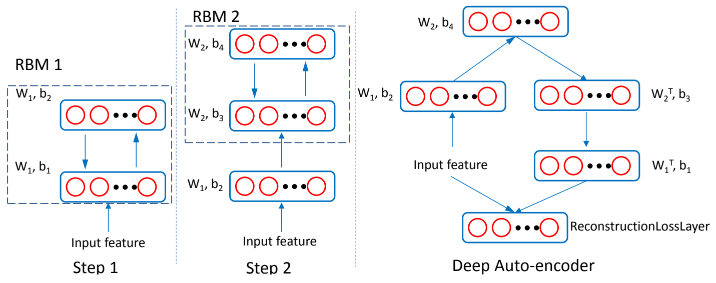


Figure 4.5: Structure of RBM and deep auto-encoders.

Generally, the deep auto-encoders are trained to reconstruct the input feature using the feature of the top layer. Hinton et al. (Hinton and Salakhutdinov, 2006) used RBM to pre-train the parameters for each layer, and fine-tuned them to minimize the reconstruction error. Figure 4.5 shows the model structure (with parser layer and data layer omitted) in SINGA. The parameters trained from the first RBM (RBM 1) in step 1 are ported (through checkpoint) into step 2 wherein the extracted features are used to train the next model (RBM 2). Once pre-training is finished, the deep auto-encoders are unfolded for fine-tuning. SINGA applies the contrastive divergence (CD) algorithm for training RBM and back-propagation (BP) algorithm for fine-tuning the deep auto-encoder.

RNN for Sequence Modelling

Recurrent neural networks (RNN) are widely used for modelling sequential data, e.g., natural language sentences. We use SINGA to train a Char-RNN model [2] over Linux kernel source code, with each character as an input unit. The model predicts the next character given the current character.

Figure 4.6 illustrates the net structure of the Char-RNN model. In each iteration, the input layer reads $unroll\_len + 1$ ($unroll\_len$ is specified by users) successive characters, e.g., "int a;" and passes the first $unroll\_len$ characters to OneHot-Layers (one per layer), and passes the last $unroll\_len$ characters as labels to

[2] https://github.com/karpathy/char-rnn

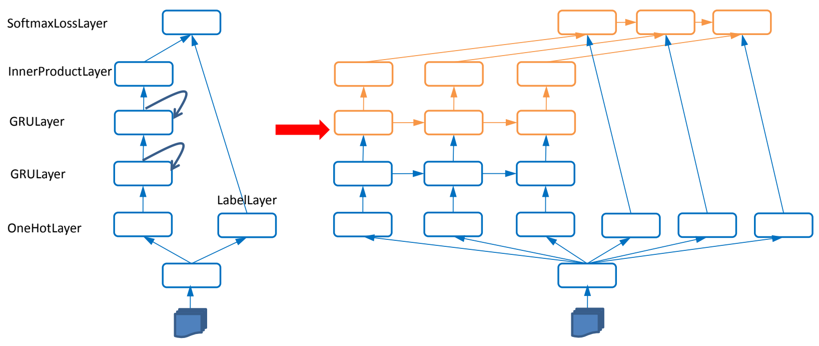


Figure 4.6: Structure of 2-stacked Char-RNN (left before unrolling; right after unrolling).

the label layer. The label of the i^{th} character is the $(i+1)^{th}$ character. In other words, the objective is to predict the next character. The model is configured similarly as for feed-forward models except the training algorithm is BPTT, and unrolling length and connection types are specified for recurrent layers. Different colors are used for illustrating the neural net partitioning which will be discussed in Section 4.4.3.

4.4 Distributed Training

In this section, we introduce SINGA's architecture, and discuss how it supports a variety of distributed training frameworks.

4.4.1 System Architecture

Figure 4.7 shows the logical architecture, which consists of multiple server groups and worker groups, and each worker group communicates with only one server group. Each server group maintains a complete replica of the model parameters, and is responsible for handling requests (e.g., get or update parameters) from worker groups. Neighboring server groups synchronize their parameters periodically. Typically, a server group contains a number of servers, and each server manages a partition of the model parameters. Each worker group trains

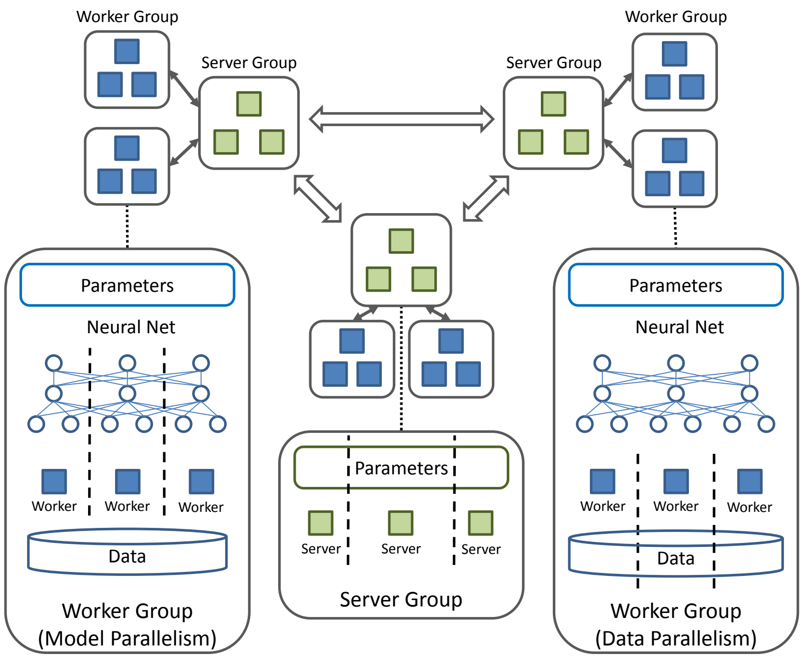


Figure 4.7: Logical architecture of SINGA.

a complete model replica against a partition of the training dataset (i.e. data parallelism), and is responsible for computing parameter gradients. All worker groups run and communicate with the corresponding server groups asynchronously. However, inside each worker group, the workers compute parameter updates synchronously for the model replica. There are two strategies to distribute the training workload among workers within a group: by model or by data. More specifically, each worker can compute a subset of parameters against all data partitioned to the group (i.e., model parallelism), or all parameters against a subset of data (i.e., data parallelism). SINGA also supports hybrid parallelism (Section 4.4.3).

In SINGA, servers and workers are execution units running in separate threads. If GPU devices are available, SINGA automatically assigns g GPU devices (g is user specified) to the first g workers on each node. A GPU worker executes the layer functions on GPU if they are implemented using GPU API (e.g., CUDA). Otherwise, the layer functions execute on CPU. SINGA provides several linear algebra functions for users to implement their own layer functions. These linear algebra functions have both GPU and CPU implementation and they determine the running device of the calling thread automatically. In this way, we keep the implementation transparent to users. Workers and servers communicate through message passing. Every process runs the main thread as a stub that aggregates local messages and forwards them to corresponding (remote) receivers. SINGA uses the ZeroMQ library for message passing over the network.

4.4.2 Training Frameworks

In SINGA, worker groups run asynchronously and workers within one group run synchronously. Users can leverage this general design to run both synchronous and asynchronous training frameworks. Specifically, users control the training framework by configuring the cluster topology, i.e., the number of worker (resp. server) groups and worker (resp. server) group size. In the following, we will discuss how to realize popular distributed training frameworks in SINGA, including Sandblaster and Downpour from Google's DistBelief system (Dean et al., 2012a), AllReduce from Baidu's DeepImage system (Wu et al., 2015) and distributed Hogwild from Caffe (Jia et al., 2014a).

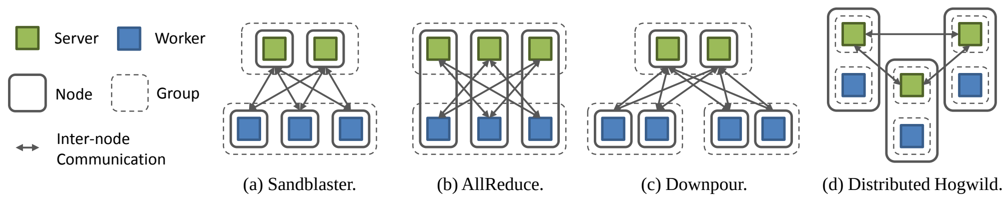


Figure 4.8: Training frameworks in SINGA.

Synchronous Training

A synchronous framework is realized by configuring the cluster topology with only one worker group and one server group. The training convergence rate is the same as that on a single node. Figure 4.8a shows the Sandblaster framework implemented in SINGA. A single server group is configured to handle requests from workers. A worker operates on its partition of the model, and only communicates with servers handling the related parameters. This framework is typically used if high performance dedicated servers with large network bandwidth are available (Chilimbi et al., 2014). Figure 4.8b shows the AllReduce framework in SINGA, in which we bind each worker with a server on the same node, so that each node is responsible for maintaining a partition of parameters and collecting updates from all other nodes. This framework is suitable for single node multi-GPU case or small GPU clusters. For large clusters, the all-to-all connection would incur huge amount of communication cost.

Synchronous training is typically limited to a small or medium size cluster, e.g. fewer than 100 nodes. When the cluster size is large, the synchronization delay and communication overhead is likely to be larger than the computation time. Consequently, the training cannot scale well.

Asynchronous Training

An asynchronous framework is implemented by configuring the cluster topology with more than one worker groups. The training convergence is likely to be different from single-node training, because multiple worker groups are working on different versions of the parameters (Zhang and Re, 2014). Figure 4.8c shows the Downpour (Dean et al., 2012a) framework implemented in SINGA. Similar to the synchronous Sandblaster, all workers send requests to a global server group. We divide workers into several groups, each running independently and working on parameters from the last *update* response. Like Sandblaster, this framework also requires the servers to have large bandwidth to handle requests for multiple worker groups. Figure 4.8d shows the distributed Hogwild framework, in which each node contains a complete server group and a complete worker group. Parameter updates are done locally, so that communication cost during each training step is minimized. However, the server group must periodically synchronize with neighboring groups to improve the training convergence. The topology (connections) of server groups can be customized (the default topology is all-to-all connection). This framework is most widely used for single node Multi-GPU environment, where server groups synchronize via shared memory. For a large cluster, the synchronization among server groups would incur significant overhead and delay.

Asynchronous training can improve the convergence rate to some degree. But the improvement typically diminishes when there are more model replicas because the delay (or staleness) of parameter updates increases. A more scalable training framework should combine both the synchronous and asynchronous training. In SINGA, users can run a hybrid training framework by launching multiple worker groups that run asynchronously to improve the convergence rate. Within each worker group, multiple workers run synchronously to accelerate one training iteration. Given a fixed budget (e.g., number of nodes in a cluster), there are opportunities to find one optimal hybrid training framework that trades off between the convergence rate and efficiency in order to achieve the minimal training time.

4.4.3 Neural Network Partitioning

In this section, we describe how SINGA partitions the neural net to support data parallelism, model parallelism, and hybrid parallelism within one worker group.

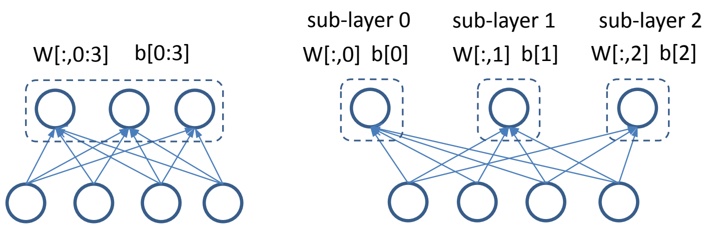


Figure 4.9: Partition the hidden layer in Figure 4.2a.

SINGA partitions a neural net at the granularity of layer. Every layer's feature blob is considered a matrix whose rows are feature vectors. Thus, the layer can be split on two dimensions. Partitioning on dimension 0 (also called batch dimension) slices the feature matrix by row. For instance, if the mini-batch size is 256 and the layer is partitioned into 2 sub-layers, each sub-layer would have 128 feature vectors in its feature blob. Partitioning on this dimension has no effect on the parameters, as every *Param* object is replicated in the sub-layers. Partitioning on dimension 1 (also called feature dimension) slices the feature matrix by column. For example, suppose the original feature vector has 50 units, after partitioning into 2 sub-layers, each sub-layer would have 25 units. This partitioning splits *Param* objects, as shown in Figure 4.9. Both the bias vector and weight matrix are partitioned into two sub-layers (workers).

Network partitioning is conducted while creating the *NeuralNet* instance. SINGA extends a layer into multiple sub-layers. Each sub-layer is assigned a location ID, based on which it is dispatched to the corresponding worker. Advanced users can also directly specify the location ID for each layer to control the placement of layers onto workers. For the MDNN model in Figure 4.4, users can configure the layers in the image path with location ID 0 and the layers in the text path with location ID 1, making the two paths run in parallel. Similarly, for the Char-RNN model shown in Figure 4.6, we can place the layers of different colors onto different workers. Connection layers will be automatically added to connect the sub-layers. For instance, if two connected sub-layers are located at two different workers, then a pair of bridge layers is inserted to transfer the feature (and gradient) blob between them. When two layers are partitioned on different dimensions, a concatenation layer which concatenates feature rows (or columns) and a slice layer which slices feature rows (or columns) are inserted. Connection layers help make the network communication and synchronization transparent to the users.

When every worker computes the gradients of the entire model parameters, we refer this process as data parallelism. When different workers compute the gradients of different parameters, we call this process model parallelism. In particular, partitioning on dimension 0 of each layer results in data parallelism, while partitioning on dimension 1 results in model parallelism. Moreover, SINGA supports hybrid parallelism wherein some workers compute the gradients of the same subset of model parameters while other workers compute on different model parameters. For example, to implement the hybrid parallelism in (Krizhevsky, 2014) for the CNN model, we set *partition_ dim = 0* for lower layers and *partition_ dim = 1* for higher layers. The following list summarizes the partitioning strategies, their trade-off is analyzed in Section 4.4.4.

1. Partitioning all layers into different subsets → model parallelism.
2. Partitioning each single layer into sub-layers on batch dimension → data parallelism.
3. Partitioning each single layer into sub-layers on feature dimension → model parallelism.
4. Hybrid partitioning of strategy 1, 2 and 3 → hybrid parallelism.

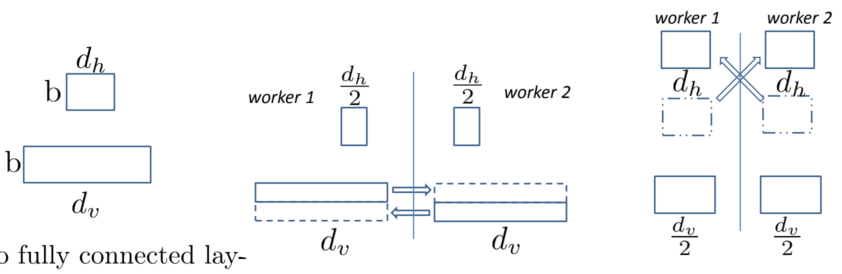
4.4.4 Optimizations of Communication

Distributed training (i.e, partitioning the neural net and running workers over different layer partitions) increases the computation power, i.e., FLOPS. However, it introduces overhead in terms of communication and synchronization. Suppose we have a homogeneous computation environment, that is, all workers run at the same speed and get the same workload (e.g., same number of training samples and same size of feature vectors). In this case, we can ignore the synchronization overhead and analyze only the communication cost. The communication cost is mainly attributed to the data transferred through PCIe over multiple GPUs in a single node, or through the network in a cluster. To cut down the overall overhead, first we try to reduce the amount of data to be transferred. Further more, we try to parallelize the computation and communication, in order to hide the communication time. Here we discuss synchronous training only (i.e., a single worker group), which has the identical theoretical convergence as training in a single worker. Optimization techniques that may affect convergence rate of SGD are not considered, e.g., asynchronous SGD (i.e., multiple worker groups) and parameter compression (Seide et al., 2014). The following analysis works for training either over multiple CPU nodes or over multiple GPU cards on a single node.

Reducing Data Transferring

There are two types of data transferring in distributed training. First, the feature vectors may be transferred as messages if two connected layers are located in different workers, e.g., by model parallelism. Second, the parameter (values and gradients) are transferred for aggregation if there are replicated due to data parallelism. The guideline for reducing data transferring is to do data parallelism for layers with fewer parameters and do model parallelism for layers with smaller feature vectors. To illustrate, we use the popular benchmark model, i.e., AlexNet, as an example. AlexNet is a feed-forward model with single path, the i^{th} layer depends on $(i-1)^{th}$ layer directly. It is not feasible to parallelize subsets of layers as in MDNN, therefore we do not consider the first partitioning strategy. Next, we discuss every type of layer involved in AlexNet one by one.

Convolutional layers contain 5% of the total parameters but 90-95% of the computation, according to AlexNet (Krizhevsky, 2014). It is essential to distribute the computation from these layers. Considering that convolutional layers have large feature vectors and a small amount of parameters, it is natural to apply data parallelism.



(a) Two fully connected layers.

(b) Partition on hidden layer.

(c) Partition on visible layer.

Figure 4.10: Distributed computing for fully connected layers.

Fully connected layers occupy 95% of the total parameters and 5-10% of computation (Krizhevsky, 2014), therefore we should avoid data parallelism and use model parallelism for them. Particularly, with data parallelism, the communication overhead per worker is $O(p)$, where p is the size of the (replicated) parameters. Let b be the effective mini-batch size (summed over all workers), K be the number of workers, and d_v (resp. d_h) be the length of the visible (resp. hidden) feature vector. Figure 4.10b shows the case for data partitioning for the visible layer and model partitioning for the hidden layer, the overhead is $O(b * d_v)$ for exchanging the visible features. Figure 4.10c applies model partitioning for the visible layer, whose overhead comes from exchanging the hidden features, i.e., $O(b*d_h)$. For the first fully connected layer in AlexNet, p is about 177 million while $d_v = d_h = 4096$. In other words, $p > b * d_v$ and $p > b * d_h$, hence data parallelism is costlier than model parallelism.

For pooling layers and local responsive normalization layers, each neuron depends on many neurons from their source layers. Moreover, they are inter-leaved with convolutional layers, thus it is cheaper to apply data parallelism than model parallelism for them. For the remaining layers, they do not have parameters and their neurons depend on source neurons element-wise, hence their partitioning strategies just need to be consistent with their source layers. Consequently, a simple hybrid partitioning strategy for AlexNet (Krizhevsky, 2014) could be applying data parallelism for layers before (or under) the first fully connected layer, and then apply model parallelism or no parallelism for all other layers. Currently, we require users to configure the partitioning strategy for each layer to get the above hybrid partitioning scheme. Automatic optimization and configuration is left as a future work. Reducing data transferring could save power but may not bring speed improvement if the communication cost is hidden due to overlapping with computation as described below.

