Imperial College London

Investigating the Behavior of Deep Convolution Networks in Image Recognition

Mohamed Hajaj

Submitted to the Department of Computing, Imperial College London In partial fulfilment of the requirements for the degree of Doctor of Philosophy

I declare that the work shown in this thesis is my own work unless stated otherwise. Any information taken from other sources has been appropriately referenced.

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.

Abstract

This research project investigates the role of key factors that led to the resurgence of deep CNNs and their success in classifying large datasets of natural images. Our investigation included the role of new network components, the role of the training data, and the role of data augmentation. Investigating the role of data augmentation led to the successful implementation of a deep CNN that can be trained using a variable input size, which increased the amount of allowable scale augmentation and led to much better single-view performance. Our analysis of the role of the training data shows the capabilities of deep CNNs to break down a large hierarchical dataset along the hierarchical lines into smaller components and learn all of them with great efficiency. This might help explain why deep CNN are very effective in classifying large and dense datasets of natural images which tend to have a hierarchical structure. Our investigation of core network components shows that the shared normalization statistics of BN allowed us to alter the behaviour of the network by controlling the structure of the training batches. We used this observation to obtain large conditional gain by training and testing the network using balanced batches. Finally, we were able to implement a successful multitasking network that were able to outperform the corresponding single task networks. Our model used the normalization statistics of BN to separate between the tasks, and our analysis shows that using a whole dataset per task increases the gains of the multitasking network by increasing the transfer of knowledge between the tasks.

List of Acronyms

AdaM Adaptive Moment estimation

BN Batch Normalization

CIFAR10 Canadian Institute For Advanced Research dataset

CNN Convolutional Neural Networks

CUDA Compute Unified Device Architecture

FC Fully Connected

FFT Fast Fourier Transform

GPU Graphics Processing Unit

GRU Gated Recurrent Unit

HD-CNN Hierarchical Deep CNN

HOG Histogram of Oriented Gradients

ILSVRC ImageNet Large Scale Visual Recognition Challenge

k-NN k-Nearest Neighbors

LSTM Long Short Term Memory

MNIST Modified National Institute of Standards and Technology database

MRN Multilinear Relationship Network

MT Multi Task

NIN Network In Network

PCA Principal Component Analysis

RBMs Restricted Boltzmann Machines

ReLU Rectified Linear Units

RMSprop Root Mean Square propagation

RNNs Recurrent Neural Networks

SGD Stochastic Gradient Descent

SIFT Scale Invariant Feature Transform

SPP Spatial Pyramid Pooling

ST Single Task

SVMs Support Vector Machines

t-SNE t-Distributed Stochastic Neighbor Embedding

VGG Visual Geometry Group

Contents

1 Introduction **9**

1.1 Introduction . . . 9

1.2 Main objectives: general context . . . 10

1.2.1 Investigating the role of core network components . . . 10

1.2.2 Investigating the role of data augmentation . . . 11

1.2.3 Investigating the role of the training data . . . 11

1.2.4 Investigating multitasking with deep CNNs . . . 12

1.3 Implementation guidelines . . . 12

1.3.1 Choosing the network application . . . 12

1.3.2 Choosing the convolutional network model . . . 13

1.3.3 Choosing the software tool . . . 13

1.3.4 Choosing the training data . . . 14

1.4 List of contributions . . . 14

1.4.1 Encoding Batch Structure through the normalization statistics of BN . . . 14

1.4.2 Training CNNs using a variable input size . . . 15

1.4.3 CNNs can discover hierarchical structures in the training data . . . 16

1.4.4 Multitasking with CNNs . . . 17

1.5 Thesis Structure . . . 18

2 Latest developments in deep CNNs **19**

2.1 Introduction . . . 19

2.2 Convolution Neural Networks CNNs . . . 20

2.2.1 Convolution Layers . . . 21

2.2.2 Pooling layers . . . 22

2.2.3 Linear rectified activation function . . . 23

2.2.4 Deep Convolution Neural Networks CNNs . . . 24

2.3 Related work in the literature . . . 32

2.3.1 Batch Normalization 32
2.3.2 Multi-Scale Training 41
2.3.3 ImageNet 42
2.3.4 Discovering hierarchical structure in the training data 43
2.3.5 Multitask learning with deep CNN 47

3 Encoding Batch Structure through the normalization statistics of BN 52

3.1 Introduction 52
3.2 Related Work 53
3.2.1 Standard Inference with Batch Normalization 55
3.2.2 Balanced Batches 56
3.3 Model selection 57
3.4 Statistical significance of the results 60
3.5 CIFAR10 experiments and Results 61
3.5.1 Hyper-parameter tuning 61
3.5.2 Data Augmentation 63
3.5.3 Results 63
3.6 Experiments and results for the sub-ImageNet datasets 65
3.6.1 Hyper-parameter tuning 66
3.6.2 Data Augmentation 67
3.6.3 Results 68
3.6.4 Shuffling the balanced test batches 70
3.7 t-SNE Visualization 71
3.8 Inference using the balanced batch's own statistics 73
3.9 Difficulty balancing the test batches 76
3.10 Summary 77

4 Training deep CNNs using a variable input size 79

4.1 Introduction 79
4.2 Related work 80
4.3 Implementation 81
4.3.1 Baseline network in relation to input size 81

4.3.2 Using a Variable Input Size 82
4.4 Experiments 88
4.4.1 Hyper-parameter Tuning 88
4.4.2 Results 91
4.4.3 Single-crop performance 92
4.4.4 Discussion and conclusions 95

5 Discovering Hierarchical Structures in the Training Data 97
5.1 Introduction 97
5.2 Related Work 98
5.3 Network performance vs problem size 100
5.4 Learning Hierarchical Data 102
5.4.1 Constructing a Hierarchical Dataset 103
5.4.2 Experiment 103
5.4.3 Analysing the Confusion Matrix 109
5.5 Visualization 113
5.6 Incorporating the Coarse Category Labels 115
5.6.1 Extend the output layer 115
5.6.2 Using the Category label based on the t-SNE visualization 117
5.7 Summary 119

6 Multitasking Convolutional networks 121
6.1 Introduction 121
6.2 Related Work 122
6.3 Multitasking Implementation 124
6.4 The Baseline Network 126
6.5 Experiments: Multitasking with BN 126
6.5.1 Setting the Hyper-parameters 127
6.5.2 Results 129
6.6 Training the Multitask Network on more Tasks 130
6.6.1 Testing the hypothesis of Transfer Learning 132
6.6.2 Another experiment on the Hypothesis of Transfer learning 133

6.7 Discussion 134

7 Final Conclusions **138**

7.1 Introduction 138

7.2 Summary of contributions 138

7.2.1 Utilizing batch structure through BN 138

7.2.2 Training the network using a variable input size 139

7.2.3 Deep CNNs can efficiently classify hierarchical data and large datasets . 140

7.2.4 Multitasking with deep CNNs 141

7.3 Less successful experiments 143

7.3.1 Reducing the computation cost of the convolutional layer 143

7.4 Parametric Rectified Linear Functions 144

7.4.1 Curriculum learning with deep CNNs 144

A Technical details of the network implementation **146**

A.1 Introduction 146

A.2 Optimization methods and Loss function 147

A.3 Practical Convolution Implementation 150

A.3.1 Convolution as Matrix Multiplication 151

A.3.2 Winograd's minimal filtering Convolution 154

A.4 Implementation and design principles 157

A.4.1 CUDA as an implementation tool 159

A.4.2 Experiments on Parametric Linear Rectified Functions 164

A.5 Interaction between residual connections and BN statistics 168

B Results and auxiliary tables **170**

B.1 Introduction 170

CHAPTER 1

Introduction

1.1 Introduction

This research project investigates the application of deep convolutional networks to the task of image classification. Convolutional networks were introduced by LeCun [20] more than two decades ago. After their initial success in the field of image recognition, they fell out of favour compared to linear classifiers such as SVMs which were usually built on top of few stages of hand-crafted features extracted from raw images. It wasn't until 2012, where Krizhevsy et al [9] refined the original implementation by LeCun to achieve state of the art results in the 2012 ILSVRC competition which is based on the large and diverse ImageNet dataset [4]. Because the margins between Krizhevsy's network (also called AlexNet) and the other classifiers (competitors) were so big, it was a turning point, and today the field of image recognition is dominated by deep convolutional networks. The main factors that led to the success of Krizhevsy's model (and were missing in LeCun's model) were using more convolutional layers and more parameters, using larger and more diverse training data, and using better data augmentation. Using non-saturating activation functions (e.g. $max(x, 0)$) made it possible to train deeper CNNs with more layers, using bigger datasets allowed the network to develop much better feature detectors, and using effective data augmentation methods allowed the network to have much more trainable parameters without overfitting the training data.

The objective of this research was to investigate the role of these factors that led to the resurgence of deep CNNs. These objectives included investigating the roles of core network components, the role of data augmentation, and the role of the training data. Investigating the role of core network components lead to some intersting observations about the role of Batch Normalization [2] which was introduced to speed up the training of deep feed-forward neural networks. Our results indicate that the network can utilize the structure of the training batches through the shared normalization statistics of BN. Investigating the role of data augmentation led to the successful construction of a deep convolutional network model that can be trained using a variable input image size. Training the network using a variable input size allows for more scale augmentation and our results show better performance compared to the standard implementation that is trained using a fixed input size. The role of the training data was investigated by analyzing the behaviour of these networks on hierarchical datasets. Large and dense datasets such as ImageNet [4] usually have a hierarchical structure, and measuring how deep CNNs perform on hierarchical datasets might help us understand why deep CNNs are effective in classifying large datasets with a large number of classes. Our results show that standard deep CNNs are very effective in classifying hierarchical datasets and are able to discover hierarchical structures in the training data.

As part of investigating the role of the training data, we used the ImageNet dataset as a resource to construct multiple datasets. We used these datasets to successfully train a deep

convolutional network on multiple tasks. Our multitasking model used the normalization statistics of BN to separate between the tasks. Our results show that the performance of the single multitask network significantly surpassed the overall performance of the single task networks. Our analysis indicates that the gains of the multitask network was caused by the transfer of knowledge between the tasks. Our work also investigated other aspects of these networks where the experiments were less successful in achieving their objectives. These included investigating the role of the activation function, the computational cost of the convolutional layer, and the idea of curriculum learning. Brief details about these experiment are included in different parts of this thesis, and in the final conclusions chapter. The main challenge in achieving our objectives was how to keep up with the fast pace at which these models were changing. These fast changes led to more powerful convolutional networks [10, 23, 1] with more layers and newer components, and also led to the abandonment of existing components. Therefore, our main objectives needed to take into consideration these new additions, and our results need to be applicable to these newer and more powerful models.

1.2 Main objectives: general context

As we have stated in the "introduction" section, the objective of our research is to investigate the main factors that led to the resurgence of convolutional networks. These factors included using better components which led to deeper and more powerful models, using larger and more diverse datasets which allowed these networks to develop powerful feature detectors, and using better data augmentation methods which allowed for efficient training of much bigger models. These objectives evolved to also investigate using deep convolutional networks for multitasking. This discussion will include a general context of all objectives including a brief introduction to those where our experiments and trials were less successful in achieving their objectives.

1.2.1 Investigating the role of core network components

The first objective of our study is to investigate the importance of different network components and their contribution to the success of deep CNNs. One of the components that proved to be effective in improving the performance of deep CNNs is Batch Normalization [2]. BN is a normalization technique that was intended to speed up the training of deep feed-forward networks. Our analysis of BN investigated the impact of using the shared normalization statistics of BN in the inference stage. Our results revealed an interesting property of BN which causes the network behaviour and performance to be affected by the structure of the training batches. This behaviour was also reported later in a different context by the authors of BN [7].

We also tried to improve on the work of He et al. [19] by further generalizing the

implementation of the rectified linear function and we were able to achieve some performance gains. These gains were short lived, and the introdution of batch normalization [2] erased all the gains of these more sophisticated parametric rectified functions in comparison to the performance of the standard rectified function $max(x, 0)$. Finally, we investigated reducing the computation cost of the convolutional layer. Our novel implementation was able to significantly reduce the computational cost, but its performance was inferior to the standard implementation. Later in 2017, Chollet et al. [26] were able to use depthwise separable convolutions to reduce the convolutional layer computational cost without performance loss.

1.2.2 Investigating the role of data augmentation

Deep CNNs have a large number of trainable parameters and data augmentation is important to combat overfitting and improve the performance of the network, even if the dataset is as large and diverse as the ImageNet dataset. Standard data augmentation methods are mainly based on random cropping and jitter scaling. The second objective of our research was to evaluate the possibility of improving the performance of deep CNNs through data augmentation. To achieve this goal we changed the structure of the network itself in order to increase the amount of scale augmentation. The standard network structure which is trained using a fixed input size was changed so that the network can be trained using a variable input size. In comparison to the standard implemenation, our results presented in chapter four show performance gains when training the network using a variable input size. Our implementation is similar to the SPP network [67] and the differences between the two approaches are discussed in details in the next section and in chapters two and four. We also tried the idea of curriculum learning where a large dataset of natural images was constructed and divided into multiple groups, and where the training process was also divided into multiple phases. Although this idea has worked in the past for smaller experiments [101, 58] , we were unable to apply it successfully on this large and more realistic scale.

1.2.3 Investigating the role of the training data

Deep CNNs are very effective in solving large recognition problems, and have won all the annual ILSVRC classification competitions (Based on the 1k ImageNet dataset) since 2012. The third main objective of our study is to investigate the role of the training data, and try to understand why deep CNNs are very effective in classifying large datasets. Datasets with large number of classes (such as ImageNet) tend to be dense, where similar classes can be grouped together to form an overall hierarchical structure. Understanding how deep CNNs deal with hierarchical datasets should help clarify why these networks are effective in classifying large datasets. To

achieve this goal we used ImageNet to construct a hierarchical dataset and used it to train the network. Our results indicate that standard CNNs are very effective in classifying hierarchical datasets, and our analysis show the ability of these networks to discover hierarchical structures in the training data. Our conclusions are similar to some degree to those in [68, 70], and the differences between them are discussed in the next section, and in chapters two and five.

1.2.4 Investigating multitasking with deep CNNs

To reduce the computation cost of many of our experiments, we used the ImageNet dataset to construct multiple smaller datasets with different sizes. We were able to use those multiple datasets to simultaneously train a multitask CNN. Our results show that a single multitask network can significantly outperform multiple standard single-task networks. Our analysis indicates that using a whole dataset per task is beneficial in increasing the gains of the multitask network, and that the normalization statistics of BN play an important role in separating between the tasks and therefore allowing the network to learn all of them simultaneously in a round robin fashion. In the next section, and in chapters two and six we discuss the differences between our multitasking model and similar implementations in the literature.

1.3 Implementation guidelines

This section outlines the main design principles that were used to construct the main experiments required to achieve the objectives outlined in the previous section. These principles include choosing the core recognition task that will be the focus of our investigation, choosing the appropriate convolutional network models, choosing the appropriate software tool to implement these models, and choosing the training data to train these models. The discussion in this section covers the guidelines used to carry out the experimental work of the four main experimental chapters.

1.3.1 Choosing the network application

Deep convolutional networks can be applied to a wide verity of image recognition tasks, such as image classification, image segmentation, object detection, pose-estimation, de-noising, face recognition etc. The focus of our study is on the task of image classification of natural images. Image classification can be considered as the core ingredient for more complex tasks such as object detection and image segmentation which are usually implemented on top of classification models. Models for instance segmentation [15] and object detection [93] use a deep CNN that was designed and pre-trained for image classification as a core component, and extend it by

adding what is called a network head designed specifically for the target task. Choosing to focus on the task of image classification makes it easier for us to investigate the core implementation of deep convolutional networks, and makes it easier to carry out a wide range of experiments.