Overlapping Computation and Communication

Overlapping the computation and communication is another common technique for system optimization. In SINGA, the communication comprises transferring

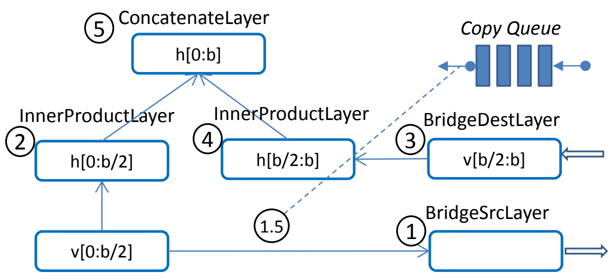


Figure 4.11: Parallelize computation and communication for a GPU worker.

parameter gradients and values, and transferring layer data and gradients. First, for parameter gradients/values, we can send them asynchronously while computing other layers. Take Figure 4.2 as an example, after the hidden layer finishes *ComputeFeature*, we can send the gradients asynchronously to the server for updates while the worker continues to load data for the next iteration. The updated parameters are transferred back to the server by pushing a copy operation into the Copy queue as shown in Figure 4.11, which is checked and executed by the worker. Second, the transferring of layer data/gradients typically comes from model partitioning as discussed in Section 4.4.4. In this case, each worker owns a small subset of data and fetches all rest from other workers. To overlap the computation and communication, each worker can just initiate the communication and then compute over its own data asynchronously. Take the Figure 4.10b as an example, to parallelize the computation and communication, SINGA runs over the layers shown in Figure 4.11 in order. The *BridgeSrcLayer::ComptueFeature* initiates the sending operations and returns immediately. The *BridgeDestLyer::ComputeFeature* waits until data arrives (by checking a signal for the ending of data transferring). All layers are sorted in topology order followed by communication priority.

4.5 Experimental Study

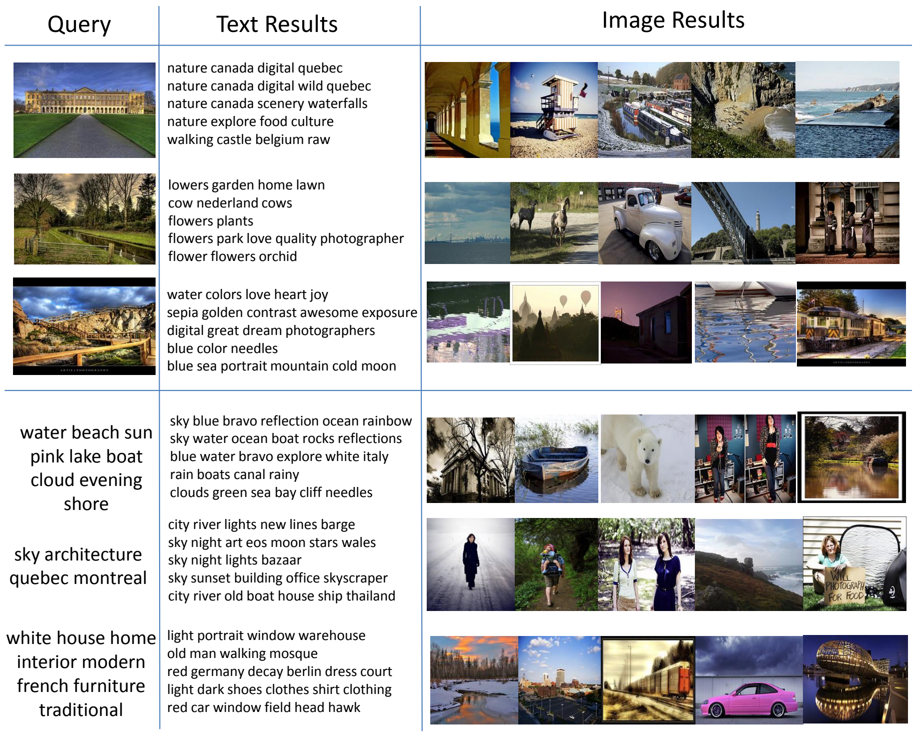
We have developed SINGA using C++ on Linux platforms. OpenBLAS and cuDNN are integrated for accelerating linear algebra and neural net operations. ZeroMQ is used for message passing. This section evaluates SINGA's usability and efficiency. Specifically, we used SINGA to train the models discussed in Section 4.3.2, which required little development effort since SINGA comes with many built-in layers and algorithms. We then compared SINGA with other open-source systems in terms of efficiency and scalability when running on CPUs and GPUs.

4.5.1 Applications of SINGA

We trained models for the example applications in Section 4.3.2 using SINGA. The neural nets were configured using the built-in layers as shown in Figure 4.4, 4.5, 4.6. Users can train these models following the instructions on-line[3].

Multi-modal Retrieval. We trained the MDNN model for multi-modal retrieval application. We used NUS-WIDE dataset (Chua et al., July 8-10, 2009), which has roughly 180,000 images after removing images without tags or from non-popular categories. Each image is associated with several tags. We used Word2Vec (Mikolov et al., 2013) to learn a word embedding for each tag and aggregated the embedding of all the tags from the same image as a text feature. Figure 4.12 shows sample search results. We first used images as queries to retrieve similar images and text documents. It can be seen that image results are more relevant to the queries. For instance, the first image result of the first query is relevant because both images are about architecture, but the text results are not very relevant. This can be attributed to the large semantic gap between different modalities, making it difficult to locate semantically relevant objects in the latent (representation) space.

[3]http://singa.apache.org/docs/examples.html



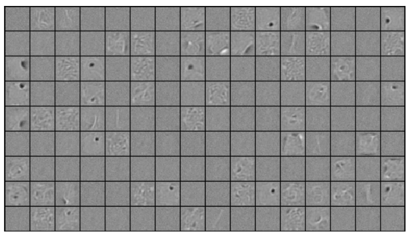
Query	Text Results	Image Results
	nature canada digital quebec nature canada digital wild quebec nature canada scenery waterfalls nature explore food culture walking castle belgium raw	
	lowers garden home lawn cow nederland cows flowers plants flowers park love quality photographer flower flowers orchid	
	water colors love heart joy sepia golden contrast awesome exposure digital great dream photographers blue color needles blue sea portrait mountain cold moon	
water beach sun pink lake boat cloud evening shore	sky blue bravo reflection ocean rainbow sky water ocean boat rocks reflections blue water bravo explore white italy rain boats canal rainy clouds green sea bay cliff needles	
sky architecture quebec montreal	city river lights new lines barge sky night art eos moon stars wales sky night lights bazaar sky sunset building office skyscraper city river old boat house ship thailand	
white house home interior modern french furniture traditional	light portrait window warehouse old man walking mosque red germany decay berlin dress court light dark shoes clothes shirt clothing red car window field head hawk	

Figure 4.12: Multi-Modal Retrieval. Top 5 similar text documents (one line per document) and images are displayed.

Dimensionality Reduction. We trained RBM models to initialize the deep auto-encoder for dimensionality reduction. We used the MNIST[4] dataset consisting of 70,000 images of hand-written digits. Following the configuration used in (Hinton and Salakhutdinov, 2006), we set the size of each layer as 784→1000→500→250→2. Figure 4.13(a) visualizes sample columns of the weight matrix of the bottom (first) RBM. We can see that Gabor-like filters are learned. Figure 4.13(b) depicts the features extracted from the top-layer of the auto-encoder, wherein one point represents one image. Different colors represent different digits. We can see that most images are well clustered according to the ground truth, except for images of digit '4' and '9' (central part) which have some overlap (in practice, handwritten '4' and '9' digits are fairly similar in shape).

Char-RNN We used the Linux kernel source code extracted using an online

[4]http://yann.lecun.com/exdb/mnist/


(a) Bottom RBM weight matrix.

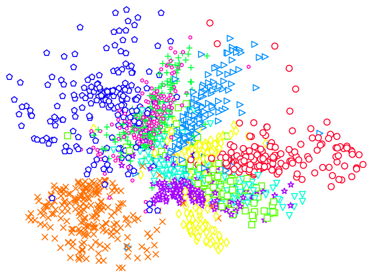
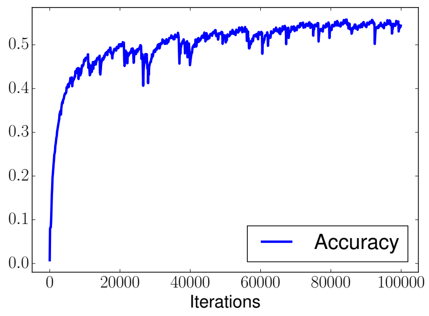
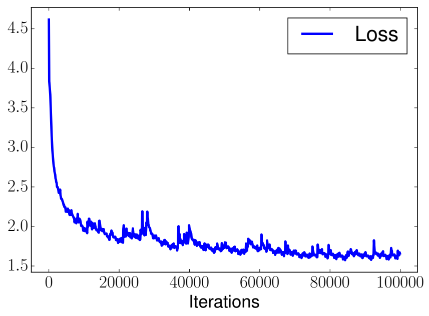

(b) Top layer features.

Figure 4.13: Visualization of the weight matrix in the bottom RBM and top layer features in the deep auto-encoder.

script[5] for this application. The dataset is about 6 MB. The RNN model is configured similar to Figure 4.6. Since this dataset is small, we used one stack of recurrent layers (Figure 4.6 has two stacks). The training loss and accuracy is shown in Figure 4.14. We can see that the Char-RNN model can be trained to predict the next character given previous characters in the source code more and more accurately. There are some fluctuations due to the variance of data samples in different mini-batches (the loss and accuracy are computed per mini-batch).



(a) Training accuracy.



(b) Training loss.

Figure 4.14: Training accuracy and loss of Char-RNN.

4.5.2 Training Performance Evaluation on CPU

We evaluated SINGA's training efficiency and scalability for both synchronous and asynchronous frameworks on a single multi-core node, and on a cluster of

[5] http://cs.stanford.edu/people/karpathy/char-rnn

commodity servers.

Methodologies

The deep convolution neural network[6] for image classification was used as the training model for benchmarking. The training was conducted over the CIFAR10 dataset[7] which has 50,000 training images and 10,000 test images. For the single-node setting, we used a 24-core server with 512GB memory. The 24 cores are distributed into 4 NUMA nodes (Intel Xeon 7540). Hyper-threading is turned on. For the multi-node setting, we used a 32-node cluster. Each cluster node is equipped with a quad-core Intel Xeon 3.1 GHz CPU and 8GB memory. The cluster nodes are connected by a 1Gbps switch.

Synchronous training

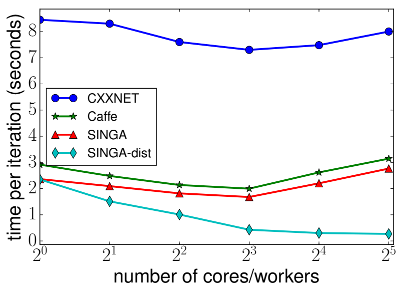
We compared SINGA with CXXNET[8] and Caffe (Jia et al., 2014a). All three systems use OpenBlas to accelerate matrix multiplications. Both CXXNET and Caffe were compiled with their default optimization levels: O3 for the former and O2 for the latter. We observed that because synchronous training has the same convergence rate as that of sequential SGD, all systems would converge after same number of iterations (i.e., mini-batches). This means the difference in total training time among these systems is attributed to the efficiency of a single iteration. Therefore, we only compared the training time for one iteration. We ran 100 iterations for each system and averaged the result time over 50 iterations: 30th to 80th iteration, in order to avoid the effect of starting and ending phases.

On the 24-core single node, we used 256 images per mini-batch and varied the number of OpenBlas's threads. The result is shown in Figure 4.15(a). *SINGA-dist* represents the SINGA configuration in which there are multiple workers, each

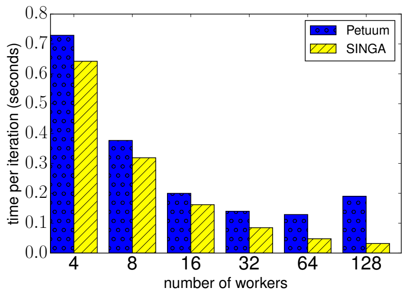
[6]https://code.google.com/p/cuda-convnet/

[7]http://www.cs.toronto.edu/ kriz/cifar.html

[8]https://github.com/dmlc/cxxnet



(a) On the single node.



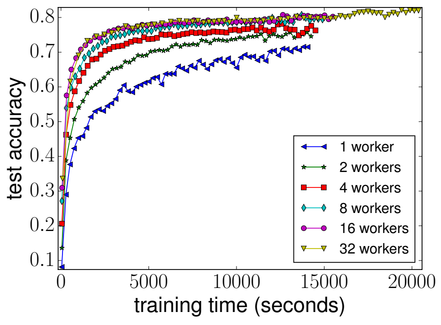
(b) On the 32-node cluster.

Figure 4.15: Synchronous training.

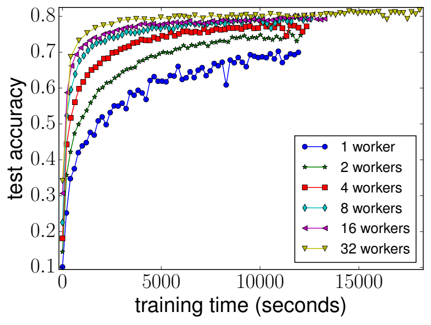
worker has 1 OpenBlas thread[9]. In contrast, *SINGA* represents the configuration which has only 1 worker. We configured SINGA-dist with the cluster topology consisting of one server group with four servers and one worker group with varying number of worker threads (Figure 4.15(a)). In other words, SINGA-dist ran as the in-memory Sandblaster framework. We can see that SINGA-dist has the best overall performance: it is the fastest for each number of threads, and it is also the most scalable. Other systems using multi-threaded OpenBlas scale poorly. This is because OpenBlas has little awareness of the application, and hence it cannot be fully optimized. For example, it may only parallelize specific operations such as large matrix multiplications. In contrast, in SINGA-dist partitions the mini-batch equally between workers and achieves parallelism at the worker level. Another limitation of OpenBlas, as shown in Figure 4.15(a), is that when there were more than 8 threads, the overheads caused by cross-CPU memory access (Tan et al., 2015) started to have negative effect on the overall performance.

On the 32-node cluster, we compared SINGA against another distributed machine learning framework called Petuum (Dai et al., 2013). Petuum runs Caffe as an application to train deep learning models. It implements a parameter server to perform updates from workers (clients), while the workers run synchronously. We used a larger mini-batch size (512 images) and disabled OpenBlas multi-

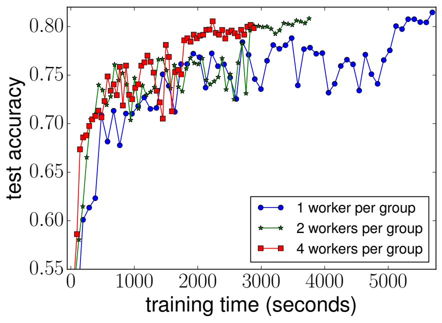
[9]OPENBLAS_NUM_THREADS=1



(a) Caffe on the single node.



(b) SINGA on the single node.



(c)SINGA on the cluster.

Figure 4.16: Asynchronous training.

threading. We configured SINGA's cluster topology to realize the AllReduce framework: there is 1 worker group and 1 server group, and in each node there are 4 workers and 1 server. We varied the size of worker group from 4 to 128, and the server group size from 1 to 32. We note that one drawback of synchronous distributed training is that it cannot scale to too many nodes. This is because the BP (computing) time would be smaller than the time of synchronization and communication, if there are too many workers. In particular, BP time increases with larger batch-size and decreases with larger group size. However, the batch-size are typically not large, e.g., less than 1024. Consequently, the overhead from distributed training would easily become the bottleneck that hurts the scalability. Figure 4.15(b) shows that SINGA achieves almost linear scalability. In contrast, Petuum scales up to 64 workers, but becomes slower when 128 workers are launched. It might be attributed to the communication overheads at the parameter server and the synchronization delays among workers.

Asynchronous training

We compared SINGA against Caffe which has support for in-memory asynchronous training. On the single node, we configured Caffe to use the in-memory Hogwild (Recht et al., 2011) framework, and SINGA to use the in-memory Downpour framework. Their main difference is that parameter updates are done by workers in Caffe and by a single server (thread) in SINGA. Figure 4.16(a) and Figure 4.16(b) show the model accuracy versus training time with varying numbers of worker groups (i.e. model replicas). Every worker processed 16 images per iteration, for a total of 60,000 iterations. We can see that SINGA trains faster than Caffe. Both systems scale well as the number of workers increases, both in terms of the time to reach the same accuracy and of the final converged accuracy. We can also observe that the training takes longer time with more workers. This is due to the increased overhead in context-switching when there are more threads (workers). Finally, we note from the results that the performance difference becomes smaller when the cluster size (i.e., the number of model replicas) reaches 16. This implies that there would be little benefit in having too many model replicas. Thus, we fixed the number of model replicas (i.e., worker groups) to 32 in the following experiments for the distributed asynchronous training.

On the 32-node cluster, we used mini-batch of 16 images per worker group and 60,000 training iterations. We varied the number of workers within one group, and configured the distributed Downpour framework to have 32 worker groups and 32 servers per server group (one server thread per node). We can see from Figure 4.16(c) that with more workers, the training is faster because each worker processes fewer images. However, the training is not as stable as in the single-node setting. This may be caused by the delay (staleness) of parameter synchronization between workers, which is not present in single-node training because parameter updates are immediately visible on the shared memory. The final stage of training (i.e., last few points of each line) is stable because there is only one worker group running during that time. We note that using a warm-up stage, which trains the model using a single worker group at the beginning, may help to stabilize the training as reported in Google's DistBelief system (Dean et al., 2012a).

4.5.3 Training Performance Evaluation on GPU

This section presents the training performance of SINGA running on GPUs. Two optimization techniques would be analyzed at first. After that, SINGA would be compared with other open source systems.

Methodologies

We used the on-line benchmark model from Soumith[10] as the training workload. The model is adapted from the AlexNet (Krizhevsky, 2014) model with some layers omitted. Two sets of hardware are used in our experiments, whose specs and software configurations are shown in Table 4.2. Cudnn V4.0 is used for all experiments.

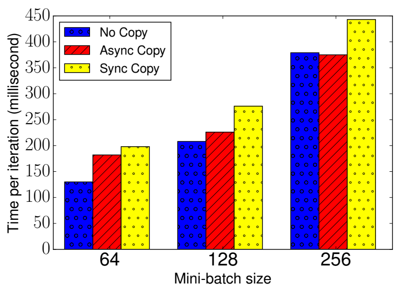
Table 4.2: Specs of hardware and software.

Type	CPU	Memory	GPU	CUDA
Single node	Intel i7-5820K	16 GB	GTX 970 (4GB)	7.0
GPU cluster (4 nodes)	Intel i7-5820K	64 GB	GTX TITAN-X (12GB)	7.5

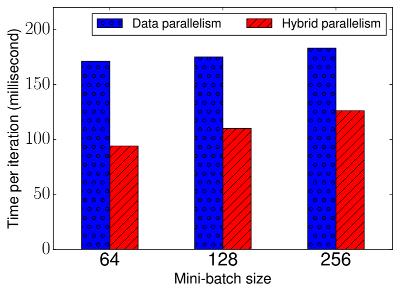
Overlapping Communication and Computation

In Section 4.4.4, we analyzed the optimization technique for hiding the communication overhead by overlapping it with the computation. Here we evaluate the effect of this technique using the single node. Particularly, we compare the efficiency in terms of time per iteration for three versions of SINGA. No Copy

[10]https://github.com/soumith/convnet-benchmarks



(a) Hide communication cost.



(b) Reduce data transferring.

Figure 4.17: Effect of optimization techniques.

version indicates that there is no communication between GPU and CPU, which is widely used for training with a single GPU, where all operations including parameter update are conducted on the single GPU. The other two versions conduct BP algorithm on GPU and parameter updating on CPU, differing only by whether data transferring is done synchronously or asynchronously.