1.3.2 Choosing the convolutional network model

As we have stated earlier, one of the main challenges is the continuous change to the implementation of deep CNNs used for image classification. Generally, it is harder to improve the performance of newer, deeper, and more powerful models, and therefore the conclusions that can be reached about older less powerful networks, might not hold when tested on newer ones. As an example, adding a local contrast normalization layer significantly improves the performance of AlexNet [9], while it has no impact on the newer VGG model [10] which is using three time the number of convolutional layers. For our results and conclusions to be relevant, they need to be applicable to newer and more powerful convolutional network models. For this reason we have chosen the residual convolutional network model [1] to carry out most of the experiments in this study. Although, deep convolutional network models continued to evolve, the residual model still contains the main fundamental components that proved to be essential to implement a powerful CNN. These components include BN and residual connections which allow for much deeper and more powerful models. A more detailed justification behind this choice is presented in the first experimental chapter, chapter three.

1.3.3 Choosing the software tool

The major practical consideration when designing and training deep convolutional networks is computation speed. Convolutional layers require a significant amount of computations, and therefore the model implementation needs to be fast enough for the training to be carried out in a reasonable time frame. The practical and mainstream choice is to run these models on fast GPUs which are about an order of magnitude faster than their CPU counterparts. In terms of choosing the software tool, there are many software packages (e.g. tensorflow, caffe) that make it easy to construct and train mainstream models on fast GPUs. Using these ready packages can save time when constructing and training standard models. The downside of using these packages, is the difficulty to make fundamental changes to standard network structures or standard training procedures. For more flexibility, we have chosen to implement our own models using CUDA, which is a highly parallel software platform that runs on fast NVIDIA GPUs. Implementing the model using CUDA allows us to make any necessary changes to standard implementations, and also allows us to use the latest optimized CUDA library implementations for the convolutional layer.

1.3.4 Choosing the training data

The main principle is avoiding trivial datasets such as the MNIST dataset [100]. Such datasets have a trivial error rate that can not be used to differentiate between state of the art models. The two image datasets that we have used are the CIFAR10 dataset [3], and the ImageNet dataset [4]. The CIFAR10 dataset is made of small images but it has a non-trivial error rate. The CIFAR10 dataset was only used in chapter three to measure the impact of training and testing the network using balanced batches. The dataset that we have used in most of our experiments is the ImageNet dataset. ImageNet is made up of more than 1.28 million real-size training images that are divided into 1000 classes. ImageNet was used as a standard benchmark in chapter four to measure the impact of training convolutional networks using a variable input size. ImageNet was also used in chapter three to construct smaller datasets with different sizes, in chapter five to construct a hierarchical dataset, and in chapter six to construct multiple datasets and use them to train a multitasking convolutional network. A brief review about the ImageNet dataset is presented in chapter two.

1.4 List of contributions

In section 1.2 we stated the main objectives of our research, including those where our efforts were less successful. In this section we will try to list the main contributions of our research in comparison to similar work in the literature. These contributions are divided into four main sections where each section represents one of the main experimental chapters in our thesis.

1.4.1 Encoding Batch Structure through the normalization statistics of BN

Batch normalization [2] was introduced in 2015 to speed up training of deep feed-forward neural networks, and it also leads to better final results. In deep CNNs, BN is implemented in all convolutional layers, and it normalizes each feature map across the current batch of images to have zero mean, and unity variance. Because the normalization statistics of BN are calculated using the output activations of all images in the current batch, the output activations of one image are influenced by all the other images in the batch. This implies that the structure of the batch (which is encoded in the shared means and variances) can influence the behaviour of the network. Our results in chapter three show that when a deep CNN (that implements BN) is trained using structured non-random batches, the network encodes that batch structure and uses it in the decision making process if inference is also carried out using the same batch structure. The batch structure that we investigated in chapter three is balanced batches (A balanced batch

contains a single instance per class and its size is equal to the number of classes). When a deep CNN is trained using only balanced batches, our results show significant improvements if inference is also carried out using balanced test batches. On the other hand, if inference is carried out on individual images then the results do not improve. This indicates that when both training and inference are carried out using the same batch structure, the network uses that structure in the decision-making process. These results were achieved using the CIFAR10 dataset [3], and 3 other datasets with different sizes that were uniformly sampled from ImageNet [4]. Further investigation and visualization proves that the network is using the structure of balanced test batches through the shared statistics of BN to achieve these big conditional gains.

In the literature, only the authors of BN [2] noticed this behaviour in their new update called Batch Renormalization [7]. Similar to our findinigs, they also found that the network encodes the structure of non-random batches and use that structure in the decision making process. However, they gave a negative example (borrowed from FaceNet [8]), where the performance of the network suffers if a batch normalized network was trained using non-random batches. Therefore, and to the best of our knowledge, our results are the first to report that a batch normalized network can use the structure of non-random batches to improve the performance of the network for specific batch structures (balanced batches). A detailed discussion of related literature is provided in chapters two and three. Finally, balancing the test batches is not attainable as it requires the labels of the test images. However, because of the significant conditional improvements achieved using balanced batches, further investigation can be done using unsupervised batch structures that do not require the labels of test images, or maybe a special application can be found where batches are inherently balanced.

1.4.2 Training CNNs using a variable input size

Standard convolutional networks [9, 10, 1, 23] are trained using a fixed input window size (e.g 224×224 pixels). Such models depend on data augmentation methods to implement scale augmentation. The amount of scale augmentation that can be introduced through data augmentation is limited when the convolutional network is trained using a fixed input size. Our proposed model, presented in chapter four, tries to maximize the amount of scale augmentation that can be introduced in the training phase by training the network using a variable input size. In each new iteration a different channel size is chosen randomly from a set of predefined sizes, and the network is adjusted and trained using that input size. Our results achieved using the ImageNet dataset show that the proposed implementation outperforms a standard implementation with comparable training time. Our implementation is similar to the Spatial Pyramid Pooling (SPP) network [67], and a full discussion about the differences between the two approaches is included in chapters two and four. In principle, both models are using

variable pooling after the last convolutional layer to fixate the number of weights in the network. However, the SPP network was implemented using the Caffe toolbox which restricted the scale of the implementation, while we implemented the model from scratch which allowed us to use more sizes and larger input range. What is new in our results that was not reported in [67], is that training the network using a variable input size has a much bigger impact on the single-crop (single-view) performance of the network than on the multi-crop performance. Their results do not reveal this discrepancy between the single-crop and multi-crop performance gains when training the network using a variable input size. Our results show that the single-crop performance gains are more substantial. Finally, the SPP network uses pyramid max-pooling which is more cumbersome to use for tasks like object detection where such pooling needs to be applied thousands of times after the last convolutional layer, while our implementation uses the same simple average pooling used in the standard implementation but with a variable size.

1.4.3 CNNs can discover hierarchical structures in the training data

The findings in chapter five show that a standard deep CNN is able to discover the hierarchical structure of the training dataset without any help, and without incorporating hierarchical classification into the implementation of the network. The approach we used in chapter five to analyze the capability of the network to discover hierarchical structures in the data is different from the mainstream approach. The standard approach is to analyze the confusion matrix of the classifier to see if the confusion between the classes forms hierarchical patterns. Our approach is to construct a hierarchical dataset where classes are divided into coarse categories and see if the network is able to discover this known hierarchical structure. Rather than relying only on analyzing the confusion matrix, we compared the performance of the network trained on all coarse categories with the performance of learning each category separately using a separate network. The results show that the network was able to discover the hierarchical structure of our data by achieving a recognition rate for each of these categories similar to that achieved by learning each coarse category on a separate network. A further analysis to the confusion matrix also shows that the confusion patterns follow the hierarchical divisions between the classes. We also incorporated the hierarchical structure of our data into the network implementation by adding an extra output layer that predicts the coarse category labels of the images. The position and depth of this layer was decided based on clues obtained from t-SNE visualizations which show that the confusion patterns of earlier layers are coarser than the confusion patterns of later layers. Although our results are similar to other results in the literature [68, 70], they were achieved using different approaches, and therefore they complement and further prove each other. A complete and detailed discussion on the similarities and differences between our results and similar results in the literature is provided in the "related work" section in chapter five.