Figure 4.17(a) shows the time per iteration with different mini-batch size. First, we can see that No Copy is the fastest one because it has no communication cost at all. Second, Async Copy is faster than Sync Copy, which suggests that the asynchronous data transferring benefits from the overlapping communication and computation. Moreover, we can see that when the mini-batch increases, the difference between Async Copy and Sync Copy decreases. This is because for large mini-batches, the BP algorithm spends more time doing computation, which increases the overlap area of computation and communication, effectively reducing the overhead. For mini-batch size = 256, Async Copy is even faster than No Copy, this is because Async Copy does not do parameter update, which is done by the server in parallel with BP. However, No Copy has to do BP and parameter updating in sequential.

Reducing Data Transferring

In Section 4.4.4, we discussed how hybrid partitioning is better than other strategies in terms of the overheads in transferring feature vectors between layers in different workers. To demonstrate its effectiveness, we ran SINGA on the single node using two partitioning strategies, i.e., data partitioning and hybrid partitioning for the first fully connected layer in AlexNet. 2 workers are launched (one per GPU). Figure 4.17(b) shows the time per iteration with different mini-batch sizes. We can see that hybrid partitioning has better performance over data partitioning. For data partitioning, only parameter gradients and values are transferred, which is independent of the mini-batch size, thus the time per iteration does not change much when mini-batchs size increases. For hybrid partitioning, when the mini-batch size increases, more feature vectors would be transferred, and then the time increases.

Comparison with Other Systems

We also compared SINGA with four other state-of-the-art deep learning systems namely, Caffe (Jia et al., 2014a), MxNet (Chen et al., 2015), Torch7 (Collobert, Kavukcuoglu, and Farabet, 2011), and TensorFlow (Abadi et al., 2015) (TF). The performance is measured using the throughput, i.e., number of processed images per second. We used (or adapted) the scripts (or instructions) from each system's multi-GPU examples [11] [12] [13] [14]. Better performance could be achieved with further tuning.

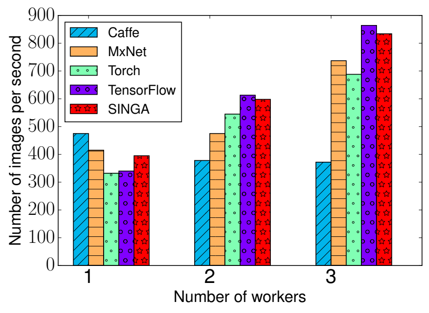
We first compared the throughput of training on a single node with different number of workers (GPUs). We varied the number of workers from 1 to 3, where

[11] Caffe, https://github.com/BVLC/caffe/blob/master/docs/multigpu.md

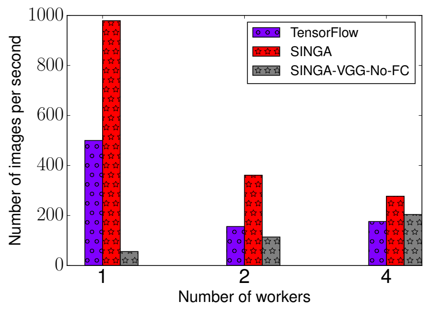
[12] MxNet, https://mxnet.readthedocs.io/en/latest/how_to/multi_devices.html

[13] Torch7, https://github.com/soumith/imagenet-multiGPU.torch

[14] Tensorflow, https://www.tensorflow.org/versions/r0.8/tutorials/deep_cnn/index.html and https://github.com/tensorflow/models/tree/master/inception



(a) A single node with multi-GPUs.



(b) A GPU cluster with 4 nodes.

Figure 4.18: Performance comparison of open source systems.

each worker ran on a GPU. Because Tensorflow ran out of memory with batch size = 128 (the setting used by Soumith's benchmark), we decreased the batch size to 96 for all runnings. The results are shown in Figure 4.18(a).

Caffe has the best single worker performance. This is because there is no parameter transferring between GPU and CPU, whereas others have parameter transferring, e.g., SINGA has to transfer the parameter from the worker (on GPU) to the server (on CPU). However, its performance decreases when more workers are added. This is because 1) the parameters have to be transferred among GPUs (via CPUs), which brings in communication cost; 2)its tree reduction communication pattern (See the link in the footnote) incurs more cost than the all-to-one or all-reduce communication pattern used by other systems when there are more than 2 workers.

For other systems, they have similar performance for the single worker case, as they all use the same cuDNN library for most of the computation. Note that their throughput is lower than Soumith's benchmark because the GPUs are slower than Soumith's and there is communication cost whereas Soumith's benchmark does not involve parameter transferring. Thanks to the optimization techniques introduced in Section 4.4.4, SINGA has almost linear scalability. Tensorflow shows the best scalability among all tested system.

Next, we ran SINGA and Tensorflow in the GPU cluster using the synchronous training framework. We varied the number of nodes (one GPU worker per node) as shown in Figure 4.18(b). For Tensorflow, we launched one parameter server, which communicates with all workers via gRPC. For SINGA, we created a single server group with 1 server, which is in the same process as the first worker and communicates with other workers using ZeroMQ. We reduced the size of the first fully connected layer to 128, because this layer has a big parameter matrix whose size exceeds the limit of the Protobuf message used by Tensorflow. Consequently, the performance for single node (i.e., single GPU) is better than that in Figure 4.18(a). We can see that SINGA performs much better than Tensorflow in terms of throughput. It is likely caused by the network communication, which is not well optimized in Tensorflow. SINGA avoids some communication cost by running the first worker and the parameter server in the same process (they transfer messages via sharing memory). Both systems show poor scalability when there are more than two workers (nodes). On the one hand, node-to-node communication cost and synchronization cost are introduced when there are more than two workers. One the other hand, the Alexnet model has too many parameters that makes the communication the bottleneck. To verify this explanation, we conducted another set of experiments using the VGG model (Simonyan and Zisserman, 2014) with fully connected layers omitted, denoted as VGG-No-FC. This model has 10 convolutional layers and a small amount of parameters, which make it more computation intensive than the Alexnet model. The result in Figure 4.18(b) shows that SINGA has good scalability for this model, which confirms our explanation. To conclude, distributed training is more suitable for models that are computation intensive and with a small amount of parameters.

4.6 Development Progress

The SINGA system is under active development (`singa.apache.org`) with 4 versions, which are described below.

4.6.1 Version 0.1

This version was released in October 2015 with the following major features,

- Programming model based on NeuralNet and Layer abstractions.
- System architecture based on Worker, Server and Stub.
- Training models from three different model categories, namely, feed-forward models, energy models and RNN models.
- Synchronous and asynchronous distributed training frameworks using CPU.

4.6.2 Version 0.2

This version was released in January 2016 with the following major features,

- Training complex models on a single node with multiple GPU cards.
- Hybrid neural net partitioning supports data and model parallelism at the same time.
- Python wrapper for configuring training jobs, including the neural net and the SGD algorithm.
- Cloud software integration includes Mesos, Docker and HDFS.

4.6.3 Version 0.3

This version was released in April 2016, with the following major features,

- Heterogeneous training using CPU and GPU devices.
- Distributed training in a GPU cluster.
- Data prefetching.

4.6.4 Version 1

Version V0.x focused on distributed training, e.g. using multiple GPU cards and a CPU or GPU cluster. A flexible architecture was implemented to run different training frameworks, including synchronous, asynchronous and hybrid training. Version V1.x would provide lower level abstractions than *Layer* and *NeuralNet*. In particular, the *Tensor* and *Device* abstractions are proposed to enable operation level optimizations and improve the extensibility for a wide range of hardware devices and general machine learning models. Therefore, they are the core components of SINGA. Other components like layers and loss functions are built on top of *Tensor* and *Device*, which could be extended and customized by users for specific applications.

```
Device(int device_id);
void SetRandSeed(unsigned seed);

Block* Malloc(size_t num);
void Free(Block *ptr);

void Exec(function<void(Context*)> func, vector<Block*> read,  vector<Block*> write);
// CopyDirection: host2device, device2host, device2device, host2host
void CopyDataToFrom(Block* dst,  Block* src,  size_t num,  CopyDirection direction);
```

Figure 4.19: The Device API.

Device and Tensor

The *Device* abstraction represents a hardware device with multiple execution units. SINGA provides at least three specific devices,

- *CudaGPU* represents an Nvidia GPU card. The execution units are the CUDA streams.
- *CppCPU* represents a normal CPU. The execution units are the CPU threads.
- *OpenclGPU* represents normal GPU card from both Nvidia and AMD. The

execution units are the CommandQueues. Given that OpenCL is compatible with many hardware devices, e.g. FPGA and ARM, the *OpenclGPU* has the potential to be extended for other devices.

The API of *Device* is shown in Figure 4.19. *Device* would schedule all operations and parallelize them onto different execution units. This is done in function *Exec*, where *func* is the real function to be executed on a given executor indicated by *Context*. *read* and *write* includes *Block*s to be read and to be written respectively, which would be analyzed for detecting data dependency. The *Block* abstraction represents a block of device memory, which has a reference counter for garbage collection and is triggered by *Tensor*. Optimizations of operation scheduling (Section 3.1.1) could be implemented in *Exec* by setting the *Context* argument of *func*. *Device* also manages the device memory and transferring data with other devices. The *Malloc* and *Free* functions would do memory optimization, including memory pool, swapping and dropping variables (Section 3.1.2).

```
Tensor(Shape s, Device dev, DataType dtype);

// element-wise addition
Tensor operator+(Tensor lhs, Tensor rhs);
// element-wise multiplication
Tensor EltMult(Tensor lhs, Tensor rhs);
// element-wise multiplication
Tensor operator*(Tensor t, float x);
// matrix-vector or matrix-matrix multiplication
Tensor operator*(Tensor lhs, Tensor rhs);
Tensor Sigmoid*(Tensor t);
// generate random numbers following a Gaussian distribution
void Gaussian(Tensor* t, float mean, float std);
...
```

Figure 4.20: The Tensor API.

Typically, users would not call the methods of *Device*. Instead, they create a device instance and pass it to a tensor instance, which would use this device to allocate memory and execute operations. The *Tensor* abstract represents

Algorithm 4.2: Train a logistic regression model using *Tensor* and *Device*.

```
   Input  : $D = \{< \boldsymbol{x}, \boldsymbol{y} >\}$
   Input  : $\alpha$ // learning rate
   Output : $W, \boldsymbol{b}$ // model parameters
1  CudaGPU dev(0);
2  Tensor W(Shape(m,n), dev, kFloat);
3  Gaussian(&W, 0, 0.01);
4  Tensor $\boldsymbol{b}$(Shape(n), dev, kFloat); // default values are 0s
5  foreach $< \boldsymbol{x}, \boldsymbol{y} >$ in $D$ do
6      $\boldsymbol{p} = Sigmoid(\boldsymbol{x} * W + \boldsymbol{b})$
7      $l = 0.5 * Sum((\boldsymbol{y} - \boldsymbol{p}) * (\boldsymbol{y} - \boldsymbol{p}))$
8      $\boldsymbol{g}$ =EltMult(EltMult($\boldsymbol{p}, \boldsymbol{1} - \boldsymbol{p}), \boldsymbol{p} - \boldsymbol{y}$)
9      $W' = \boldsymbol{x} * \boldsymbol{g}^{\boldsymbol{T}}$
10     $W = W - \alpha * W'$
11     $\boldsymbol{b} = \boldsymbol{b} - \alpha * \boldsymbol{g}$
12 end
```

a multi-dimensional array. A set of linear algebra and random operations are provided against *Tensor*, as shown in Figure 4.20. *Tensor* has another field for data type, which could be float, int, char and float16, etc. float16 saves the memory by half compared with float which uses 4 Bytes per number. For each type of *Device* and each data type, there is a set of corresponding tensor operations. The Tensor functions would automatically detect its device and data type and submit the correct tensor operations to its device instance for execution.

Most machine learning algorithms could be expressed using (dense or sparse) tensors. Therefore, with the *Tensor* abstraction, SINGA would be able to run a wide range of models, including deep learning models and other traditional machine learning models. For example, users could train a logistic regression model as shown in Algorithm 4.2.

Layer and NeuralNet

The implementation of *Layer* and *NeuralNet* is replaced with *Tensor* and *Device* abstractions. The API of *Layer* is shown in Figure 4.21. For each of the three popular categories of deep learning models, one specific *NeuralNet* subclass would be provided. For instance, Figure 4.22 includes the functions of feed-forward neural nets for the single input case. Algorithm 4.3 shows the logistic regression model constructed using built-in layers and the *FeedForwardNet*. Users can build other models, e.g. CNN+RNN used in image caption generation (Mao et al., 2014), using the built-in layers or *Tensor* and *Device* directly.

```
// forward propagation with a single input
Tensor Forward(int flag, Tensor x);
// backward propagation with a single input .
// returned tensors include the gradient of x and parameters.
pair<Tensor, vector<Tensor>> Backward(int flag, Tensor g);

// forward propagation with multiple inputs
Tensor Forward(int flag, vector<Tensor> x);
// backward propagation with multiple inputs
// returned tensors include the gradient of all x and parameters.
pair<vector<Tensor>, vector<Tensor>> Backward(int flag, vector<Tensor> g);
```

Figure 4.21: The Layer API.

```
// constructor
FeedForwardNet(Optimzier opt, Loss loss, Metric metric);
// train on the given batch data <x, y>
Tensor TrainOnBatch(Tensor x, Tensor y);
// evaluate on the given batch data <x, y>
Tensor EvaluateOnBatch(Tensor x, Tensor y);
// return the output of the top layer
Tensor Forward(Tensor x);
// return the gradients of all parameters
vector<Tensor> Backward(Tensor g);
```

Figure 4.22: The FeedForwardNet API.

Algorithm 4.3: Train a logistic regression model using *Layer* and *NeuralNet*.

```
Input  : $D = \{< \boldsymbol{x}, \boldsymbol{y} >\}$
Input  : $\alpha$ // learning rate
Output : $W, \boldsymbol{b}$
1 CudaGPU dev(0);
2 net=FeedForwardNet(new EuclideanLoss(), new SGD($\alpha$));
3 net.Add(InnerProductLayer(n));
4 net.Add(SigmoidLayer());
5 foreach $< \boldsymbol{x}, \boldsymbol{y} >$ in $D$ do
6     net.TrainOnBatch($\boldsymbol{x}, \boldsymbol{y}$)
7 end
```

Distributed Training

SINGA V1.x would be used by users as a library for stand-alone training. For distributed training, the training frameworks from Section 4.4.2 would still be supported using the architecture proposed in Section 4.4. In addition, optimizations in terms of communication, fault-tolerance would be studied and implemented.

Status and Schedule of Version 1.x

SINGA V1.x is under development. The status and schedule is listed below (updated in January 2017)[15],

- V1.0 (September 2016) Added Tensor and Device abstractions and reimplemented the layer, neural net classes, etc.
- V1.1 (January 2017) Improved the usability with flexible installation approaches and more documentation (examples).
- V1.2 (May 2017) Migrate and improve the distributed training components

[15]The latest schedule is at `http://singa.apache.org/en/develop/schedule.html`

from V0.3 including communication and consistency optimization.

- V1.3 (September 2017) Optimize memory usage and operation execution.

4.7 Summary

In this chapter, we introduced a distributed deep learning platform, called SINGA. SINGA offers a simple and intuitive programming model, making it accessible to even non-experts. SINGA is extensible and able to support a wide range of applications using different deep learning models. The flexible training architecture gives the user the chance to balance the trade-off between the training efficiency and convergence rate. We demonstrated the use of SINGA for representative applications, and showed that the platform is both usable and scalable. The future work includes optimization from the system perspective and theory perspective. In particular, convergence analysis of different consistency models is necessary to guide users choose the optimal framework from all possible frameworks for their cluster and model. The SINGA system is published in the following papers,

- *Wei Wang, Gang Chen, Tien Tuan Anh Dinh, Jinyang Gao, Beng Chin Ooi, Kian-Lee Tan, Sheng Wang. SINGA: Putting Deep Learning in the Hands of Multimedia Users. ACM Multimedia, p25-34, 2015*

- *Beng Chin Ooi, Kian-Lee Tan, Sheng Wang, Wei Wang, Qingchao Cai, Gang Chen, Jinyang Gao, Zhaojing Luo, Anthony K. H. Tung, Yuan Wang, Zhongle Xie, Meihui Zhang, Kaiping Zheng. SINGA: A Distributed Deep Learning Platform. ACM Multimedia, p685-688, 2015*

- *Wei Wang, Gang Chen, Haibo Chen, Tien Tuan Anh Dinh, Jinyang Gao, Beng Chin Ooi, Kian-Lee Tan, Sheng Wang. Deep Learning At Scale and At Ease. ACM Transactions on Multimedia Computing Communications and Applications(TOMM) - Special Section on Best Papers of ACM Multimedia 2015, Volume 12 Issue 4s, November 2016*

Chapter 5

Deep Learning based Approaches for Multi-modal Retrieval

The SINGA system introduced in Chapter 4 is general to train deep learning models for various applications including multi-modal retrieval. This chapter will present the multi-modal retrieval application in details and introduce our approaches using feed-forward models.

5.1 Introduction

The prevalence of social networking has significantly increased the volume and velocity of information shared on the Internet. A tremendous amount of data in various media types is being generated every day in social networking systems. These data, together with other domain specific data, such as medical data, surveillance and sensory data, are big data that can be exploited for insights and contextual observations. However, effective retrieval of such huge amounts of media from heterogeneous sources remains a big challenge.

This chapter will present new approaches based on deep learning techniques to solve the problem of large-scale information retrieval from multiple modalities. Each modality represents one type of media such as text, image or video. Depending on the heterogeneity of data sources, there are two types of searches:

1. **Intra-modal search** has been extensively studied and widely used in commercial systems. Examples include web document retrieval via keyword

queries and content-based image retrieval.

2. **Cross-modal search** enables users to explore relevant resources from different modalities. For example, a user can use a tweet to retrieve relevant photos and videos from other heterogeneous data sources. Meanwhile he can search relevant textual descriptions or videos by submitting an interesting image as a query.

There has been a long stream of research on multi-modal retrieval (Bronstein et al., 2010; Zhu et al., 2013; Song et al., 2013; Kumar and Udupa, 2011; Zhen and Yeung, 2012; Lu et al., 2013). These works followed the same query processing strategy, which consists of two major steps. First, a set of mapping functions are learned to project data from different modalities into a common latent space. Second, a multi-dimensional index for each modality in the common space is built for efficient similarity retrieval. Since the second step , known as the classic kNN problem, was extensively studied (Hjaltason and Samet, 2003; Weber, Schek, and Blott, 1998; Zhang et al., 2011), we focused on the optimization of the first step and proposed two types of novel mapping functions based on deep learning techniques.

We proposed a general learning objective that could effectively capture both intra-modal and inter-modal semantic relationships of data from heterogeneous sources. In particular, we differentiated modalities in terms of their representations' ability to capture semantic information and robustness in terms of noisy data. The modalities with better representations were assigned with higher weight for the sake of learning more effective mapping functions. Based on the objective function, we designed an unsupervised algorithm using stacked auto-encoders (SAEs). SAE is a deep learning model that has been widely applied in many unsupervised feature learning and classification tasks (Rifai et al., 2011; Vincent et al., 2008; Goroshin and LeCun, 2013; Socher et al., 2011). For media with semantic labels, we designed a supervised algorithm to realize the learning objective. The supervised approach uses a deep convolutional neural network (DCNN) and neural language model (NLM). It exploits the label information, thus can learn robust mapping functions against noisy input data. DCNN and NLM have shown great success in learning image features (Krizhevsky, Sutskever, and Hinton, 2012; Girshick et al., 2014; Donahue et al., 2013) and text features (Socher and Manning, 2013; Mikolov et al., 2013) respectively.

Compared with existing solutions for multi-modal retrieval, our approaches exhibit three major advantages. First, our mapping functions are non-linear and are more expressive than the linear projections used in IMH (Song et al., 2013) and CVH (Kumar and Udupa, 2011). The deep structures of our models can capture more abstract concepts at higher layers, which is very useful in modeling categorical information of data for effective retrieval. Second, our approaches require minimum prior knowledge in the training. Our unsupervised approach only needs relevant data pairs from different modalities as the training input. The supervised approach requires additional labels for the media objects. In contrast, MLBE (Zhen and Yeung, 2012) and IMH (Song et al., 2013) require a big similarity matrix of intra-modal data for each modality. LSCMR (Lu et al., 2013) uses training examples, each of which consists of a list of objects ranked according to their relevance (based on manual labels) to the first object. Third, our training process is memory efficient because it splits the training dataset into mini-batches and iteratively loads and trains each mini-batch in memory. However, many existing works (e.g., CVH, IMH) have to load the whole training dataset into memory which is infeasible when the training dataset is too large.

In summary, the main contributions of this chapter include:

- a general learning objective for learning mapping functions to project data from different modalities into a common latent space for multi-modal retrieval.
- one unsupervised approach and one supervised approach to implement the general learning objective using deep learning techniques.
- extensive experiments on three real datasets to evaluate the proposed

mapping mechanisms. Experimental results showed that the performance of our method was superior to state-of-the-art methods.

Using this application, we also verified that SINGA is able to handle complex deep learning models. The remainder of the chapter is organized as follows. The problem statement is provided in Section 5.2, followed by the general training objective in Section 5.3. After that, Section 5.4 and Section 5.5 describe the unsupervised and supervised approaches respectively. Query processing is presented in Section 5.6 followed by the experimental study in Section 5.7. Section 5.8 concludes this chapter.

5.2 Preliminary

In our data model, the database $\mathbb{D}$ consists of objects from multiple modalities. For ease of presentation, we use images and text as two sample modalities to explain our idea. In other words, we assume that $\mathbb{D} = \mathbb{D}_I \bigcup \mathbb{D}_T$. To conduct multi-modal retrieval, we need a relevance measurement for the query and the database object. However, the database consists of objects from different modalities, there is no such widely accepted measurement. A common approach is to learn a set of mapping functions that project the original feature vectors into a common latent space such that semantically relevant objects (e.g., image and its tags) are located close. Consequently, our problem includes the following two sub-problems.

Definition 1. *Common Latent Space Mapping*
Given an image $x \in \mathbb{D}_I$ and a text document $y \in \mathbb{D}_T$, find two mapping functions $f_I : \mathbb{D}_I \to \mathbb{Z}$ and $f_T : \mathbb{D}_T \to \mathbb{Z}$ such that if x and y are semantically relevant, the distance between $f_I(x)$ and $f_T(y)$ in the common latent space $\mathbb{Z}$, denoted by $dist_{\mathbb{Z}}(f_I(x), f_T(y))$, is small.

The common latent space mapping provides a unified approach to measuring distance of objects from different modalities. As long as all objects can be mapped into the same latent space, they become comparable. Once the mapping functions

f_I and f_T have been determined, the multi-modal search can then be transformed into the classic kNN problem, defined as following:

Definition 2. *Multi-Modal Search*

Given a query object $Q \in \mathbb{D}_q$ and a target domain $\mathbb{D}_t$ $(q, t \in \{I, T\})$, find a set $O \subset \mathbb{D}_t$ with k objects such that $\forall o \in O$ and $o' \in \mathbb{D}_t/O$, $dist_{\mathbb{Z}}(f_q(Q), f_t(o')) \geq dist_{\mathbb{Z}}(f_q(Q), f_t(o))$.

Since both q and t have two choices, four types of queries can be derived, namely $\mathbb{Q}_{q \to t}$ and $q, t \in \{I, T\}$. For instance, $\mathbb{Q}_{I \to T}$ searches relevant text in $\mathbb{D}_T$ given an image from $\mathbb{D}_I$. By mapping objects from different high-dimensional feature spaces into a low-dimensional latent space, queries could be efficiently processed using existing multi-dimensional indexes (Hjaltason and Samet, 2003; Weber, Schek, and Blott, 1998). Our goal is then to learn a set of effective mapping functions which preserve well both intra-modal semantics (i.e., semantic relationships within each modality) and inter-modal semantics (i.e., semantic relationships across modalities) in the latent space. The effectiveness of mapping functions is measured by the accuracy of multi-modal retrieval using latent features.

5.3 General Training Objective

The flowchart of our multi-modal retrieval framework is illustrated in Figure 5.1. It consists of three main steps: 1) offline model training 2) offline indexing 3) online kNN query processing. In step 1, relevant image-text pairs are used as input training data to learn the mapping functions. For example, image-text pairs can be collected from Flickr where the text features are extracted from tags and descriptions for images. If they are associated with additional semantic labels (e.g., categories), a supervised training algorithm would be applied. Otherwise, an unsupervised training approach would be used. After step 1, we would obtain a mapping function $f_m : \mathbb{D}_m \to \mathbb{Z}$ for each modality $m \in \{I, T\}$. In step 2, objects from different modalities are first mapped into the common space $\mathbb{Z}$ by function f_m. With such unified representation, the latent features from the same modality

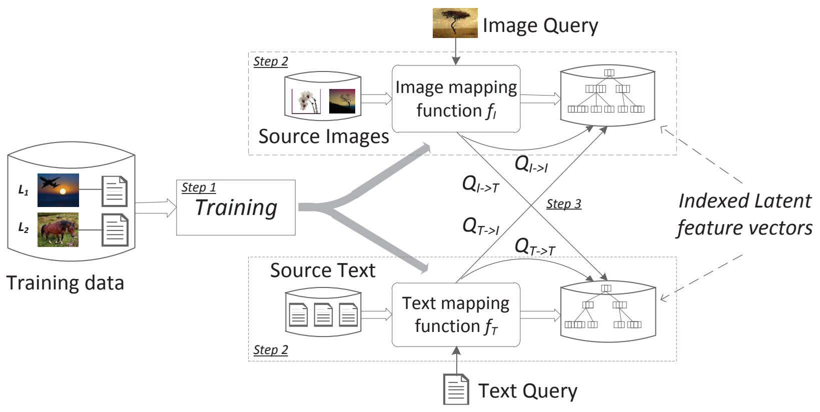


Figure 5.1: Flowchart of multi-modal retrieval framework. Step 1 is offline model training that learns mapping functions. Step 2 is offline indexing that maps source objects into latent features and creates proper indexes. Step 3 is online multi-modal kNN query processing.

are then inserted into a high dimensional index for kNN query processing. When a query $Q \in \mathbb{D}_m$ comes, it is first mapped into $\mathbb{Z}$ using its modal-specific mapping function f_m. Based on the query type, k nearest neighbors are retrieved from the index built for the target modality and returned to the user. For example, image index is used for queries of type $\mathbb{Q}_{I\to I}$ and $\mathbb{Q}_{T\to I}$ against the image database.

General learning objective A good objective function plays a crucial role in learning effective mapping functions. In our multi-modal search framework, we designed a general learning objective function $\mathcal{L}$. By taking into account the image and text modalities, our objective function is defined as follows:

$$\mathcal{L} = \beta_I \mathcal{L}_I + \beta_T \mathcal{L}_T + \mathcal{L}_{I,T} + \xi(\theta) \tag{5.1}$$

where $\mathcal{L}_m$, $m \in \{I, T\}$ is called the intra-modal loss to reflect how well the intra-modal semantics are captured by the latent features. The smaller the loss, the more effective the learned mapping functions are. $\mathcal{L}_{I,T}$ is called the inter-modal loss which is designed to capture inter-modal semantics. The last term is used as regularization to prevent over-fitting (Hinton, 2010) (L_2 Norm is used in our experiment). θ denotes all parameters involved in the mapping functions. β_m,

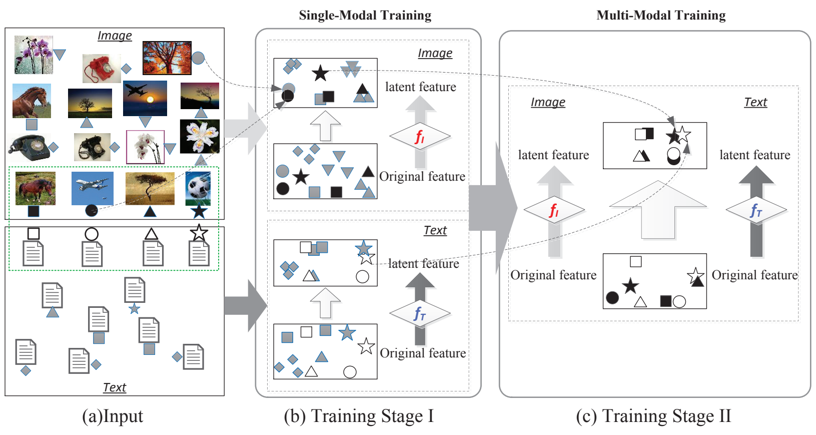


Figure 5.2: Flowchart of training. Relevant images (or text) are associated with the same shape (e.g., ■). In single-modal training, objects of same shape and modality are moving close to each other. In multi-modal training, objects of same shape from all modalities are moving close to each other.

$m \in \{I, T\}$ denotes the weight of the loss for modality m in the objective function. We observed in our training process that assigning different weights to different modalities according to the nature of its data offers better performance than treating them equally. For the modality with lower quality input feature (due to noisy data or poor data representation), we would assign smaller weight for its intra-modal loss in the objective function. The *intuition* of setting β_I and β_T in this way is that, by relaxing the constraints on intra-modal loss, we would enforce the inter-modal constraints. Consequently, the intra-modal semantics of the modality with lower quality input feature could be preserved or even enhanced through their inter-modal relationships with high-quality modalities. Details of setting β_I and β_T is discussed in Section 5.4.2 and Section 5.5.2.

Training Training is to find the optimal parameters involved in the mapping functions that minimizes $\mathcal{L}$. Two types of mapping functions are proposed in this chapter. One is trained by an unsupervised algorithm, which uses simple image-text pairs for training. No other prior knowledge is required. The other one is trained by a supervised algorithm which exploits additional label information to learn robust mapping functions against noisy training data. For both mapping functions, we designed a two-stage training procedure to find the optimal param-

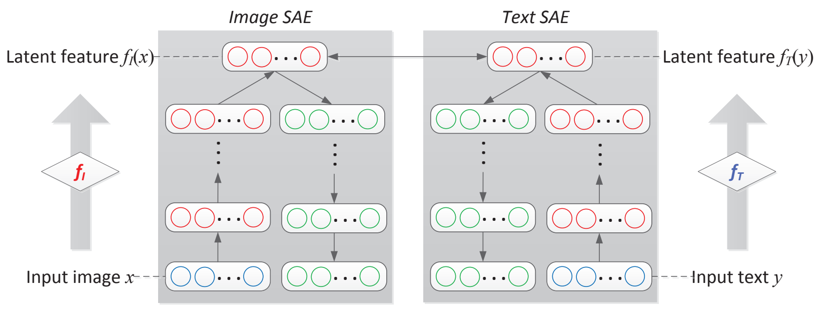


Figure 5.3: Model of MSAE, which consists of one SAE for each modality. The trained SAE maps input data into latent features.

eters. A complete training process is illustrated in Figure 5.2. In *stage I*, one mapping function is trained independently for each modality with the objective to map similar features in one modality close to each other in the latent space. This training stage serves as the pre-training of stage II by providing a good initialization for the parameters. *stage II* optimizes Equation 5.1 to capture both intra-modal semantics and inter-modal semantics. The learned mapping functions project semantically relevant objects close to each other in the latent space as shown in the figure.

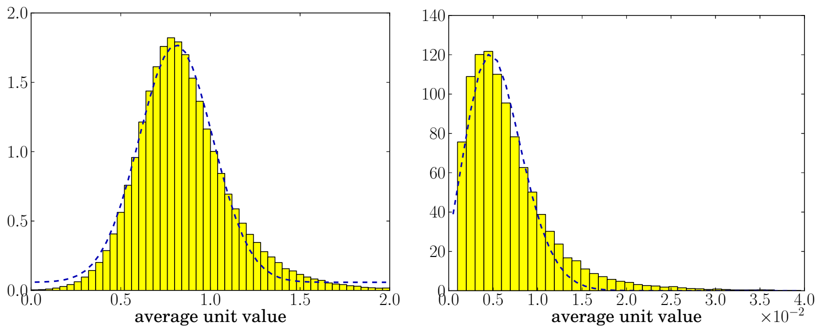
5.4 Unsupervised Approach – MSAE

This section presents an unsupervised learning algorithm called **MSAE** (Multi-modal Stacked Auto-Encoders) for learning the mapping function f_I and f_T. The model is shown in Figure 5.3 and would be explained in the following sections.

5.4.1 Realization of the Learning Objective

Modeling Intra-modal Semantics of Data

We extended SAEs (Section 2.1.3) to model intra-modal losses in the general learning objective (Equation 5.1). Specifically, $\mathcal{L}_I$ and $\mathcal{L}_T$ were modeled as the reconstruction errors for the image SAE and the text SAE respectively. Intuitively,



(a) (b)

Figure 5.4: Distribution of image (5.4a) and text (5.4b) features extracted from NUS-WIDE training dataset (See Section 5.7). Each figure is generated by averaging the units for each feature vector, and then plot the histogram for all data.

if the two reconstruction errors are small, the latent features generated by the top auto-encoder would be able to reconstruct the original input well, and consequently, capture the regularities of the input data well. This implies that, with small reconstruction error, two objects from the same modality that are similar in the original space would also be close in the latent space. In this way, we could capture the intra-modal semantics of data by minimizing $\mathcal{L}_I$ and $\mathcal{L}_T$ respectively. But to use SAEs, the decoders of the bottom auto-encoders should be designed carefully to handle different input features.

The raw (input) feature of an image is a high-dimensional real-valued vector (e.g., color histogram or bag-of-visual-words). In the encoder, each input image feature is mapped to a latent vector using Sigmoid function as the activation function s_e (Equation 2.10). However, in the decoder, the Sigmoid activation function, whose range is [0,1], performs poorly on reconstruction because the raw input unit (referring to one dimension) is not necessarily within [0,1]. To solve this issue, we followed Hinton (Hinton, 2010) and modeled the raw input unit as a linear unit with independent Gaussian noise. As shown in Figure 5.4a, the average unit value of image feature typically follows Gaussian distribution. When the input data is normalized with zero mean and unit variance, the Gaussian noise term can be omitted. In this case, we used an identity function for the activation function s_d in the bottom decoder. Let x_0 denote the input image feature vector, x_{2h} denote the feature vector reconstructed from the top latent feature x_h (h is the depth of the stacked auto-encoders). Using Euclidean distance to measure the reconstruction error, we defined $\mathcal{L}_I$ for x_0 as:

$$\mathcal{L}_I(x_0) = ||x_0 - x_{2h}||_2^2 \tag{5.2}$$

The raw (input) feature of text is a word count vector or tag occurrence vector [1]. We adopted the Rate Adapting Poisson model (Salakhutdinov and Hinton, 2009) for reconstruction because the histogram for the average value of text input unit generally follows Poisson distribution (Figure 5.4b). In this model, the activation function in the bottom decoder is

$$x_{2h} = s_d(z_{2h}) = l\frac{e^{z_{2h}}}{\sum_j e^{z_{2h_j}}} \tag{5.3}$$

where $l = \sum_j x_{0j}$ is the number of words in the input text, and $z_{2h} = W_{2h}^T x_{2h-1} + b_{2h}$. The probability of a reconstruction unit x_{2h_i} being the same as the input unit x_{0_i} is:

$$p(x_{2h_i} = x_{0_i}) = Pois(x_{0_i}, x_{2h_i}) \tag{5.4}$$

where $Pois(n, \lambda) = \frac{e^{-\lambda}\lambda^n}{n!}$. Based on Equation 5.4, we defined $\mathcal{L}_T$ using negative log likelihood:

$$\mathcal{L}_T(x_0) = -log \prod_i p(x_{2h_i} = x_{0_i}) \tag{5.5}$$

By minimizing $\mathcal{L}_T$, x_{2h} would be trained to be similar as x_0. In other words, the latent feature x_h is trained to reconstruct the input feature well, and thus preserves the regularities of the input data well.

[1]The binary value for each dimension indicates whether the corresponding tag appears or not.

Modeling Inter-modal Semantics of Data

Each relevant image-text pair (x_0, y_0) would be forwarded through the encoders of their stacked auto-encoders to generate latent feature vectors (x_h, y_h) (h is the height of the SAE). The inter-modal loss is then defined as,

$$\mathcal{L}_{I,T}(x_0, y_0) = dist(x_h, y_h) = ||x_h - y_h||_2^2 \tag{5.6}$$

By minimizing $\mathcal{L}_{I,T}$, the learned features would capture the inter-modal semantics of data. The intuition is quite straightforward: if two objects x_0 and y_0 are relevant, the distance between their latent features x_h and y_h shall be small.

5.4.2 Training

Following the training flow shown in Figure 5.2, in stage I, a SAE for the image modality and a SAE for the text modality are trained separately. Back-Propagation (LeCun et al., 1998) is used to calculate the gradients of the objective loss, i.e., $\mathcal{L}_I$ or $\mathcal{L}_T$, w.r.t., the parameters. Then the parameters are updated according to mini-batch Stochastic Gradient Descent (SGD), which averages the gradients contributed by a mini-batch of training records (images or text documents) and then adjusts the parameters. The learned image and text SAEs are fine-tuned in stage II by Back-Propagation and mini-batch SGD with the objective to find the optimal parameters that minimize the learning objective (Equation 5.1). In our experiment, we observed that the training would be more stable if we alternatively adjust one SAE with the other SAE fixed.

Setting $\boldsymbol{\beta_I}$ & $\boldsymbol{\beta_T}$ β_I and β_T are the weights of the reconstruction error of image and text SAEs respectively in the objective function (Equation 5.1). As mentioned in Section 5.3, they are set based on the quality of each modality's raw (input) feature. We use an example to illustrate the intuition. Consider a relevant object pair (x_0, y_0) from modality x and y. Assume x's feature is of low quality in capturing semantics (e.g., due to noise) while y's feature is of

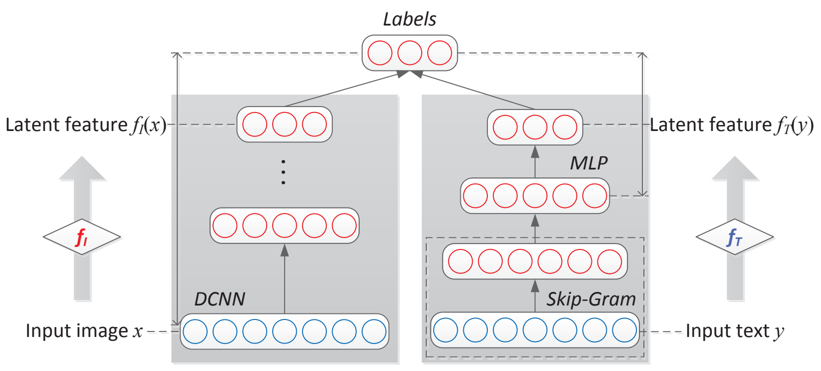


Figure 5.5: Model of MDNN, which consists of one DCNN for image modality, and one Skip-Gram + MLP for text modality. The trained DCNN (or Skip-Gram + MLP) maps input data into latent features.

high quality. If x_h and y_h are the latent features generated by minimizing the reconstruction error, then y_h can preserve the semantics well while x_h is not as meaningful due to the low quality of x_0. To solve this problem, we combine the inter-modal distance between x_h and y_h in the learning objective function and assign smaller weight to the reconstruction error of x_0. This is the same as increasing the weight of the inter-modal distance from x_h to y_h. As a result, the training algorithm would move x_h towards y_h to make their distance smaller. In this way, the semantics of low quality x_h could be enhanced by the high quality feature y_h.