1.4.4 Multitasking with CNNs

A multitasking deep CNN model is presented in chapter six which is capable of learning multiple classification tasks, where each task is represented by a dataset that is constructed by sampling classes from ImageNet. The results show that with BN the network is able to learn multiple homogeneous classification tasks by circulating through them in a round robin fashion. If BN was omitted then the network struggles to separate between the tasks, and treats all of them as a single big task. The results show that the performance of the multitask network significantly outperforms the performance of the single task networks, if tasks are uniformly sampled from ImageNet. The results also show, that the gains of the multitask network increase if the number of similar tasks learned by the network increases, and that such gains decrease if the similarity between the tasks decreases. These results indicate that the gains obtained from the multitask network are caused by the transfer of knowledge between similar tasks, and that such transfer increases if the similarity between the tasks increases, or the number of similar tasks learned by the network increases. The proposed model is similar to Caruana's model [30], in that all hidden layers are shared among all tasks and only the output layer is task specific. The main difference with Caruana's model, is that in our model each task has its own dataset, while in Caruana's model all tasks share the same inputs where each input data point is annotated with multiple labels for multiple tasks. Our results indicate that using a separate dataset per task has the advantage of increasing the gains obtained from the multitask network by increasing the transfer of knowledge between the tasks.

In our model the separation between tasks is maintained through the normalization statistics of BN (by using a different set of means and variances for each task in the inference stage). Further analysis of these statistics shows that different tasks are using similar normalization statistics in the early and middle layers and that such statistics become more discriminative in the later stages. In the Cross-stitch multitask CNN model [75] such discrimination is implemented by using a whole baseline network for each task. The relationship multitask CNN model [76] tries to model the relationship between tasks in an effort to minimize negative transfer of knowledge between the different tasks. In our model such relationship was not explicitly modelled, however the results show that the gains obtained from the multitask network can be used as an implicit indicator of the kind of relation that exists between the tasks. In terms of scope, our model was only tested on homogeneous tasks from the image recognition domain, and a comparison to the comprehensive approach in [77] shows the obstacles that need to be resolved in order for a multitask model to be able to learn multiple tasks from multiple domains. A detailed discussion about similar models in the literature is provided in chapter two, and a comparison to our model is provided in the "related work" section in chapter six. In conclusion, the main contributions include using the normalization statistics of BN to separate between multiple tasks,

and establishing a relationship between the gains of the multitask network and the transfer of knowledge between tasks, which points to the benefits of using a whole dataset per task.

1.5 Thesis Structure

This thesis is divided into seven chapters, four main experimental chapters plus the introduction chapter, the literature review chapter, and the final conclusions chapter. The next chapter (chapter two) covers the literature review required for the main experimental chapters. Chapter two starts by reviewing the main stream deep convolutional network models used for image classification, and it then reviews the literature work that is related to our experiments and findings. Chapter three presents experiments, analysis, and visualizations about training and testing deep CNNs with BN using balanced batches. Chapter four introduces a deep convolutional network model that can be trained using a variable input size (variable image size), where the results show significant improvements to the single-crop performance of the network. Chapter five investigates the behaviour of deep CNN when trained using hierarchical datasets, and how such structure can be incorporated to improve the performance of the network. Chapter six presents a multitask model where a deep convolutional network can be trained on multiple tasks and achieve results that outperform the corresponding single task networks. Finally, chapter seven provides a summary of the main conclusions and possible future work.

CHAPTER 2

Latest developments in deep CNNs

2.1 Introduction

Since 2012 and the introduction of the first deep convolutional network [20], there has been a great deal of research about the application of these networks to various tasks in the field of image recognition. This review covers the main stream convolutional network models used for image classification, and literature work that is related to our main results and findings.

The first section in this chapter reviews the basic structure of convolutional networks including the structures of the convolution layer and the pooling layer. This section also looks at the rectified linear function $max(x, 0)$ which replaces saturating activation functions in current models. Next a detailed section introduces many of the current main stream deep CNNs models, starting with 2012 Krizhevsky's model called AlexNet [9] which was a turning point in the field of image recognition. Deeper and more sophisticated models quickly followed dominating the field of image recognition, and among those presented in this review are the VGG model [10], the Residual Network model [1], the Inception model [23], and the Xception model [26]. The Residual models use identity jump-ahead connections which allow for efficient and fast training convergence when training very deep models, and they are among the most used models for various image recognition tasks. The residual network model [1] is the chosen model to be used in this research, and a detailed justification for this choice is presented in chapter three.

The remaining sections include a detailed review of related literature work. The first of these sections is about Batch Normalization [2] which reviews the implementation of batch normalization and other normalization methods such as Layer Normalization [6], Weight Normalization [5], and Batch Renormalization [7]. Most of these normaliztion methods were inspired by the success of BN and were designed to stablize and speed up the training of deep feed-forward networks and recurrent networks. The results presented in chapter three examine the effect of using shared statistics to normalize balanced batches, and the results presented in chapter six show the influnce of BN on multitasking.

The second section is about training deep CNNs using multiple image scales and reviews one approach in the literature called the SPP (Spatial Pyramid Pooling) Network [67] which is similar to our implementation presented in chapter four. Both approaches use variable pooling between convolutional layers and FC layers to fix the number of parameters in the network and allow it to be trained using a variable input size. The SPP model [67] uses variable pyramid max-pooling while our model uses variable average pooling. The other differences between the two models are in the scale of implementation and in the final conclusions reached by both approaches.

The third section looks at how deep CNNs discover hierarchical structures in training data, and it also introduces ImageNet [4] which is the main benchmark used in most of our experiments.

This section reviews hierarchical classification in shallow classifiers and in deep convolutional networks. These methods are related to the results presented in chapter five which show the capabilities of plain deep CNNs to discover hierarchical structures in the training data without incorporating hierarchical classification, which is also confirmed by the findings in [68]. Chapter five also investigates how to incorporate the hierarchical structure of the training data in the network implementation without increasing the computation cost.

The forth section is related to multitasking and reviews different implementations [73, 75, 76, 77] that are related to our model presented in chapter six. The results in chapter six show that training a single CNN with BN on multiple tasks (datasets) achieves a better performance than learning each task using a single task network. The results also show that these improvements increase by increasing the number of tasks, which points to transfer learning between the datasets.

In addition to the material covered in this chapter, a "related work" section is presented later in each of the main experimental chapters which tries to pinpoint the differences and similarities between our results and similar work in the literature. All the technical information related to the implementation details is presented in appendix A. These technical details include choosing practical (fast) implementations of the convolution layer which may include millions of 2D convolutions, choosing the right optimization algorithm to update the network parameters, and choosing a practical software and hardware platform that is suitable to train such large networks in reasonable time frames.

2.2 Convolution Neural Networks CNNs

Convolution neural networks are a special kind of feed forward neural networks that were designed for image recognition by LeCun et al. [20], and when first applied to the MNIST dataset of handwritten digits, they produced state of the art results, with error rates of less than 1%. They were the only successful deep feed forward neural networks with many hidden layers before the introduction of unsupervised pre-training in 2006. So why did convolutional networks with multiple hidden layers work well in the past, while dense fully connected deep neural networks didn't? Convolution networks are built upon the following concepts that made them very effective in dealing with images, and very effective in propagating back the error signal:

- **Local receptive fields: -** in convolution layers, units in the output channels are connected only to a small square of units in the input channels, and not to all of them. Input and output channels are organized as square matrices called feature maps. Convolution layers utilize the fact that small features in images can be extracted from a small group of

neighbouring pixels (in this case a square of neighbouring pixels), and that the values of neighbouring pixels are highly correlated (have similar values). This locality of connection facilitates the backward propagation of the error signal.