In the experiment, we evaluated the quality of each modality's raw feature on a validation dataset by performing intra-modal search against the latent features learned in single-modal training. Modality with worse search performance is assigned a smaller weight. Notice that, because the dimensions of the latent space and the original space are usually of different orders of magnitude, the scale of $\mathcal{L}_I$, $\mathcal{L}_T$ and $\mathcal{L}_{I,T}$ are different. In the experiment, we also scaled β_I and β_T to make the losses comparable, i.e., within an order of magnitude.

5.5 Supervised Approach–MDNN

This section presents a supervised learning algorithm called **MDNN** (Multi-modal Deep Neural Network) based on a deep convolutional neural network (DCNN, Section 2.1.3) model and a neural language model (NLM, Section 2.1.3) to learn mapping functions for the image modality and the text modality respectively. The model is shown in Figure 5.5 and would be explained in the following sections.

5.5.1 Realization of the Learning Objective

Modeling Intra-modal Semantics of Data

Considering the outstanding performance of DCNNs in learning features for visual data (Donahue et al., 2013; Girshick et al., 2014), and NLMs in learning features for text data (Socher and Manning, 2013), we extended one instance of DCNN – AlexNet (Krizhevsky, Sutskever, and Hinton, 2012) and one instance of NLM – Skip-Gram model (SGM) (Mikolov et al., 2013) to model the intra-modal semantics of images and text respectively.

Image AlexNet is employed to serve as the mapping function f_I for image modality. An image x is represented by an RGB vector. The feature vector $f_I(x)$ learned by AlexNet is used to predict the associated labels of x. However, the objective of the original AlexNet is to predict single label of an image while in our case images are annotated with multiple labels. We thus followed (Gong et al., 2013a) to extend the softmax loss (Equation 2.13) to handle multiple labels as follows:

$$\mathcal{L}_I(x,t) = -\frac{1}{\sum_i t_i}\sum_i t_i \log p_i(x) \tag{5.7}$$

where $p_i(x)$ is defined in Equation 2.12. Different from SAE, which models reconstruction error to preserve intra-modal semantics, the extended AlexNet tries to minimize the prediction error $\mathcal{L}_I$ shown in Equation 5.7. By minimizing prediction error, the learned high-level feature vectors $f_I(x)$ are trained to be discriminative in predicting labels. Images with similar labels shall have similar feature vectors. In this way, the intra-modal semantics are preserved.

Text We extended SGM to learn the mapping function f_T for text modality. Due to the noisy nature of text (e.g., tags) associated with images (Liu et al., 2009), directly training the SGM over the tags would carry noise into the learned features. However, labels associated with images are carefully annotated and are more accurate. Hence, we adapted the SGM to integrate label information in order to learn robust features against noisy text (tags). To be specific, a SGM (Mikolov et al., 2013) is train with all tags associated with one image as an input sentence. After training, we would obtain one word embedding for each tag. By averaging word embeddings of all tags of one image, one text feature vector could be generated for those tags. Next, a Multi-Layer Perceptron (MLP) with two hidden layers is built on top of the SGM. The text feature vectors are fed into the MLP to predict image labels. Let y denote the input text (e.g., a set of image tags), $\tilde{y}$ denote the averaged word embedding generated by SGM for tags in y. MLP together with SGM serves as the mapping function f_T for the text modality,

$$f_T(y) = W_2 \cdot s(W_1\tilde{y} + b_1) + b_2 \tag{5.8}$$

$$s(v) = max(0, v) \tag{5.9}$$

where W_1 and W_2 are weight matrices, b_1 and b_2 are bias vectors, and $s()$ is the ReLU activation function (Krizhevsky, Sutskever, and Hinton, 2012)[2]. The loss function of MLP is similar to that of the extended AlexNet for image label

[2]We tried both the Sigmoid function and ReLU activation function for $s()$. ReLU offers better performance

prediction:

$$\mathcal{L}_T(y,t) = -\frac{1}{\sum_i t_i}\sum_i \log q_i(y) \tag{5.10}$$

$$q_i(y) = \frac{e^{f_T(y)_i}}{\sum_j e^{f_T(y)_j}} \tag{5.11}$$

By requiring the learned text latent features $f_T(y)$ to be discriminative for predicting labels, we could model the intra-modal semantics for the text modality [3].

Modeling Inter-modal Semantics of Data

The general learning objective in Equation 5.1 is realized using Equation 5.7 and 5.10 respectively. Euclidean distance is used to measure the difference of the latent features for an image-text pair, i.e., $L_{I,T}$ is defined similarly as in Equation 5.6. By minimizing the distance of latent features for an image-text pair, their latent features would be trained to be closer in the latent space. In this way, the inter-modal semantics are preserved.

5.5.2 Training

Similar to the training of MSAE, the training of MDNN consists of two steps. The first step trains the extended AlexNet and the extended NLM (i.e., MLP+Skip-Gram) separately[4]. The learned parameters are used to initialize the joint model. All training is conducted by Back-Propagation using mini-batch SGD to minimize the objective loss (Equation 5.1).

Setting $\boldsymbol{\beta_I}$ & $\boldsymbol{\beta_T}$ In the unsupervised training, we assigned larger β_I to make the training prone to preserve the intra-modal semantics of images if the input image feature is of higher quality than the text input feature, and vice versa.

[3]Notice that in our model, we fixed the word vectors learned by SGM. It can also be fine-tuned by integrating the objective of SGM (Equation 2.14) into Equation 5.10

[4]In our experiment, we used the parameters trained by Caffe (Jia et al., 2014b) to initialize the AlexNet to accelerate the training. We use Gensim (`http://radimrehurek.com/gensim/`) to pre-train the Skip-Gram model with the dimension of word vectors being 100

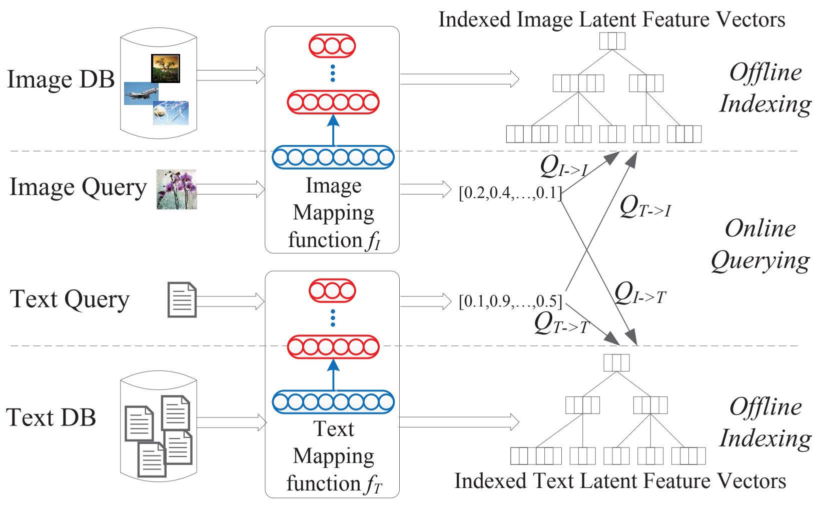


Figure 5.6: Illustration of query processing.

For supervised training, since the intra-modal semantics are preserved based on reliable labels, we did not distinguish the image modality from the text one in the joint training. In the experiment, we set $\beta_I = \beta_T = 1$. To make the three losses within one order of magnitude, we scaled the inter-modal distance by 0.01.

5.6 Query Processing

After the unsupervised (or supervised) training, each modality has a mapping function. Given a set of heterogeneous data sources, high-dimensional raw features (e.g., bag-of-visual-words or RGB feature for images) are extracted from each source and mapped into a common latent space using the learned mapping functions. MSAE uses the image (resp. text) SAE to project image (resp. text) input features into the latent space. MDNN uses the extended DCNN (resp. extended NLM) to map the image (resp. text) input feature into the common latent space.

After the mapping, we created VA-Files (Weber, Schek, and Blott, 1998) over the latent features (one per modality). VA-File is a classic index that can overcome the curse of dimensionality when answering nearest neighbor queries. It encodes each data point into a bitmap and the whole bitmap file is loaded into memory for efficient scanning and filtering. Only a few data points will be loaded into memory for verification. Given a query input, the search algorithm would check its media type and map it into the latent space through its modal-specific mapping function. Next, intra-modal and inter-modal searches are conducted against the corresponding index (i.e., the VA-File) shown in Figure 5.6. For example, the task of searching relevant tags of one image, i.e., $\mathbb{Q}_{I \to T}$, is processed by the index for the text latent vectors.

To further improve the search efficiency, the real-valued latent features are converted into binary features, whose distance are calculated using Hamming distance. The conversion is conducted using existing hash methods that preserve the neighborhood relationship. For example, in our experiment (Section 5.7.2), we used Spectral Hashing (Weiss, Torralba, and Fergus, 2008) , which converts real-valued vectors (data points) into binary codes with the objective to minimize the Hamming distance of data points that are close in the original Euclidean space. Other hashing approaches like (Song et al., 2011; Gong et al., 2013b) are also applicable.

The conversion from real-valued features to binary features trades off effectiveness for efficiency. Since there is information loss when real-valued data is converted to binaries, it affects the retrieval performance. The experiment section would present the trade-off between efficiency and effectiveness on binary features and real-valued features.

5.7 Experimental Study

This section provides an extensive performance study of our solution in comparison with the state-of-the-art methods. We examined both efficiency and effectiveness of our method including training overhead, query processing time and accuracy. Visualization of the training process is also provided to help understand the algorithms. In the rest of this section, we first introduce our evaluation metrics, and then present the performance of unsupervised approach and supervised approach respectively.

5.7.1 Evaluation Metrics

We evaluated the effectiveness of the mapping mechanism by measuring the accuracy of the multi-modal search, i.e., $\mathbb{Q}_{q\to t}(q, t \in \{T, I\})$, using the mapped latent features. Without specifications, searches were conducted against real-valued latent features using Euclidean distance. We used Mean Average Precision (MAP) (Manning, Raghavan, and Schütze, 2008), one of the standard information retrieval metrics, as the major evaluation metric. Given a set of queries, the Average Precision (AP) for each query q is calculated as,

$$AP(q) = \frac{\sum_{k=1}^{R} P(k)\delta(k)}{\sum_{j=1}^{R} \delta(j)} \tag{5.12}$$

where R is the size of the test dataset; $\delta(k) = 1$ if the k-th result is relevant, otherwise $\delta(k) = 0$; $P(k)$ is the precision of the result ranked at position k, which is the fraction of true relevant documents in the top k results. By averaging AP for all queries, we can get the MAP score. The larger the MAP score, the better the search performance. In addition to MAP, we measured the *precision* and *recall* of search tasks. Given a query, the *ground truth* is defined as: *if a result shares at least one common label (or category) with the query, it is considered as a relevant result; otherwise it is irrelevant.*

Besides effectiveness, we also evaluated the training overhead in terms of time cost and memory consumption. Query processing time would be reported at last.

5.7.2 Experimental Study of MSAE

First, we describe the datasets used for unsupervised training. Second, an analysis of the training process by visualization is presented. Last, comparison with

Table 5.1: Statistics of Datasets for Unsupervised Training.

Dataset	**NUS-WIDE**	**Wiki**	**Flickr1M**
Total size	190,421	2,866	1,000,000
Training set	60,000	2,000	975,000
Validation set	10,000	366	6,000
Test set	120,421	500	6,000
Average Text Length	6	131	5

previous works, including CVH (Kumar and Udupa, 2011), CMSSH (Bronstein et al., 2010) and LCMH (Zhu et al., 2013) are provided. [5] All experiments were conducted on CentOS 6.4 using CUDA 5.5 with NVIDIA GPU (GeForce GTX TITAN). The size of main memory is 64GB and the size GPU memory is 6GB. The original code and hyper-parameter settings are available online [6].

Datasets

Unsupervised training uses relevant image text pairs as training data, which are easy to collect. Three datasets were selected to evaluate the performance—NUS-WIDE (Chua et al., July 8-10, 2009), Wiki (Rasiwasia et al., 2010) and Flickr1M (Huiskes and Lew, 2008).

NUS-WIDE The dataset contains 269,648 images from Flickr, with each image associated with 6 tags on average. We refer to the image and its tags as an image-text pair. There are 81 ground truth labels manually annotated for evaluation. Following previous works (Liu et al., 2011; Zhu et al., 2013), we extracted 190,421 image-text pairs annotated with the most frequent 21 labels and split them into three subsets for training, validation and test respectively. The size of each

[5]The code and parameter configurations for CVH and CMSSH are available online at http://www.cse.ust.hk/~dyyeung/code/mlbe.zip; The code for LCMH is provided by the authors. Parameters are set according to the suggestions provided in the paper.

[6]http://www.comp.nus.edu.sg/~wangwei/code

subset is shown in Table 5.1. 100 (resp. 1000) queries were randomly selected from the validation (resp. test) dataset. Image and text features are provided in the dataset (Chua et al., July 8-10, 2009). An image is represented by a 500 dimensional bag-of-visual-words (SIFT) vector. Image tags are represented by a $1,000$ dimensional tag occurrence vector.

Wiki This dataset contains 2,866 image-text pairs from the Wikipedia's featured articles. An article in Wikipedia contains multiple sections. The text and its associated image in one section is considered as an image-text pair. Every image-text pair has a label inherited from the article's category (there are 10 categories in total). We randomly split the dataset into three subsets as shown in Table 5.1. For validation (resp. test), we randomly selected 50 (resp. 100) pairs from the validation (resp. test) set as the query set. Images were represented by 128 dimensional bag-of-visual-words vectors based on SIFT feature. For text, we constructed a vocabulary with the most frequent 1,000 words excluding stop words, and represented one text section by 1,000 dimensional word count vector like (Lu et al., 2013). The average number of words in one section was 131 (much larger than that in NUS-WIDE). To avoid overflow in Equation 5.4 and smooth the text input, we normalized each unit x as $\log(x+1)$ (Salakhutdinov and Hinton, 2009).

Flickr1M This dataset contains 1 million images associated with tags from Flickr. 25,000 of them are annotated with labels (there are 38 labels in total). The image feature is a 3,857 dimensional vector concatenated by SIFT feature, color histogram, etc (Srivastava and Salakhutdinov, 2012). Like NUS-WIDE, the text feature is represented by a tag occurrence vector with 2,000 dimensions. All the image-text pairs without annotations were used for training. For validation and test, we randomly selected 6,000 pairs with annotations respectively, among which 1,000 pairs were used as queries.

Before training, we used ZCA whitening (Krizhevsky, 2009) to normalize each dimension of image feature to have zero mean and unit variance.

Baseline Approaches

We compare our approach against the following baseline methods,

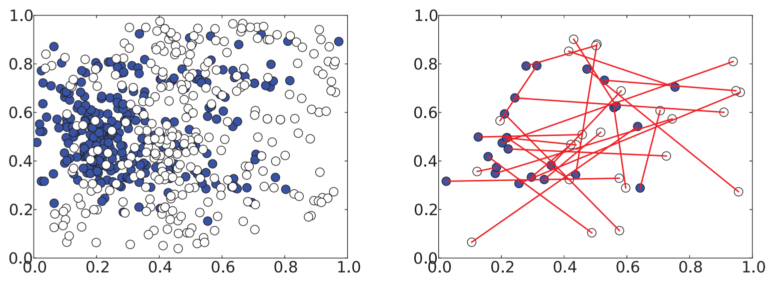
- LCMH (Zhu et al., 2013) exploits the intra-modal correlations by representing data from each modality using its distance to cluster centroids of the training data.
- CMSSH (Bronstein et al., 2010) uses a boosting method to learn the projection function for each dimension of the latent space.
- CVH (Kumar and Udupa, 2011) extends the Spectral Hashing (Weiss, Torralba, and Fergus, 2008) to learn a linear projection for each modality that minimizes the Euclidean distance of relevant data in the latent space.

Training Visualization

In this section, we present the visualization of the training process of MSAE using the NUS-WIDE dataset as an example to help understand the intuition of the training algorithm and the setting of the weight parameters, i.e., β_I and β_T. The training goal is to learn a set of effective mapping functions such that the mapped latent features capture both intra-modal semantics and inter-modal semantics well. Generally, the inter-modal semantics is preserved by minimizing the distance of the latent features of relevant inter-modal pairs. The intra-modal semantics is preserved by minimizing the reconstruction error of each SAE and through inter-modal semantics (see Section 5.4 for details).

First, following the training procedure in Section 5.4, we trained a 4-layer image SAE with the dimension of each layer as $500 \to 128 \to 16 \to 2$. Similarly, a 4-layer text SAE (the structure is $1000 \to 128 \to 16 \to 2$) was trained[7]. There is no standard guideline for setting the number of latent layers and units in each latent layer for deep learning (Bengio, 2012). In all our experiments, we adopted

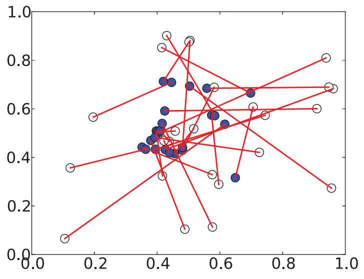
[7]The last layer with two units is for visualization purpose, such that the latent features could be showed in a 2D space



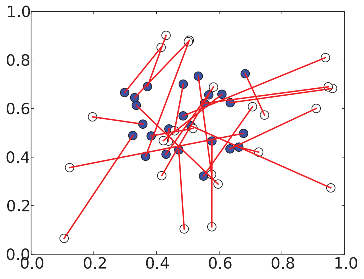
(a) 300 random image-text pairs. (b) 25 image-text pairs.

Figure 5.7: Visualization of latent features after projecting them into 2D space (Blue points are image latent features; White points are text latent features. Relevant image-tex pairs are connected using red lines).

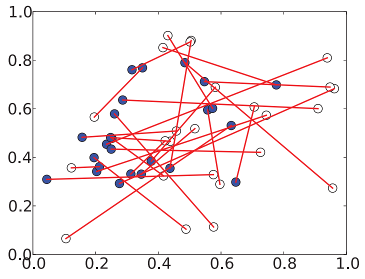
the widely used pyramid-like structure (Hinton and Salakhutdinov, 2006; Ciresan et al., 2012), i.e. decreasing layer size from the bottom (or first hidden) layer to the top layer. In our experiment, we observed that 2 latent layers perform better than a single latent layer. But there was no significant improvement from 2 latent layers to 3 latent layers. Latent features of sampled image-text pairs from the validation set are plotted in Figure 5.7a. The pre-training stage initializes SAEs to capture regularities of the original features of each modality in the latent features. On the one hand, the original features may be of low quality to capture intra-modal semantics. In such a case, the latent features would also fail to capture the intra-modal semantics. We evaluated the quality of the mapped latent features from each SAE by intra-modal search on the validation dataset. The MAP of the image intra-modal search is about 0.37, while that of the text intra-modal search is around 0.51. On the other hand, as the SAEs were trained separately, inter-modal semantics were not considered. We randomly picked 25 relevant image-text pairs and connected them with red lines as shown in Figure 5.7b. We can see the latent features of most pairs are far away from each other, which indicates that the inter-modal semantics are not captured by these latent features. To solve the above problems, we integrated the inter-modal loss in the learning objective as Equation 5.1. In the following figures, we only plot the distribution of these 25 pairs for ease of illustration.