- **Shared weights: -** convolution layers apply the same weight filters to all possible locations in the input channels in a sliding window fashion to generate a single output channel. All the output units of a single output channel are produced using the same set of weight filters, resulting in a weight sharing between all the units of that output channel. This weight sharing reduces the number of trainable parameters substantially, and can result in finding features that are common throughout the input images.

- **Pooling and subsampling: -** convolution layers can be interleaved with a number of pooling layers. A pooling layer takes the output channels of a convolution layer as its input, and produces the same number of channels with reduced resolution. A square of adjacent units are pooled together and a single output unit is sampled to represent the pooled region. Simple pooling methods such as average pooling and max pooling are often used. Pooling layers introduce invariance to small deformations in images.

2.2.1 Convolution Layers

A convolution layer takes a set of input channels, and produces the same number of output channels, where a weight filter w_{ij} connects input channel i to output channel j. The outputs of a single output channel are calculated by sliding the same weight filters to all possible squares in input channels, and this operation can be implemented as a convolution between the input channels and the weight filters. If there are n input channels, then the pre-activations of a single output channel are calculated by convolving n weight filters with the n input channels and aggregating the results by element-wise summation. Then a nonlinear activation is applied to produce the final outputs of output channel j:

$$y_j = \sigma(b_j + \sum_{i=1}^{n} w_{ij} * x_i) \tag{2.1}$$

x_i is the input channel i, w_{ij} is the ith weight filter of output channel j, b_j is the bias of output channel j, y_j is the output channel j, and (*) is the convolution operation. If the size of the input channel is $n \times m$, and the size of the weight filter is $r \times c$, then the size of the output channel without padding is $(n – r + 1) \times (m – c + 1)$. Figure (2.1) shows a convolution layer with 3 input channels and 4 output channels, and using a rectified linear activation function. The figure also shows a max pooling layer following the convolution layer.

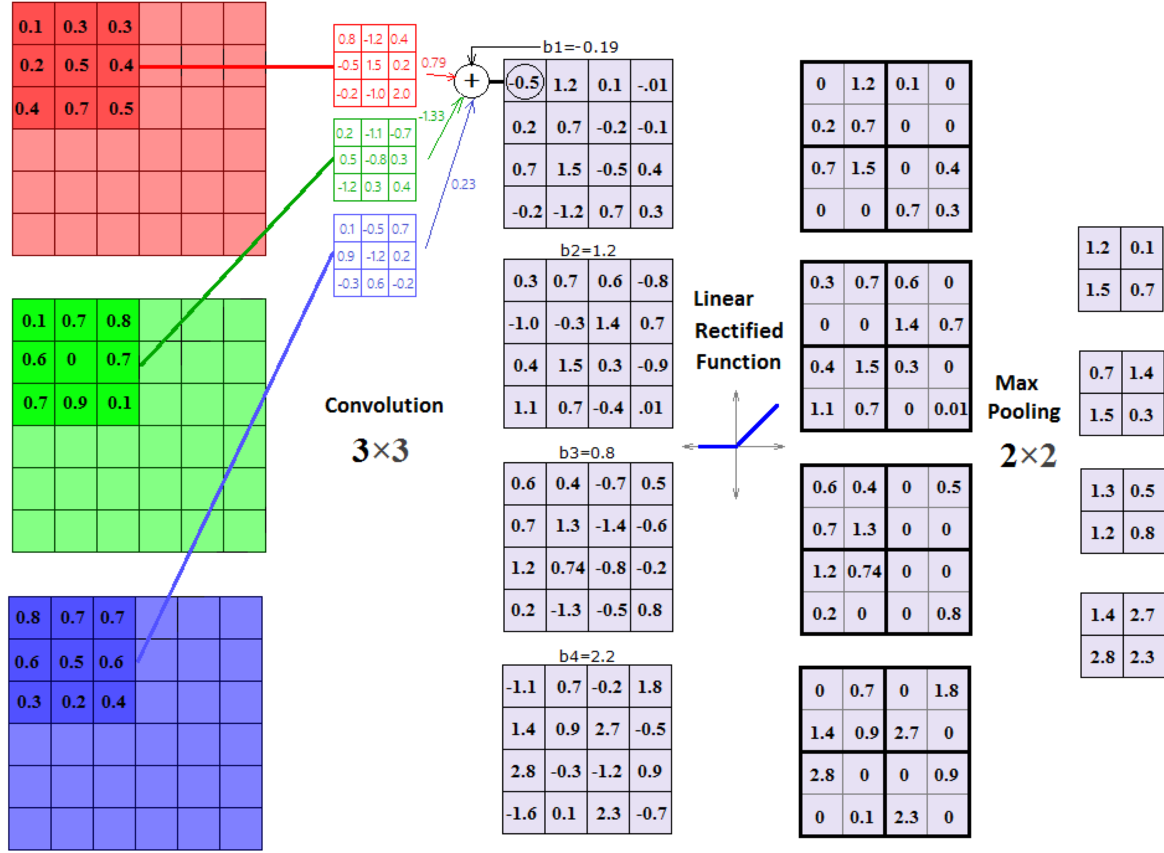

Figure 2.1: The structure of a convolution layer with 3 input layers, and 4 output layers, using a linear rectified activation function. It is followed by a max pooling layer.

2.2.2 Pooling layers

Pooling layers take the output channels produced by convolution layers as their input and produce the same number of channels with reduced resolution. Sophisticated pooling operators like stochastic pooling by Zeiler et al. [56] have been proposed, but max pooling and average pooling are still the most commonly used ones. Pooling resembles the complex cells in the mammal visual cortex, and it introduces resilience to small local changes in images. Resolution reduction is also important because it increases the receptive field of the output features which gradually transforms them from local to global with more semantic meaning. For a pooling region of size $R \times C$, the output channel y_i of the pooling layer is calculated from the input channel x_i using max or average pooling as follows:

$$y_{i,u,v} = MAX_{r,c=1,1}^{r,c=R,C}\left(x_{i,j+r,k+c}\right) \tag{2.2}$$

$$y_{i,u,v} = \frac{1}{R \times C} \sum_{r,c=1,1}^{r,c=R,C} \left(x_{i,j+r,k+c}\right) \tag{2.3}$$

Where $x_{i,j,k}$ is the value of the ith input channel at position (j,k), and $y_{i,u,v}$ is the value of the ith output channel at position (u,v). For non-overlapped pooling, $u = \frac{i}{R}$, and $v = \frac{j}{C}$. Overlapped pooling happens when the pooling stride is smaller than the pooling region. For pooling layers the number of input channels is equal to the number of output channels as pooling is done locally within each input channel and not across channels.

2.2.3 Linear rectified activation function

The fundamental difference between shallow neural networks used before 2006, and newer and deeper networks with tens even hundreds of layers, is the replacement of saturating activation functions such as the $sigmoid(x)$, and $tanh(x)$, with the simple non-saturating rectified linear function ReLU $max(x,0)$. In 2011 Glorot et al. [42] successfully used this function to train a deep feed-forward neural network without the need of unsupervised training of RBMs or autoencoders. The main difference between the Glorot & Bengio network and older feed-forward neural networks is replacing the $tanh(x)$ activation function with the $max(x,0)$ activation function. The rectified linear function had been suggested as early as the 1980's, but its potential was not realized at the time since most of the research was directed towards optimizing shallow networks with saturating functions.

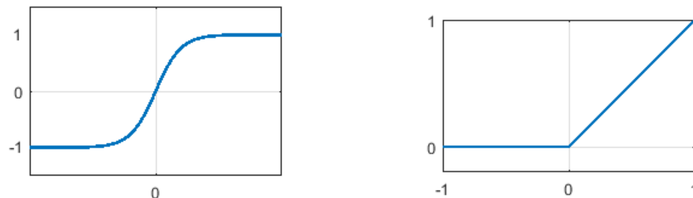

Figure 2.2: **Left:** saturating activation function tanh(x), **right:** linear rectified function max(x,0).

With SGD, multiplying the back propagated error signal by the near zero derivatives of saturating activation functions makes it harder for saturated neurons to escape the saturation region. Using non-saturating activation functions such as $max(x,0)$ eliminates this problem. The job of the rectified linear function $max(x,0)$ is to select the combination of ON neurons in the hidden layers. Once the active neurons are selected, the mapping between the inputs and outputs is linear. Changing the inputs will change the active set of neurons and therefore changes the linear mapping between the inputs and outputs. This is like approximating a complex nonlinear mapping by piecing together a very large number of linear mappings between the inputs and outputs of the network (Figure (2.3)). This allows for fast convergence when using stochastic gradient descent even for very deep networks with many hidden layers.

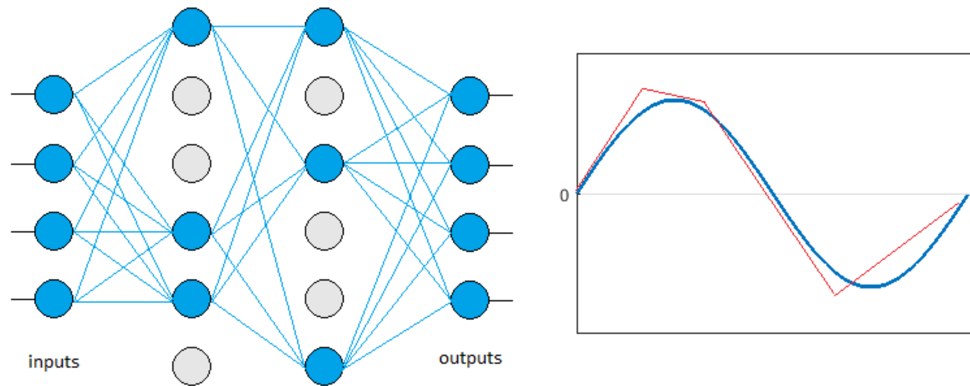

Figure 2.3: The mechanics of ANN with Rectified Linear Functions. **(Left):** each new input might change the combination of active hidden units, which is similar to approximating a nonlinear function with a linear piecewise approximation **(right)**.

2.2.4 Deep Convolution Neural Networks CNNs

One of the earliest successful convolution network models was designed by LeCun et al. 1998 [20] which had 5 weight layers divided into 2 convolution layers, and 3 fully connected layers. After each convolution layer there was an average pooling layer. The design used $tanh(x)$ as an activation function for both convolution and fully connected layers. LeCun's model achieved state of the art performance for recognizing the handwritten digits of the MNIST dataset, and later was used to verify handwritten signatures. After their initial success CNNs fell out of favor and the field of image recognition was dominated by shallow linear classifiers such as SVMs which were usually built on top of few stages of handcrafted features [78, 79, 80]. It wasn't until the seminal work by Alex Krizhevsky [9] in 2012 which led to the implementation of a deep CNN that significantly outperformed shallow linear classifiers. Today deep CNNs dominate the field of image recognition, and this section presents the most successful models used for image classifications developed since the introduction of AlexNet.

AlexNet, Alex Krizhevsky's model

In 2012 Krizhevsky [9] used the same design principles of LeCun's 1998 network to design a deep CNN with 5 convolution layers and three fully connected layers that significantly outperformed all competitors in the ImageNet ILSVCR 2012 competition. Figure (2.4, left) shows the structure of AlexNet. The model used max-pooling after the first, second, and fifth convolution layers to reduce channel resolutions. The model also used a local contrast normalization [44] layer after the first and second convolution layers, and dropout regularization [46] for the two fully connected hidden layers. However, the fundamental difference between the LeCun 1998 model and Krizhevsky 2012 model is replacing the saturating $tanh(x)$ activation function with the

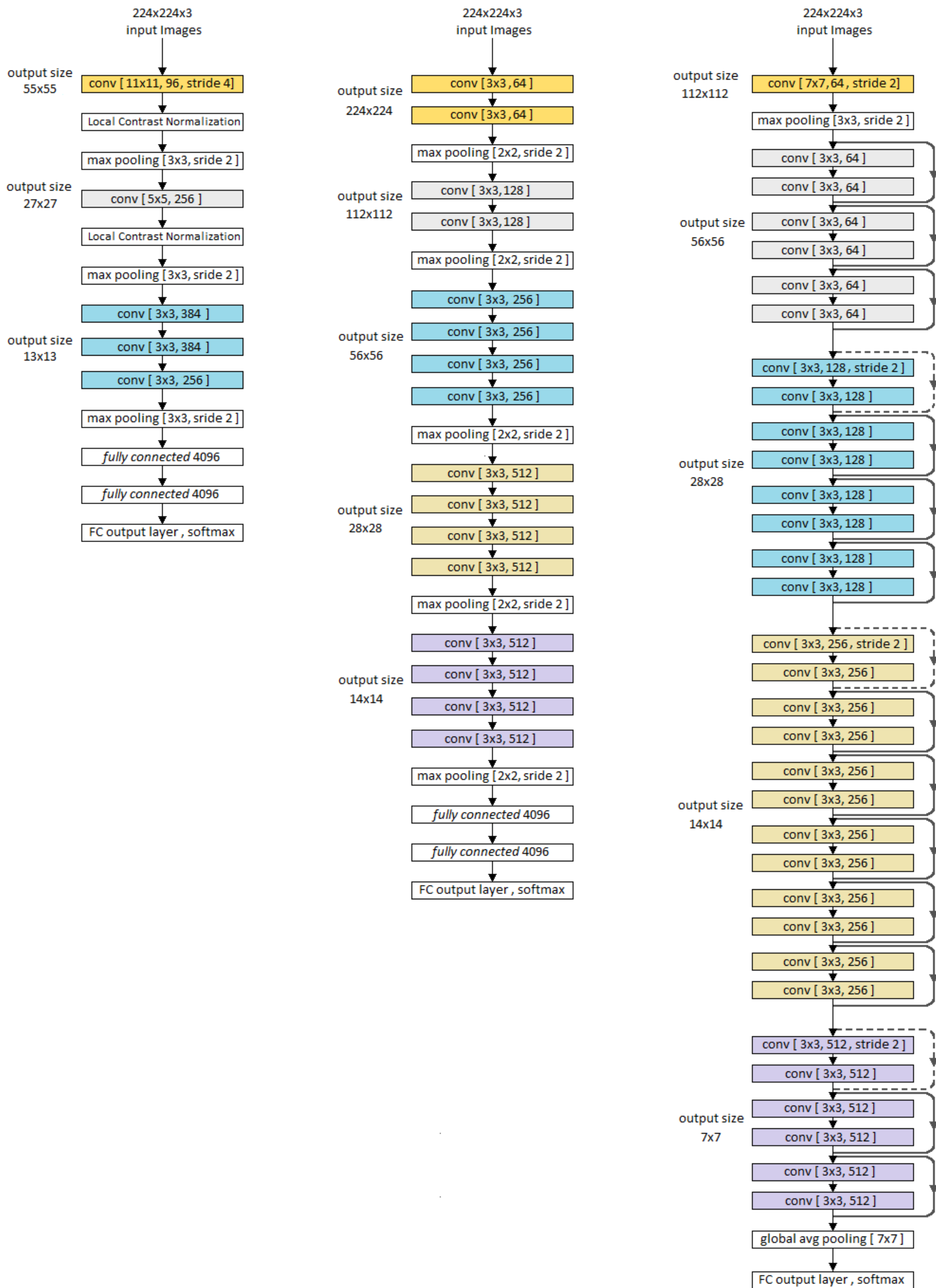

Figure 2.4: **Left:** Krizhevsky model 2012, **middle:** VGG model 2014, **right:** deep residual model 2015.

linear rectified function $max(x, 0)$. Krizhevsky showed that using $max(x, 0)$ instead of $tanh(x)$ speeds up training 6 times in a small experiment using the CIFAR10 dataset trained on a convolution network with 4 convolution layers.