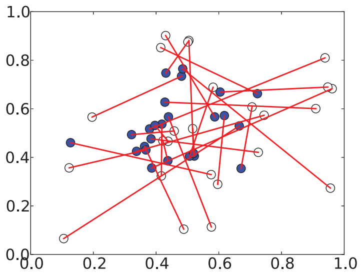
(a) $\beta_I = 0$, epoch 1



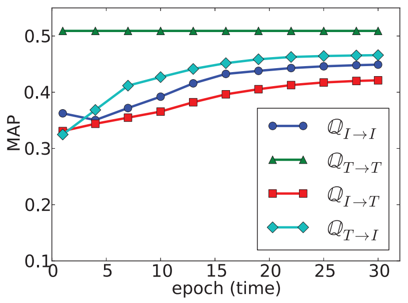
(b) $\beta_I = 0$, epoch 30



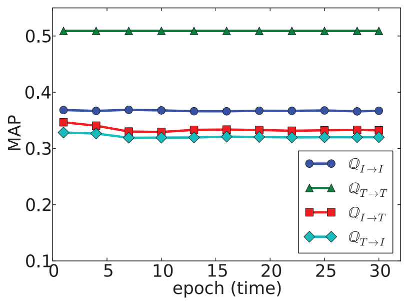
(c) $\beta_I = 0.01$, epoch 1



(d) $\beta_I = 0.01$, epoch 30



(e) $\beta_I = 0$

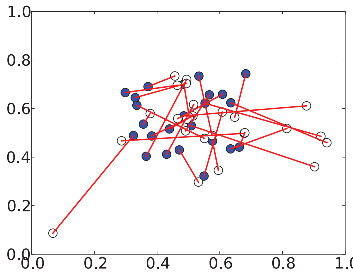


(f) $\beta_I = 0.01$

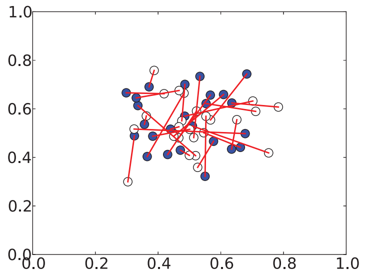
Figure 5.8: Adjusting image SAE with different β_I and text SAE fixed (a-d show the positions of features of image-text pairs in 2D space).

Second, we adjusted the image SAE with the text SAE fixed from epoch 1 to epoch 30. One epoch means one pass of the whole training dataset. Since the MAP of the image intra-modal search is worse than that of the text intra-modal search, according to the intuition in Section 5.3, we should use a small β_I to decrease the weight of image reconstruction error $\mathcal{L}_I$ in the objective function, i.e., Equation 5.1. To verify this, we compared the performance of two choices of β_I, namely $\beta_I = 0$ and $\beta_I = 0.01$. The first two rows of Figure 5.8 show the latent features generated by the image SAE after epoch 1 and epoch 30. Comparing image-text pairs in Figure 5.8b and 5.8d, we can see that with smaller β_I, the image latent features move closer to their relevant text latent features. This is in accordance with Equation 5.1, where smaller β_I relaxes the restriction on the image reconstruction error, and in turn increases the weight for inter-modal distance $\mathcal{L}_{I,T}$. By moving close to relevant text latent features, the image latent features gain more semantics. As shown in Figure 5.8e, the MAPs increase as training goes on. MAP of $\mathbb{Q}_{T\to T}$ does not change because the text SAE is fixed. When $\beta_I = 0.01$, the MAPs do not increase in Figure 5.8f. This is because image latent features hardly move close to the relevant text latent features as shown in Figure 5.8c and 5.8d. We can see that the text modality is of better quality for this dataset. Hence, it should be assigned a larger weight. However, we cannot set a too large weight for it as explained in the following paragraph.

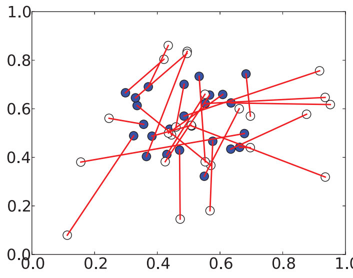
Third, we adjusted the text SAE with the image SAE fixed from epoch 31 to epoch 60. We also compared two choices of β_T, namely 0.01 and 0.1. β_I is set to 0. Figure 5.9 shows the snapshots of latent features and the MAP curves of each setting. From Figure 5.8b to 5.9a, which are two consecutive snapshots taken from epoch 30 and 31 respectively, we can see that the text latent features move much closer to the relevant image latent features. It leads to the big changes of MAPs at epoch 31 in Figure 5.9e. For example, $\mathbb{Q}_{T\to T}$ substantially drops from 0.5 to 0.46. This is because the sudden moves towards images change the intra-modal relationships of text latent features. Another big change happens on $\mathbb{Q}_{I\to T}$, whose MAP increases dramatically. The reason is that when we fix



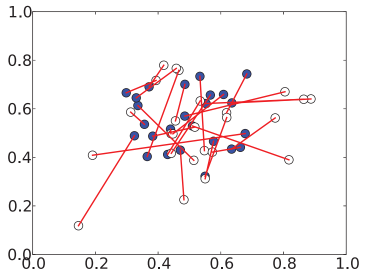
(a) $\beta_T = 0.01$,epoch 31



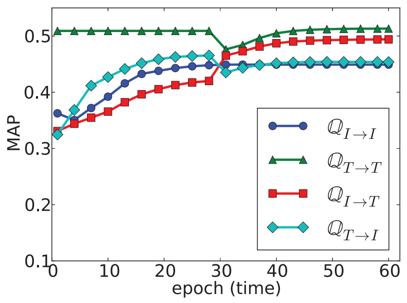
(b) $\beta_T = 0.01$,epoch 60



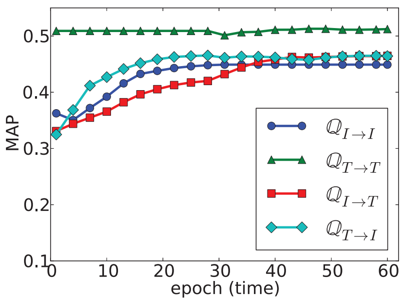
(c) $\beta_T = 0.1$,epoch 31



(d) $\beta_T = 0.1$,epoch 60



(e) $\beta_T = 0.01$



(f) $\beta_T = 0.1$

Figure 5.9: Adjusting text SAE with different β_T and image SAE fixed (a-d show the positions of features of image-text pairs in 2D space).

the text features from epoch 1 to 30, an image feature I is pulled to be close to (or nearest neighbor of) its relevant text feature T. However, T may not be the reverse nearest neighbor of I. In epoch 31, T is moved towards I such that T is more likely to be the reverse nearest neighbor of I. Hence, the MAP of query $\mathbb{Q}_{I\to T}$ is greatly improved. On the contrary, $\mathbb{Q}_{T\to I}$ decreases. From epoch 32 to epoch 60, the text latent features on the one hand move close to relevant image latent features slowly, and on the other hand rebuild their intra-modal relationships. The latter is achieved by minimizing the reconstruction error $\mathcal{L}_T$ to capture the semantics of the original features. Therefore, both $\mathbb{Q}_{T\to T}$ and $\mathbb{Q}_{I\to T}$ grows gradually. Comparing Figure 5.9a and 5.9c, we can see the distance of relevant latent features in Figure 5.9c is larger than that in Figure 5.9a. The reason is that when β_T is larger, the objective function in Equation 5.1 pays more effort to minimize the reconstruction error $\mathcal{L}_T$. Consequently, less effort is paid to minimize the inter-modal distance $\mathcal{L}_{I,T}$. Hence, relevant inter-modal pairs cannot move closer. This effect is reflected as minor changes of MAPs at epoch 31 in Figure 5.9f in contrast with that in Figure 5.9e. Similarly, small changes happen between Figure 5.9c and 5.9d, which leads to minor MAP changes from epoch 32 to 60 in Figure 5.9f.

Evaluation of Model Effectiveness on NUS_WIDE Dataset

We first report the mean average precision (MAP) of our method using Euclidean distance against real-valued features. Let L be the dimension of the latent space. Our MSAE was configured with 3 layers, where the image features were mapped from 500 dimensions to 128, and finally to L. Similarly, the dimension of text features were reduced from $1000 \to 128 \to L$ by the text SAE. β_I and β_T were set to 0 and 0.01 respectively according to Section 5.7.2. We tested L with values 16, 24 and 32. The results compared with other methods are reported in Table 5.2, which shows that MSAE achieves the best performance for all four search tasks with an average improvement of 17%, 27%, 21%, and 26% for $\mathbb{Q}_{I\to I}$, $\mathbb{Q}_{T\to T}$,$\mathbb{Q}_{I\to T}$, and $\mathbb{Q}_{T\to I}$ respectively. CVH and CMSSH prefer smaller L in queries $\mathbb{Q}_{I\to T}$ and

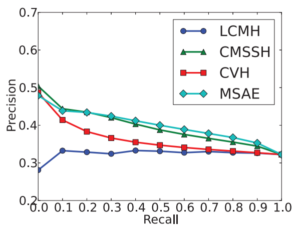
Table 5.2: Mean average precision on NUS-WIDE dataset.

Task		$Q_{I \to I}$				$Q_{T \to T}$				$Q_{I \to T}$				$Q_{T \to I}$			
Algorithm		LCMH	CMSSH	CVH	MSAE	LCMH	CMSSH	CVH	MSAE	LCMH	CMSSH	CVH	MSAE	LCMH	CMSSH	CVH	MSAE
Dimension of	16	0.353	0.355	0.365	**0.417**	0.373	0.400	0.374	**0.498**	0.328	0.391	0.359	**0.447**	0.331	0.337	0.368	**0.432**
Latent Space	24	0.343	0.356	0.358	**0.412**	0.373	0.402	0.364	**0.480**	0.333	0.388	0.351	**0.444**	0.323	0.336	0.360	**0.427**
L	32	0.343	0.357	0.354	**0.413**	0.374	0.403	0.357	**0.470**	0.333	0.382	0.345	**0.402**	0.324	0.335	0.355	**0.435**

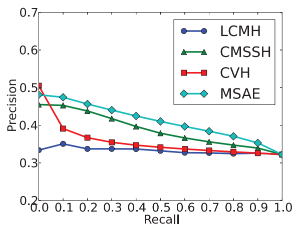
$\mathbb{Q}_{T\rightarrow I}$. The reason is that it needs to train far more parameters with Larger L and the learned models will be farther from the optimal solutions. Our method is less sensitive to the value of L. This is probably because with multiple layers, MSAE has stronger representation power and thus is more robust under different L.

Figure 5.10 shows the precision-recall curves, and the recall-candidates ratio curves (used by (Zhen and Yeung, 2012; Zhu et al., 2013)) which show the change of recall when inspecting more results on the returned rank list. We omit the figures for $\mathbb{Q}_{T\rightarrow T}$ and $\mathbb{Q}_{I\rightarrow I}$ as they show similar trends as $\mathbb{Q}_{T\rightarrow I}$ and $\mathbb{Q}_{I\rightarrow T}$. Our method achieves the best accuracy except when recall $= 0$ [8], where precision p implies that the nearest neighbor of the query appears in the $\frac{1}{p}$-th returned result. This indicates that our method performs the best for general top-k similarity retrieval except k=1. For the recall-candidates ratio, the curve of MSAE is always above those of other methods. It shows that MSAE has better recall when inspecting the same number of objects. In other words, our method ranks more relevant objects at higher (front) positions.

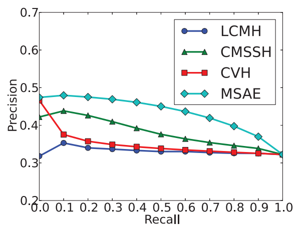
Besides real-valued features, we also conducted experiments against binary latent features for which Hamming distance is used as the distance function. In our implementation, we used Spectral Hashing (Weiss, Torralba, and Fergus, 2008) to convert real-valued latent feature vectors into binary codes. Other comparison algorithms used their own conversion mechanisms. The MAP scores are reported in Table 5.3. We can see that 1) MSAE performs better than other methods. 2) The MAP scores using Hamming distance is not as good as that of Euclidean distance. This is due to the possible information loss by converting real-valued features into binary features.



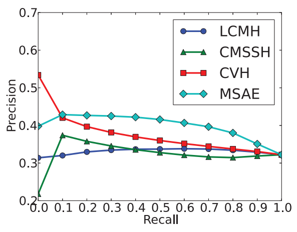
(a) $\mathbb{Q}_{I\to T}$, $L = 16$



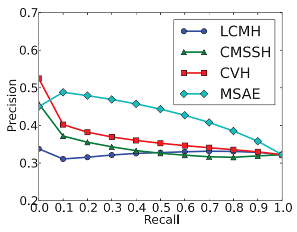
(b) $\mathbb{Q}_{I\to T}$, $L = 24$



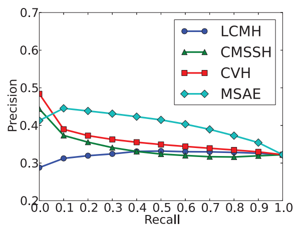
(c) $\mathbb{Q}_{I\to T}$, $L = 32$



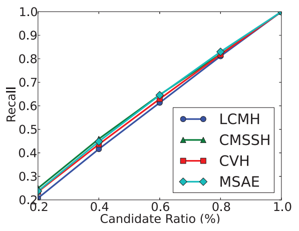
(d) $\mathbb{Q}_{T\to I}$, $L = 16$



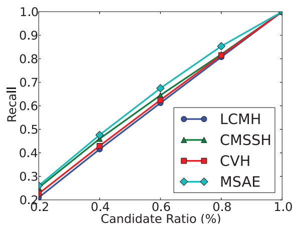
(e) $\mathbb{Q}_{T\to I}$, $L = 24$



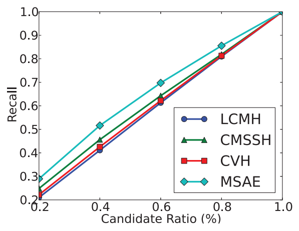
(f) $\mathbb{Q}_{T\to I}$, $L = 32$



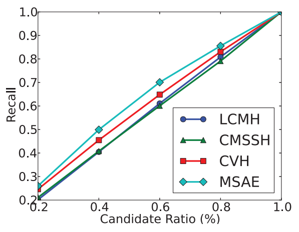
(g) $\mathbb{Q}_{I\to T}$, $L = 16$



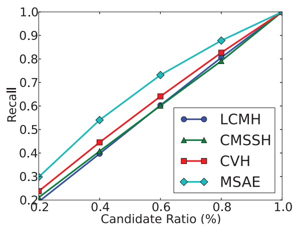
(h) $\mathbb{Q}_{I\to T}$, $L = 24$



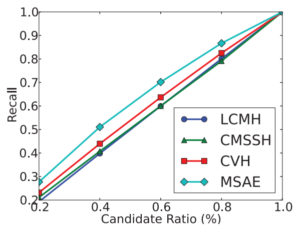
(i) $\mathbb{Q}_{I\to T}$, $L = 32$



(j) $\mathbb{Q}_{T\to I}$, $L = 16$



(k) $\mathbb{Q}_{T\to I}$, $L = 24$



(l) $\mathbb{Q}_{T\to I}$, $L = 32$

Figure 5.10: Precision-Recall (P-R) and Recall-Candidates ratio on NUS-WIDE dataset.

Table 5.3: Mean average precision on NUS-WIDE dataset (using Binary Latent Features).

Task		$\mathbb{Q}_{I \to I}$				$\mathbb{Q}_{T \to T}$				$\mathbb{Q}_{I \to T}$				$\mathbb{Q}_{T \to I}$			
Algorithm		LCMH	CMSSH	CVH	MSAE	LCMH	CMSSH	CVH	MSAE	LCMH	CMSSH	CVH	MSAE	LCMH	CMSSH	CVH	MSAE
Dimension of Latent Space L	16	0.353	0.357	0.352	**0.376**	0.387	0.391	0.379	**0.397**	0.328	0.339	0.359	**0.364**	0.325	0.346	0.359	**0.392**
	24	0.347	0.358	0.346	**0.368**	0.392	0.396	0.372	**0.412**	0.333	0.346	0.353	**0.371**	0.324	0.352	0.353	**0.380**
	32	0.345	0.358	0.343	**0.359**	0.395	0.397	0.365	**0.434**	0.320	0.340	0.348	**0.373**	0.318	0.347	0.348	**0.372**

Table 5.4: Mean average precision on Wiki dataset.

Task		$\mathbb{Q}_{I\to I}$				$\mathbb{Q}_{T\to T}$				$\mathbb{Q}_{I\to T}$				$\mathbb{Q}_{T\to I}$			
Algorithm		LCMH	CMSSH	CVH	MSAE	LCMH	CMSSH	CVH	MSAE	LCMH	CMSSH	CVH	MSAE	LCMH	CMSSH	CVH	MSAE
Dimension of	16	0.146	0.148	0.147	**0.162**	0.359	0.318	0.153	**0.462**	0.133	0.138	0.126	**0.182**	0.117	0.140	0.122	**0.179**
Latent Space	24	0.149	0.151	0.150	**0.161**	0.345	0.320	0.151	**0.437**	0.129	0.135	0.123	**0.176**	0.124	0.138	0.123	**0.168**
L	32	0.147	0.149	0.148	**0.162**	0.333	0.312	0.152	**0.453**	0.137	0.133	0.128	**0.187**	0.119	0.137	0.123	**0.179**

Evaluation of Model Effectiveness on Wiki Dataset

We conducted similar evaluations on **Wiki** dataset as on NUS-WIDE. For MSAE with latent feature of dimension L, the structure of its image SAE was $128 \rightarrow 128 \rightarrow L$, and the structure of its text SAE was $1000 \rightarrow 128 \rightarrow L$. Similar to the settings on NUS-WIDE, β_I was set to 0 and β_T was set to 0.01.

The performance is reported in Table 5.4. MAPs on **Wiki** dataset are much smaller than those on NUS-WIDE except for $\mathbb{Q}_{T \rightarrow T}$. This is because the images of **Wiki** are of much lower quality. It contains only $2,000$ images that are highly diversified, making it difficult to capture the semantic relationships within images, and between images and text. Query task $\mathbb{Q}_{T \rightarrow T}$ is not affected as Wkipedia's featured articles are well edited and rich in text information. In general, our method achieved an average improvement of 8.1%, 30.4%, 32.8%, 26.8% for $\mathbb{Q}_{I \rightarrow I}$, $\mathbb{Q}_{T \rightarrow T}$,$\mathbb{Q}_{I \rightarrow T}$, and $\mathbb{Q}_{T \rightarrow I}$ respectively. We do not plot the precision-recall curves and recall-candidates ratio curves as they showed similar trends to those of NUS-WIDE.