The other fundamental reason behind the success of Krizhevsky's deep convolutional network model is the availability of a very large labeled dataset. The model was trained using the large and diverse ImageNet dataset [4] which has about 1.28 million training images divided into 1000 class categories. Later results and findings about transfer learning [57, 10] showed that convolutional layers develop better feature detectors when deep CNNs are trained on larger datasets. Finally, data augmentation is very important to prevent overfitting when training such large networks, even if the training dataset is as large as 1K ImageNet. The model used a basic cropping technique which scales the shorter side of the input images to a fixed value of 256 pixels, and then crops a random square of 224×224 pixels. This square, or its horizontal reflection was used as input to train the network. The model also used color augmentation which adds small random values to each of the RGB channels.

The VGG model

The VGG model by Simonyan & Zisserman [10], added on the original idea of Krizhevsky [9] and the improvements done by Zeiler & Fergus 2013 [48] to design a deeper and more powerful convolutional network. They used a more systematic approach to design their network, and they used guidelines which were widely adopted by later models. First, they only used smaller convolution filters of size 1×1 and 3×3 for the convolution layers. Their results which were verified by later models showed that using many convolution layers with small filters is more powerful than using fewer convolution layers with bigger filters. Using only small convolution filters allows the network to be very deep without increasing the number of parameters and the computation cost to train the network. Second, they doubled the number of output channels after each resolution reduction stage (i.e. pooling stage) to account for the loss of information. In addition to enhancing the network implementation, they also enhanced the data augmentation technique used in AlexNet (which was mainly based on random cropping) by adding jitter scaling, where the shorter side of the training images were scaled to a random value between 256 and 512 pixels before cropping. Their design also dropped the local contrast normalization layers used in AlexNet, as they found that such layers didn't improve the results when many convolutional layers were used. Figure (2.4,middle) shows their deepest model with 16 convolution layers and 3 fully connected layers.

The Residual Network model

Like the VGG model, the residual network model only uses small convolution filters of size 1×1 and 3×3, and doubles the number of output channels after each resolution reduction. However, with models that are built by stacking convolutional layers one after the other (like the VGG model) the network can not get very deep (e.g. hundreds of layers). Results in the literature [24, 25] show that by stacking more layers to a deep convolution network the performance saturates, and then by adding even more layers the performance starts to degrade. Such degradation however is not caused by overfitting because the training error increases. For example, if a network with 40 layers has failed to match the performance of a network with 35 layers, then the last 5 layers have failed to at least learn the identity mapping.

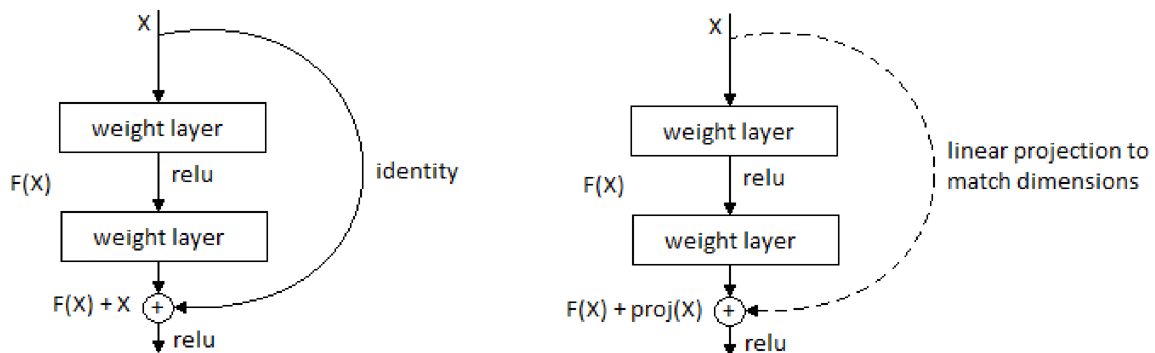

Figure 2.5: the building block for residual networks. **Left:** using identity mapping when the input X and the output of the second weight layer F(x) have the same number of channels. **Right:** linear projection to match the dimensions of the input X and F(x).

Residual networks use skip-ahead connections to solve the degradation problem in very deep convolutional networks. The building block for deep residual networks is called a residual block and is shown in figure (2.5). This block contains multiple consecutive convolution layers (in this case two). The output of the block is a combination of two signals. The first signal is the output of the last convolutional layer in the block, and the second signal is the block's input which is mapped to the block's output using an identity residual connection. These identity connections provide a highway to propagate the error signal more efficiently even for hundreds of layers which resembles the function provided by the gating mechanism in the LSTM cells of recurrent networks. If the added block is redundant then the network relies on the part of the output signal provided by the identity connection and lowers the contribution of the convolutional layers in the block. However, the results from He et al. [1] show that later stages of a very deep residual network rely on the identity mapping as the main component and a residual added value is obtained from the residual blocks by updating the weights of the convolution layers. This mechanism allows a deeper network to always improve residually on a shallower one, by starting from the identity mapping and then adding residual improvements.

Figure (2.5) shows two ways to implement the identity mapping to pass the input X to the output of the residual block $F(X)$. If the number of the channels in X is equal to the number of the channels in $F(X)$, then X is directly added to $F(X)$ to produce the output. On the other hand, if the number of channels in X doesn't equal the number of channels in $F(X)$, then X is linearly projected to match the dimensions of $F(x)$. This linear projection is implemented as a 1×1 convolutional layer with the number of input channels equal to the number of channels in X, and the number of output channels equal to the number of channels in $F(X)$. Using the residual building blocks in figure (2.5) to build deep residual networks, allows such networks to be trained using stochastic gradient decent starting from random weights, even if they have hundreds of layers.

As we have stated the residual model borrows some principles from the VGG model, but it also borrows from the Inception model [23] by aggressively reducing the channel resolution after the first convolution layer and by eliminating all hidden fully connected layers. A deep residual network starts with a single convolution layer, followed by multiple residual building blocks, followed by one fully connected layer, which is the output layer. A standard residual network will not contain any hidden fully connected layers, and therefore there is no need to use dropout to regulate such layers. A convolution with a stride of 2 was used to reduce channel resolution instead of max-pooling (except for the first layer). Finally, batch normalization [2] is used to normalize the inputs of the ReLU functions which speeds up training and allows the network to achieve better results. Batch normalization is covered in more details in the next section as it is related to the results presented in chapters three and six. Figure (2.4, right) shows a residual network with 34 weight layers.

The Inception Model

The GoogleNet convolution network by Szegedy et al. [23] uses the inception layer as its building block. The inception layer is based on a simple concept of dividing a convolution layer into multiple convolutions with different filter sizes, and adding a pooling stage to the mix. The convolutions are done in parallel, and the output channels of all the convolutions are concatenated together to form the output of that inception layer. They [23] claim that the inception layer tries to represent a sparse model with a sequence of convolutions, and also using filters with different sizes mimics processing images at different scales which might introduce scale invariance. Figure (2.6) shows a naïve implementation of the inception block.

With the naïve implementation in figure (2.6), the computation of the 5×5 convolution stage can be expensive, and the addition of a pooling stage within the inception layer can double the number of the output channels after each layer, which will cause the size of the network to grow exponentially. A simple modification is to introduce a 1×1 convolution stage before

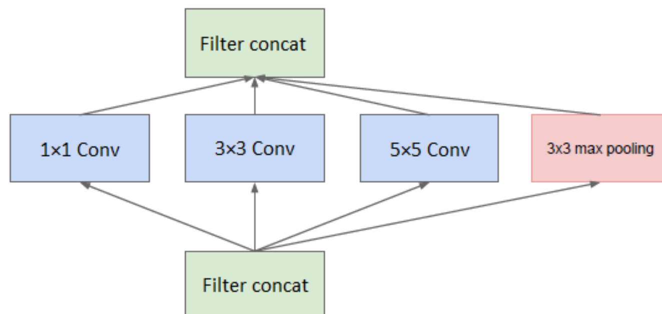

Figure 2.6: adopted from [23], Inception Layer Naïve model.

the 3×3 and 5×5 convolution stages to substantially reduce the number of input channels to those stages and therefore reduce computations significantly. Also adding a 1×1 convolution stage after the pooling stage will reduce its number of output channels and keep the size of the network from exploding. The added 1×1 convolutions act as data reduction stages by reducing the number of channels, and therefore reducing the number of computations for inception layers. Figure (2.7) shows the structure of the inception block used to implement the first model of GoogleNet [23].