Evaluation of Model Effectiveness on Flickr1M Dataset

We configured a 4-layer image SAE as $3857 \rightarrow 1000 \rightarrow 128 \rightarrow L$, and a 4-layer text SAE as $2000 \rightarrow 1000 \rightarrow 128 \rightarrow L$ for this dataset. Different from the other two datasets, the original image feature of Flickr1M are of higher quality as it consists of both local and global features. For intra-modal search, the image latent feature performed equally well as the text latent feature. Therefore, we set both β_I and β_T to 0.01.

The MAP performances of MSAE and CVH are compared in Table 5.5. MSAE outperforms CVH in most of the search tasks. LCMH and CMSSH ran out of memory in the training stage, hence we do not report them.

[8]Here, recall $r = \frac{1}{\#all\ relevant\ results} \approx 0$.

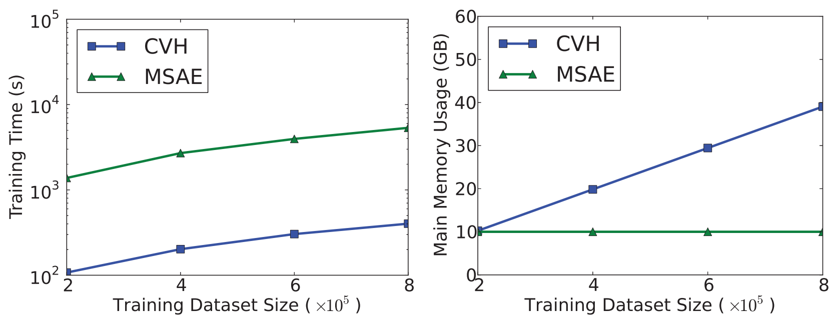
Table 5.5: Mean average precision on Flickr1M dataset.

Task		$\mathbb{Q}_{I\to I}$		$\mathbb{Q}_{T\to T}$		$\mathbb{Q}_{I\to T}$		$\mathbb{Q}_{T\to I}$	
Algorithm		CVH	MSAE	CVH	MSAE	CVH	MSAE	CVH	MSAE
L	16	**0.622**	0.621	0.610	**0.624**	0.610	**0.632**	**0.616**	0.608
	24	0.616	**0.619**	0.604	**0.629**	0.605	**0.628**	**0.612**	**0.612**
	32	0.603	**0.622**	0.587	**0.630**	0.588	**0.632**	0.598	**0.614**

Evaluation of Training Cost

We used the largest dataset Flickr1M to evaluate the training cost of time and memory consumption. The results are reported in Figure 5.11. The training cost of LCMH and CMSSH are not reported because they ran out of memory on this dataset. We can see that the training time of MSAE and CVH increases linearly with respect to the size of the training dataset. Due to the stacked structure and multiple iterations of passing the dataset, MSAE is not as efficient as CVH. Roughly, the overhead is proportional to the number of training iterations times the height of MSAE.

Figure 5.11b shows the memory usage of the training process. Given a training dataset, MSAE splits them into mini-batches and conducts the training batch by batch. It stores the model parameters and one mini-batch in memory, both of which are independent of the training dataset size. Hence, the memory usage stays constant when the size of the training dataset increases. The actual minimum memory usage for MSAE could be smaller than 10GB. In our experiments, we allocated more space to load multiple mini-batches into memory to save disk reading cost. CVH has to load all training data into memory for matrix operations. Therefore, its memory usage increases with respect to the size of the training dataset.



(a) (b)

Figure 5.11: Training cost comparison on Flickr1M dataset.

Evaluation of Query Processing Efficiency

We compared the efficiency of query processing using binary latent features and real-valued latent features. Notice that all methods (i.e., MSAE, CVH, CMSSH and LCMH) performed similarly in query processing after mapping the original data into latent features of same dimension. Data from the **Flickr1M** training dataset was mapped into a 32 dimensional latent space to form a large dataset for searching. To speed up the query processing of real-valued latent features, we created an index (i.e., VA-File (Weber, Schek, and Blott, 1998)) for each modality. For binary latent features, we did not create any indexes, as linear scan offered decent performance as shown in Figure 5.12. It shows the time of searching 50 nearest neighbors (averaged over 100 random queries) against datasets represented using binary latent features (based on Hamming distance) and real-valued features (based on Euclidean distance) respectively. We can see that the querying time increases linearly with respect to the dataset size for both binary and real-valued latent features. But, the searching against binary latent features is 10× faster than that against real-valued latent features. This is because the computation of Hamming distance is more efficient than that of Euclidean distance.

By taking into account the results from effectiveness evaluations, we can see that there is a trade-off between efficiency and effectiveness in feature representation.

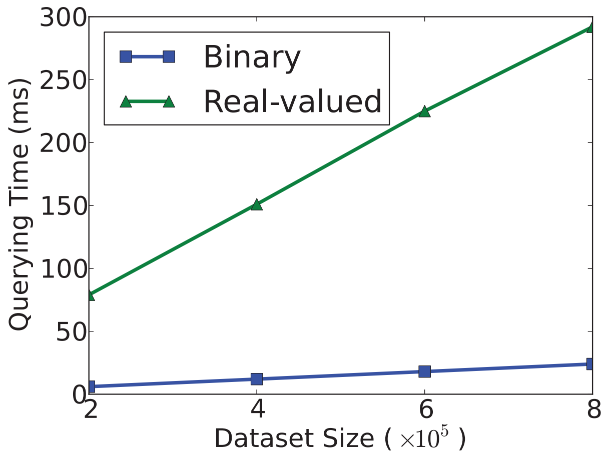


Figure 5.12: Querying time comparison using real-valued and binary latent features.

The binary encoding greatly improves the efficiency in the expense of accuracy degradation.

5.7.3 Experimental Study of MDNN

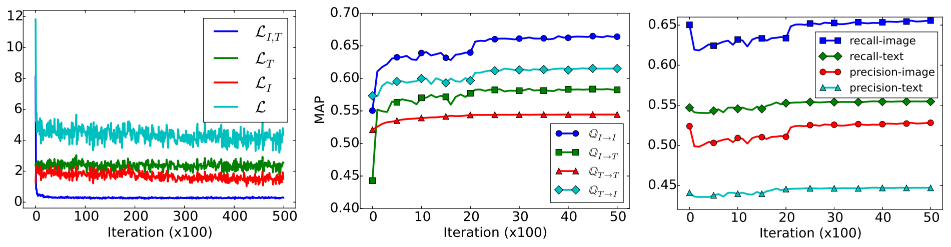
Datasets

Supervised training requires the input image-text pairs to be associated with additional semantic labels. Since Flickr1M does not have labels and Wiki dataset has too few labels that are not discriminative enough, we used NUS-WIDE dataset to evaluate the performance of supervised training. We extracted $203,400$ labeled pairs, among which $150,000$ were used for training. The remaining pairs were evenly partitioned into two sets for validation and testing. From both sets, we randomly selected 2000 pairs as queries. This labeled dataset is named NUS-WIDE-a.

We further extracted another dataset from NUS-WIDE-a by filtering those pairs with more than one label. This dataset, denoted as NUS-WIDE-b, was used to compare with DeViSE (Frome et al., 2013), which was designed for training against images annotated with single label. In total, we obtained $76,000$ pairs.

Table 5.6: Statistics of datasets for supervised training.

Dataset	**NUS-WIDE-a**	**NUS-WIDE-b**
Total size	203, 400	76,000
Training set	150,000	60,000
Validation set	26,700	80,000
Test set	26,700	80,000



(a) Training loss (b) MAP on validation data (c) P-R on validation data

Figure 5.13: Visualization of training on NUS-WIDE-a.

Among them, we randomly selected $60,000$ pairs for training and the rest were evenly partitioned for validation and testing. 1000 queries were randomly selected from the two datasets respectively.

Training Analysis

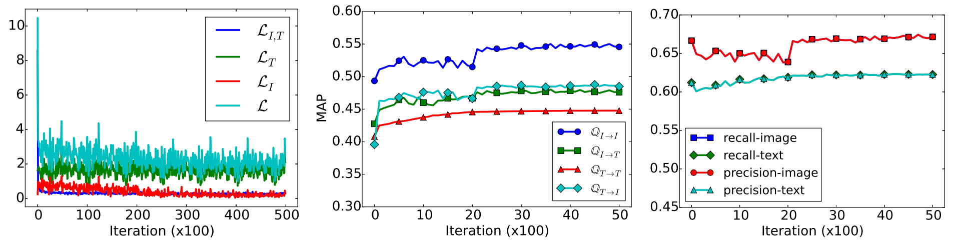
NUS-WIDE-a In Figure 5.13a, we plot the total training loss $\mathcal{L}$ and its components ($\mathcal{L}_I$, $\mathcal{L}_T$ and $\mathcal{L}_{I,T}$) in the first $50,000$ iterations (one iteration for one mini-batch) against the NUS-WIDE-a dataset. We can see that the training converges rather quickly. The training loss drops dramatically at the very beginning and then decreased slowly. This is because initially the learning rate is large and the parameters approaches quickly towards the optimal values. Another observation is that the intra-modal loss $\mathcal{L}_I$ for the image modality is smaller than $\mathcal{L}_T$ for the text modality. This is because some tags may be noisy or not very relevant to the associated labels for the main visual content in the images. It is difficult to learn a set of parameters to map noisy tags into the latent space and well predict the ground truth labels. The inter-model training loss was calculated at a different scale and was normalized to be within one order of magnitude as $\mathcal{L}_I$ and $\mathcal{L}_T$.

The MAPs for all types of searches using supervised training model are shown in Figure 5.13b. As can be seen, the MAPs first gradually increases and then becomes stable in the last few iterations. It is worth noting that the MAPs are much higher than the results of unsupervised training (MSAE) in Figure 5.9. There are two reasons for the superiority. First, the supervised training algorithm (MDNN) exploits DCNN and NLM to learn better visual and text features respectively. Second, labels bring in more semantics and enable latent features learn more robust to noises in input data (e.g., visual irrelevant tags).

Besides MAP, we also evaluated MDNN for multi-label prediction based on precision and recall. For each image (or text), we looked at its labels with the largest k probabilities based on Equation 2.12 (or 5.11). For the i-th image (or text), let N_i denote the number of labels out of k that belong to its ground truth label set, and T_i the size of its ground truth label set. The precision and recall are defined according to (Gong et al., 2013a) as follows:

$$precison = \frac{\sum_{i=1}^{n} N_i}{k * n}, \quad recall = \frac{\sum_{i=1}^{n} N_i}{\sum_{i=1}^{n} T_i} \tag{5.13}$$

where n is the test set size. As shown in Figure 5.13c ($k = 3$), the performance decreases at the early stage and then goes up. This is because at the early stage, in order to minimize the inter-modal loss, the training may disturb the pre-trained parameters fiercely, which affects the intra-modal search performance. Once the inter-modal loss is reduced to a certain level, it starts to adjust the parameters to minimize both inter-modal loss and intra-modal loss. Correspondingly, the classification performance starts to increase. We can also see that the performance of latent text features is not as good as that of latent image features due to the noises in tags. We used the same experiment setting as that in (Gong et al.,



(a) Training loss (b) MAP on validation data (c) P-R on validation data

Figure 5.14: Visualization of training on NUS-WIDE-b.

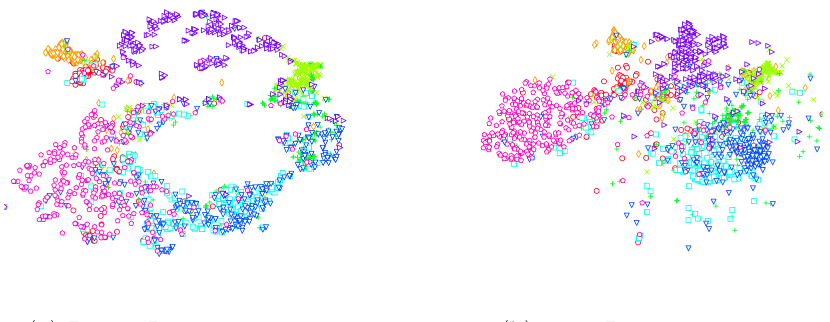
Table 5.7: Mean average precision using real-valued latent feature.

Task		$\mathbb{Q}_{I\to I}$			$\mathbb{Q}_{T\to T}$			$\mathbb{Q}_{I\to T}$			$\mathbb{Q}_{T\to I}$		
Algorithm		MDNN	DeViSE-L	DeViSE-T	MDNN	DeViSE-L	DeViSE-T	MDNN	DeViSE-L	DeViSE-T	MDNN	DeViSE-L	DeViSE-T
Dataset	NUS-WIDE-a	**0.669**	0.5619	0.5399	**0.541**	0.468	0.464	**0.587**	0.483	0.517	**0.612**	0.502	0.515
	NUS-WIDE-b	**0.556**	0.432	0.419	**0.466**	0.367	0.385	**0.497**	0.270	0.399	**0.495**	0.222	0.406

2013a), the (over all) precision and recall was 7% and 7.5% higher than that in (Gong et al., 2013a) respectively.

NUS-WIDE-b Figure 5.14 shows the training results against the NUS-WIDE-b dataset. The results demonstrate similar patterns to those in Figure 5.13. However, MAPs becomes lower, possibly due to smaller training dataset size and fewer number of associated labels. In Figure 5.14c, the precision and recall for classification using image (or text) latent features are the same. This is because each image-text pair has only one label and $T_i = 1$. When we set $k = 1$, the denominator $k * n$ in precision is equal to $\sum_{i=1}^{n} T_i$ in recall.

2D Visualization To demonstrate that the learned mapping functions can generate semantic discriminative latent features, we extracted top-8 most popular labels and for each label, we randomly sampled 300 image-text pairs from the test dataset of NUS-WIDE-b. Their latent features were projected into a 2-dimensional space by t-SNE (van der Maaten, 2014). Figure 5.15a shows the 2-dimensional image latent features where one point represents one image feature and Figure 5.15b shows the 2-dimensional text features. Labels are distinguished using different shapes. We can see that the features are well clustered according



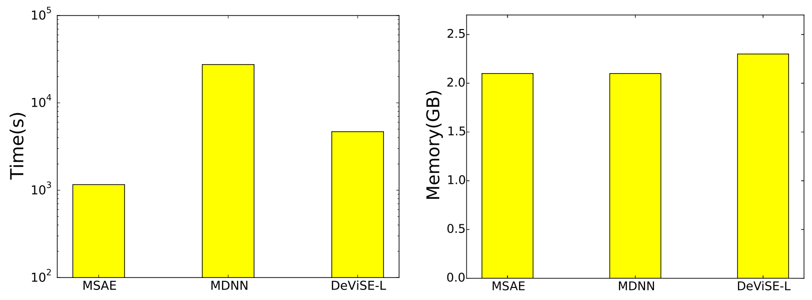
(a) Image Latent Feature (b) Text Latent Feature

Figure 5.15: Visualization of latent features learned by MDNN for the test dataset of NUSWDIE-a (features represented by the same shapes and colors are annotated with the same label).

to their labels. Further, the image features and text features semantically relevant to the same labels are projected to similar positions in the 2D space. For example, in both figures, the red circles are at the left side, and the blue right triangles are in the top area. The two figures together confirm that our supervised training is very effective in capturing semantic information for multi-modal data.

Evaluation of Model Effectiveness on NUS-WIDE Dataset

We compared MDNN with DeViSE (Frome et al., 2013) in terms of effectiveness of multi-modal retrieval. DeViSE maps image features into text feature space. The learning objective is to minimize the rank hinge loss based on the latent features of an image and its labels. We implemented this algorithm and extended it to handle multiple labels by averaging their word vector features. We denote this algorithm as DeViSE-L. Besides, we also implemented a variant of DeViSE denoted as DeViSE-T, whose learning objective is to minimize the rank hinge loss based on the latent features of an image and its tag(s). Similarly, if there were multiple tags, we averaged their word vectors. The results are shown in Table 5.7. The retrieval was conducted using real-valued latent feature and cosine similarity as the distance function. We can see that MDNN performs much better than both DeViSE-L and DeViSE-T for all four types of searches on both



(a) Training Time (b) Memory Consumption

Figure 5.16: Training cost comparison on NUSWIDE-a dataset.

NUS-WIDE-a and NUS-WIDE-b. The main reason is that the image tags are not all visually relevant, which prevents the text (tag) feature from capturing the visual semantics in DeViSE. MDNN exploits the label information in the training, which helps to train a model that can generate more robust feature against noisy input tags. Hence, the performance of MDNN is better.

Evaluation of Training Cost

We report the training cost in terms of training time (Fig. 5.16a) and memory consumption (Fig. 5.16b) on NUS-WIDE-a dataset. Training time includes the pre-training for each single modality and the joint multi-modal training. MDNN and DeViSE-L take longer time to train than MSAE, because the convolution operations in them are time consuming. Further, MDNN has pre-training stages for the image modality and text modality, and thus incurs longer training time than DeViSE-L. The memory footprint of MDNN is similar to that of DeViSE-L, as the two methods both rely on DCNN, which consumes most of the memory. DeViSE-L used features of higher dimension (100 dimension) than MDNN (81 dimension), which resulted in about 100 MegaBytes difference as shown in Fig. 5.16b.

Comparison with Unsupervised Approach

By comparing Table 5.7 and Table 5.2, we can see that the supervised approach–MDNN performs better than the unsupervised approach–MSAE. This is not surprising because MDNN consumes more information than MSAE. Although the two methods share the same general training objective, the exploitation of label semantics helps MDNN learn better features in capturing the semantic relevance of the data from different modalities. For memory consumption, MDNN and MSAE perform similarly (Fig. 5.16b).

5.8 Summary

In this chapter, we presented an application on top of SINGA using feed-forward neural networks. Particularly, we proposed a general framework (objective) for learning mapping functions for effective multi-modal retrieval. Both intra-modal and inter-modal semantic relationships of data from heterogeneous sources are captured in the general learning objective function. Given this general objective, we have implemented one unsupervised training algorithm and one supervised training algorithm separately to learn the mapping functions based on deep learning techniques. The unsupervised algorithm uses stacked auto-encoders as the mapping functions for the image modality and the text modality. It only requires simple image-text pairs for training. The supervised algorithm uses an extended DCNN as the mapping function for images and an extended NLM as the mapping function for text data. Label information is integrated in the training to learn robust mapping functions against noisy input data. The results of experiment confirmed the improvements of our method over previous works in search accuracy. The approaches presented in this chapter are published in the following papers.

- *Wei Wang, Beng Chin Ooi, Xiaoyan Yang, Dongxiang Zhang, and Yueting Zhuang. Effective multi-modal retrieval based on stacked auto-encoders. PVLDB,* *7(8):649-660, 2014*

- *Wei Wang, Xiaoyan Yang, Beng Chin Ooi, Dongxiang Zhang, and Yueting Zhuang. Effective deep learning-based multi-modal retrieval. VLDB Journal, Special issue of VLDB'14 best papers, 25(1):79-101, 2016*

CHAPTER 6

Conclusion and Future Work

6.1 Summary

In this dissertation, we conducted thorough study on improving the usability of deep learning from the system and application perspective.

We first investigated and analyzed the challenges of using deep learning models from the training system perspective, including stand-alone training and distributed training. For stand-alone training, operation scheduling and memory management would be significant for improving the usability and efficiency. We discussed some possible optimization techniques such as cost model for optimal operation scheduling and swapping data between CPU and GPU for reducing memory footprint. For distributed training, communication, consistency and fault-tolerance are major issues that restrict the scalability of the training system. Existing solutions and possible optimization approaches were highlighted, including bounded-asynchronous training and parameter compression.

Next, based on the analysis, we proposed a distributed system for improving the training efficiency. The system, called SINGA, was designed to be a general system to support various deep learning models, and as a scalable system that could effectively reduce the training time with more computing resources. The first design goal was achieved through the layer and neural net abstraction which are intrinsic data structures of deep learning models. We proposed a flexible system architecture towards the second goal, which could be customized to optimize different distributed training frameworks. We demonstrated the usability of SINGA by training three representative models with different structures on it. By comparing with other open source systems, SINGA was verified to be scalable.

Lastly, we applied SINGA to train deep learning models for the multi-modal retrieval application. We aimed at exploiting the excellent performance of deep learning models in feature extraction to improve the search accuracy. Our first model consisted of two simple auto-encoder models for extracting features for images and text documents respectively. In comparison, with existing works, the experiments showed that our proposed model could learn effective features for multi-modal retrieval. The accuracy was increased significantly on benchmark datasets. Our second model combined the advantages of the convolutional neural network (CNN) model and the word vector model. CNN is the state-of-the-art model for extracting image features, and the word vector model is a simple and effective approach for extracting features for text documents. The experimental results confirmed that the new model can further improve against the first model for multi-modal retrieval. Our models were important in the light of applying deep learning techniques for retrieval problems.

6.2 Future Work

I would like to extend the current works in the following two aspects.

6.2.1 Deep learning system optimization

The goal is to make deep learning systems easy to use, efficient, scalable and extensible.

- Deep learning is being adopted in many applications using different devices. It is desirable to provide a simple, flexible and extensible system in terms of installation, programming interface and result analysis (e.g., visualization).

- Efficiency is vital for a deep learning system due to the high computation cost and memory requirement from the training and prediction procedures. Runtime analysis has the potential to outperform the static or manual analysis for optimizing memory and execution scheduling. Hardware level optimization is also under intensive research, e.g., using FPGA and designing deep learning chips.

- Communication and consistency overheads are the major challenges for good scalability of distributed training. Cluster topology optimization could be an effective approach for reducing the communication cost. Theoretical analysis of different consistency models are essential to guide the consistency model selection and implementation.

In Chapter 3, we presented a couple of challenges and possible solutions for optimizing deep learning systems. SINGA (Version 0.x) was proposed with special focus on the architecture, which is just one of many issues. We would like to study these issues in details and implement them in SINGA Version 1.x, including memory management, operation scheduling, communication and concurrency.

6.2.2 Multi-modal data analysis with deep learning

A huge amount of multimedia data is being generated every day, e.g. social media data and e-commerce data. There are many opportunities of multi-modal data analysis with deep learning. I am keen to extend the techniques of multi-modal retrieval to multi-modal recommendation and classification. One approach is to fuse the features from different domains to generate a better representation for the object. Another approach is to exploit the data (e.g., text) from one domain to assist the analysis (e.g., feature extraction) of the other domain (e.g., images). The challenges lie in the designing of the fusion (or interaction) layer to capture the shared semantics among different domains.

References

Abadi, Martín, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Abdel-rahman Mohamed, George E. Dahl, and Geoffrey E. Hinton. 2012. Acoustic modeling using deep belief networks. *IEEE Transactions on Audio, Speech & Language Processing*, 20(1):14–22.

Bastien, Frédéric, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. 2012. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop.

Bengio, Yoshua. 2012. Practical recommendations for gradient-based training of deep architectures. *CoRR*, abs/1206.5533.

Bengio, Yoshua, Réjean Ducharme, Pascal Vincent, and Christian Janvin. 2003. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155.

Bengio, Yoshua, Patrice Simard, and Paolo Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166.

Blei, David M. and Michael I. Jordan. 2003. Modeling annotated data. In *SIGIR*, pages 127–134.

Bronstein, Michael M., Alexander M. Bronstein, Fabrice Michel, and Nikos Paragios. 2010. Data fusion through cross-modality metric learning using similarity-sensitive hashing. In *CVPR*, pages 3594–3601.

Chen, Jianmin, Rajat Monga, Samy Bengio, and Rafal Józefowicz. 2016. Revisiting distributed synchronous SGD. *CoRR*, abs/1604.00981.

Chen, Tianqi, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274.

Chen, Tianqi, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174.

Chilimbi, Trishul, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, pages 571–582. USENIX Association, October.

Cho, Kyunghyun, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder-decoder approaches. *ArXiv e-prints*, abs/1409.1259, September.

Chua, Tat-Seng, Jinhui Tang, Richang Hong, Haojie Li, Zhiping Luo, and Yan-Tao. Zheng. July 8-10, 2009. NUS-WIDE: A real-world web image database from National University of Singapore. In *CIVR'09*.

Ciresan, Dan Claudiu, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. 2010. Deep big simple neural nets excel on handwritten digit recognition. *CoRR*, abs/1003.0358.

Ciresan, Dan Claudiu, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. 2012. Deep big multilayer perceptrons for digit recognition. volume 7700. Springer, pages 581–598.

Coates, Adam, Brody Huval, Tao Wang, David J. Wu, Bryan C. Catanzaro, and Andrew Y. Ng. 2013. Deep learning with cots hpc systems. In *ICML (3)*, pages 1337–1345.

Collobert, R., K. Kavukcuoglu, and C. Farabet. 2011. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*.

Collobert, Ronan, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel P. Kuksa. 2011. Natural language processing (almost) from scratch. *JMLR*, 12:2493–2537.

Courbariaux, Matthieu, Yoshua Bengio, and Jean-Pierre David. 2014. Low precision arithmetic for deep learning. *arXiv preprint arXiv:1412.7024*.

Cui, Henggang, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. 2016. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *EuroSys*, page 4. ACM.

Dahl, George E., Student Member, Dong Yu, Senior Member, Li Deng, and Alex Acero. 2012. Context-dependent pre-trained deep neural networks for large vocabulary speech recognition. In *IEEE Transactions on Audio, Speech, and Language Processing*.

Dai, Wei, Jinliang Wei, Xun Zheng, Jin Kyu Kim, Seunghak Lee, Junming Yin, Qirong Ho, and Eric P. Xing. 2013. Petuum: A framework for iterative-convergent distributed ML. *CoRR*, abs/1312.7651.

Dean, Jeffrey, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. 2012a. Large scale distributed deep networks. In *NIPS Lake Tahoe, Nevada, United States.*, pages 1232–1240.

Dean, Jeffrey, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. 2012b. Large scale distributed deep networks. In *NIPS*, pages 1232–1240.

Dean, Jeffrey and Sanjay Ghemawat. 2004. Mapreduce: Simplified data processing on large clusters. In *(OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150.

Deng, J., W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

Donahue, Jeff, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. 2013. Decaf: A deep convolutional activation feature for generic visual recognition. *arXiv preprint arXiv:1310.1531*.

Duchi, John C., Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159.

Feng, Fangxiang, Xiaojie Wang, and Ruifan Li. 2014. Cross-modal retrieval with correspondence autoencoder. In *ACM Multimedia*, pages 7–16.

Frome, Andrea, Gregory S. Corrado, Jonathon Shlens, Samy Bengio, Jeffrey Dean, Marc'Aurelio Ranzato, and Tomas Mikolov. 2013. Devise: A deep visual-semantic embedding model. In *NIPS, Lake Tahoe, Nevada, United States.*, pages 2121–2129.

Girshick, Ross B., Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR 2014, Columbus, OH, USA, June 23-28, 2014*, pages 580–587.

Goldberg, Yoav. 2015. A primer on neural network models for natural language processing. *CoRR*, abs/1510.00726.

Gong, Yunchao, Yangqing Jia, Thomas Leung, Alexander Toshev, and Sergey Ioffe. 2013a. Deep convolutional ranking for multilabel image annotation. *CoRR*, abs/1312.4894.

Gong, Yunchao, Svetlana Lazebnik, Albert Gordo, and Florent Perronnin. 2013b. Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(12):2916–2929.

Goroshin, Rostislav and Yann LeCun. 2013. Saturating auto-encoder. *CoRR*, abs/1301.3577.

Gupta, Suyog, Wei Zhang, and Josh Milthorpe. 2015. Model accuracy and runtime tradeoff in distributed deep learning. *arXiv preprint arXiv:1509.04210.*

Hadjis, Stefan, Ce Zhang, Ioannis Mitliagkas, and Christopher Ré. 2016. Omnivore: An optimizer for multi-device deep learning on cpus and gpus. *CoRR*, abs/1606.04487.

Hinton, Geoffrey and Ruslan Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504 – 507.

Hinton, Geoffrey E., Simon Osindero, and Yee Whye Teh. 2006. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554.

Hinton, Georey. 2010. A practical guide to training restricted boltzmann machines. Technical report.

Hjaltason, Gísli R. and Hanan Samet. 2003. Index-driven similarity search in metric spaces. *ACM Trans. Database Syst.*, 28(4):517–580.

Hochreiter, Sepp. 1991. Untersuchungen zu dynamischen neuronalen netzen. *Master's thesis, Institut fur Informatik, Technische Universitat, Munchen.*

Hochreiter, Sepp and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Huiskes, Mark J. and Michael S. Lew. 2008. The mir flickr retrieval evaluation. In *Multimedia Information Retrieval*, pages 39–43.

Jia, Yangqing, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014a. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093.*

Jia, Yangqing, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. 2014b. Caffe: Convolutional architecture for fast feature embedding. In *MM'14, Orlando, FL, USA, 2014*, pages 675–678.

Krizhevsky, Alex. 2009. Learning multiple layers of features from tiny images. Technical report.

Krizhevsky, Alex. 2014. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1106–1114.

Kumar, Shaishav and Raghavendra Udupa. 2011. Learning hash functions for cross-view similarity search. In *IJCAI*, pages 1360–1365.

Lacey, Griffin, Graham W. Taylor, and Shawki Areibi. 2016. Deep learning on fpgas: Past, present, and future. *CoRR*, abs/1602.04283.

Le, Quoc V., Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Greg Corrado, Kai Chen, Jeffrey Dean, and Andrew Y. Ng. 2012. Building high-level features using large scale unsupervised learning. In *ICML*.

LeCun, Y. 1985. Une procédure d'apprentissage pour réseau a seuil asymmetrique (a learning scheme for asymmetric threshold networks). In *Proceedings of Cognitiva 85*, pages 599–604, Paris, France.

LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature*, 521(7553):436–444.

LeCun, Yann, Leon Bottou, Genevieve Orr, and Klaus Müller. 1998. Efficient BackProp. In Genevieve Orr and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade*, volume 1524 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, March, chapter 2, pages 9–50.

LeCun, Yann, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. 1996. Effiicient backprop. In *Neural Networks: Tricks of the Trade*, pages 9–50.

Liu, Dong, Xian-Sheng Hua, Linjun Yang, Meng Wang, and Hong-Jiang Zhang.

2009. Tag ranking. In *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, pages 351–360.

Liu, Wei, Jun Wang, Sanjiv Kumar, and Shih-Fu Chang. 2011. Hashing with graphs. In *ICML*, pages 1–8.

Lu, Xinyan, Fei Wu, Siliang Tang, Zhongfei Zhang, Xiaofei He, and Yueting Zhuang. 2013. A low rank structural large margin method for cross-modal ranking. In *SIGIR*, pages 433–442.

Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to information retrieval.* Cambridge University Press.

Mao, Junhua, Wei Xu, Yi Yang, Jiang Wang, and Alan L. Yuille. 2014. Deep captioning with multimodal recurrent neural networks (m-rnn). *CoRR*, abs/1412.6632.

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781.

Mikolov, Tomas, Stefan Kombrink, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. 2011. Extensions of recurrent neural network language model. In *ICASSP*, pages 5528–5531. IEEE.

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119.

Ngiam, Jiquan, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y. Ng. 2011. Multimodal deep learning. In *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, pages 689–696.

Paine, Thomas, Hailin Jin, Jianchao Yang, Zhe Lin, and Thomas S. Huang. 2013. GPU asynchronous stochastic gradient descent to speed up neural network training. *CoRR*, abs/1312.6186.

Putthividhya, Duangmanee, Hagai Thomas Attias, and Srikantan S. Nagarajan.

2010. Topic regression multi-modal latent dirichlet allocation for image annotation. In *CVPR*, pages 3408–3415.

Rasiwasia, Nikhil, Jose Costa Pereira, Emanuele Coviello, Gabriel Doyle, Gert R. G. Lanckriet, Roger Levy, and Nuno Vasconcelos. 2010. A new approach to cross-modal multimedia retrieval. In *ACM Multimedia*, pages 251–260.

Recht, Benjamin, Christopher Re, Stephen J. Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701.

Rifai, Salah, Pascal Vincent, Xavier Muller, Xavier Glorot, and Yoshua Bengio. 2011. Contractive auto-encoders: Explicit invariance during feature extraction. In *ICML*, pages 833–840.

Rosenblatt, F. 1957. The perceptron - a perceiving and recognizing automaton. *Tech. Rep. 85-460-1 (Cornell Aeronautical Laboratory, 1957).*

Rumelhart, D. E., G. E. Hinton, and R. J. Williams. 1986. Learning representations by back-propagating errors. *Nature*, 323(Oct):533–536+.

Salakhutdinov, Ruslan and Geoffrey E. Hinton. 2009. Semantic hashing. *Int. J. Approx. Reasoning*, 50(7):969–978.

Seide, Frank, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *INTERSPEECH*, pages 1058–1062.

Selfridge, O. G. 1958. Pandemonium: a paradigm for learning in Mechanisation of Thought Processes. In *Proceedings of a Symposium Held at the National Physical Laboratory*, pages 513–526, London, November. HMSO.

Shen, Heng Tao, Beng Chin Ooi, and Kian-Lee Tan. 2000. Giving meanings to WWW images. In *ACM Multimedia*, pages 39–47.

Simonyan, Karen and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556.

Socher, Richard and Christopher D. Manning. 2013. Deep learning for NLP (without magic). In *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 9-14, 2013, Westin Peachtree Plaza Hotel, Atlanta, Georgia, USA*, pages 1–3.

Socher, Richard, Jeffrey Pennington, Eric H. Huang, Andrew Y. Ng, and Christopher D. Manning. 2011. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *EMNLP*, pages 151–161.

Song, Jingkuan, Yang Yang, Yi Yang, Zi Huang, and Heng Tao Shen. 2013. Inter-media hashing for large-scale retrieval from heterogeneous data sources. In *SIGMOD Conference*, pages 785–796.

Song, Jingkuan, Yi Yang, Zi Huang, Heng Tao Shen, and Richang Hong. 2011. Multiple feature hashing for real-time large scale near-duplicate video retrieval. In *MM, 2011*, pages 423–432. ACM.

Srivastava, Nitish and Ruslan Salakhutdinov. 2012. Multimodal learning with deep boltzmann machines. In *NIPS*, pages 2231–2239.

Sutskever, Ilya, Oriol Vinyals, and Quoc VV Le. 2014. Sequence to sequence learning with neural networks. In *NIPS*, pages 3104–3112.

Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going deeper with convolutions. *CoRR*, abs/1409.4842.

Tan, Kian-Lee, Qingchao Cai, Beng Chin Ooi, Weng-Fai Wong, Chang Yao, and Hao Zhang. 2015. In-memory databases: Challenges and opportunities from software and hardware perspectives. *ACM SIGMOD Record*, 44(2):35–40.

van der Maaten, Laurens. 2014. Accelerating t-SNE using Tree-Based Algorithms. *Journal of Machine Learning Research*, 15:3221–3245.

Vincent, Pascal, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. 2008. Extracting and composing robust features with denoising autoencoders. In *ICML*, pages 1096–1103.

Waibel, Alexander H., Toshiyuki Hanazawa, Geoffrey E. Hinton, Kiyohiro Shikano, and Kevin J. Lang. 1989. Phoneme recognition using time-delay neural networks. *IEEE Trans. Acoustics, Speech, and Signal Processing*, 37(3):328–339.

Wan, Ji, Dayong Wang, Steven Chu Hong Hoi, Pengcheng Wu, Jianke Zhu, Yongdong Zhang, and Jintao Li. 2014. Deep learning for content-based image retrieval: A comprehensive study. In *ACM Multimedia*, pages 157–166.

Wang, Wei, Gang Chen, Tien Tuan Anh Dinh, Jinyang Gao, Beng Chin Ooi, Kian-Lee Tan, and Sheng Wang. 2015. SINGA: Putting deep learning in the hands of multimedia users. In *ACM Multimedia*.

Wang, Wei, Beng Chin Ooi, Xiaoyan Yang, Dongxiang Zhang, and Yueting Zhuang. 2014. Effective multi-modal retrieval based on stacked auto-encoders. *PVLDB*, 7(8):649–660.

Wang, Wei, Xiaoyan Yang, Beng Chin Ooi, Dongxiang Zhang, and Yueting Zhuang. 2015. Effective deep learning-based multi-modal retrieval. *The VLDB Journal*, pages 1–23.

Wang, Xiaogang and Eric Grimson. 2007. Spatial latent dirichlet allocation. In *NIPS*.

Weber, Roger, Hans-Jörg Schek, and Stephen Blott. 1998. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, pages 194–205.

Wei, Jinliang, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2015. Managed communication and consistency for fast data-parallel iterative analytics. In *SoCC*, pages 381–394.

Weiss, Yair, Antonio Torralba, and Robert Fergus. 2008. Spectral hashing. In *NIPS*, pages 1753–1760.

Wu, Ren, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. 2015. Deep image: Scaling up image recognition. *CoRR*, abs/1501.02876.

Yadan, Omry, Keith Adams, Yaniv Taigman, and Marc'Aurelio Ranzato. 2013. Multi-GPU training of convnets. *CoRR*, abs/1312.5853.

Yao, C., D. Agrawal, G. Chen, Q. Lin, B. C. Ooi, W. F. Wong, and M. Zhang. 2016. Exploiting single-threaded model in multi-core in-memory systems. *IEEE Trans. Knowl. Data Eng.*

Yu, Dong, Adam Eversole, Mike Seltzer, Kaisheng Yao, Zhiheng Huang, Brian Guenter, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Huaming Wang, et al. 2014. An introduction to computational networks and the computational network toolkit. Technical report, Technical report, Tech. Rep. MSR, Microsoft Research, 2014, 2014. research. microsoft. com/apps/pubs.

Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA. USENIX.

Zhang, Ce and Christopher Re. 2014. Dimmwitted: A study of main-memory statistical analytics. *PVLDB*, 7(12):1283–1294.

Zhang, Dongxiang, Divyakant Agrawal, Gang Chen, and Anthony K. H. Tung. 2011. Hashfile: An efficient index structure for multimedia data. In *ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 1103–1114. IEEE Computer Society.

Zhang, Hanwang, Yang Yang, Huan-Bo Luan, Shuicheng Yang, and Tat-Seng Chua. 2014. Start from scratch: Towards automatically identifying, modeling, and naming visual attributes. In *ACM Multimedia*, pages 187–196.

Zhang, Hao, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-memory big data management and processing: A survey. *IEEE Trans. Knowl. Data Eng.*, 27(7):1920–1948.

Zhen, Yi and Dit-Yan Yeung. 2012. A probabilistic model for multimodal hash function learning. In *KDD*, pages 940–948.

Zhu, Xiaofeng, Zi Huang, Heng Tao Shen, and Xin Zhao. 2013. Linear cross-modal hashing for efficient multimedia search. In *ACM Multimedia Conference, MM '13, Barcelona, Spain, October 21-25, 2013*, pages 143–152.

Zhuang, Yueting, Yi Yang, and Fei Wu. 2008. Mining semantic correlation of heterogeneous multimedia data for cross-media retrieval. *IEEE Transactions on Multimedia*, 10(2):221–229.