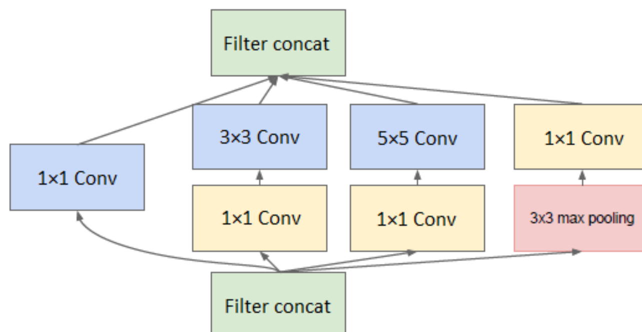

Figure 2.7: adopted from [23], Inception Layer with dimensionality reduction.

Later versions of GoogleNet [36, 37] made other modifications to the design of the inception block. The latest 2016 model used three different types of residual blocks. In the early stages of the network, the inception block shown in figure (2.8) was used. This block is very similar to the early design used in the first GoogleNet network and shown in figure (2.7). The only difference is substituting the 5×5 convolution by 2 consecutive 3×3 convolutions to increase performance. The second building block which is used in the middle stages of the network is shown in figure (2.9 left), and the third building block which is used in the later stages of the network is shown in figure (2.9 right). The difference between these new blocks and the one used in the first GoogleNet [23] is factorizing a square 2D $n \times n$ convolution filter into two consecutive or parallel $1 \times n$ and $n \times 1$ 1D filters. This reduces computations while adding more activation layers. They also used grid reduction layers (instead of max pooling or convolution with a stride

greater than 1) to reduce channel resolutions, which were implemented as a combination of max pooling and convolution.

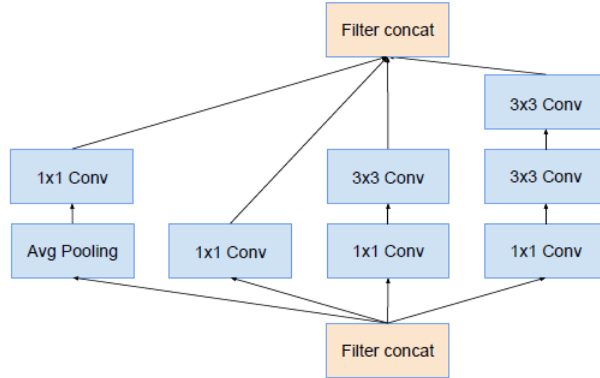

Figure 2.8: adopted from [37], Inception model A used in early stages of Inception-V4 network.

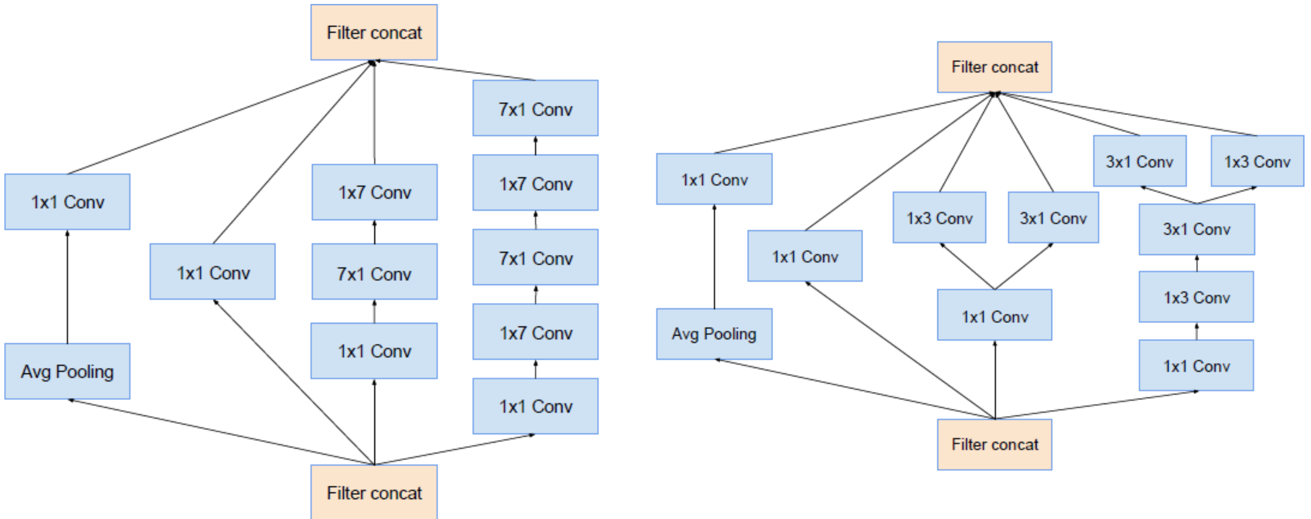

Figure 2.9: adopted from [37]. **Left:** Inception model B used in middle stages of Inception-V4 network. **Right:** Inception model C used in later stages of Inception-V4 network.

The Xception Model

In this model [26] a standard convolution layer is substituted by a depth-wise separable convolution operation. With the standard implementation of the convolution layer as shown in figure (2.1), each output channel is computed by aggregating the results of N 2D convolutions, where N is the number of input channels. If the convolution size is $n \times n$, and the input channel size is $H \times W$, and the number of output channels is M, then for a standard convolution layer the number of required computations is $N \times M \times H \times W \times n \times n$. The reason why standard convolution is computationally expensive is because it is doing spatial convolution and channel projection at the same time. In depth-wise separable convolution, spatial filtering is

separated from channel projection which leads to great reduction in computations especially if the convolution size $n \times n$ is big. Figure (2.10) shows a depth-wise separable convolution layer, where the convolution operation is divided into 2 stages. First, a depthwise spatial convolution is performed independently over each input channel, and then a pointwise convolution (standard 1×1 convolution) is performed which projects the channels produced by the depthwise convolution onto a new channel space. The total number of computations for the 2 stages is $N \times H \times W \times (M + n \times n)$, and for the same layer size it reduces the number of computations by a factor of $(M \times n \times n)/(M + n \times n)$ ($\approx n \times n$ for small convolution sizes) compared to standard convolution. Depthwise separable convolution also reduces the number of trainable parameters from $N \times M \times n \times n$ to $N \times (M + n \times n)$.

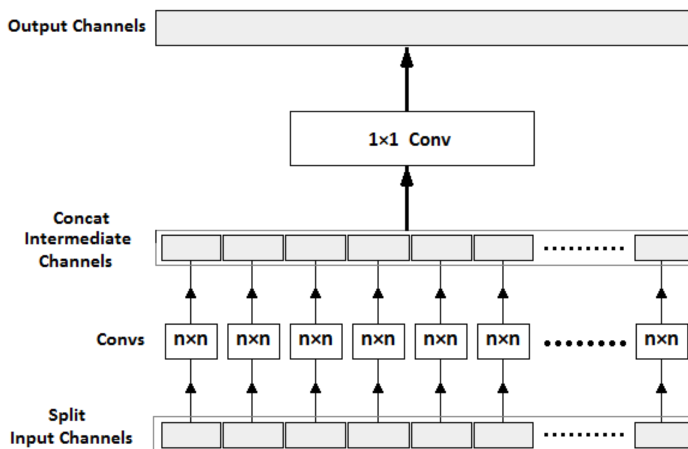

Figure 2.10: Depthwise separable convolution. A depthwise convolution is performed first, then followed by channel projection.

Depthwise separable convolution was first introduced by Laurent Sifre [27] and was used to replace standard convolution in AlexNet to obtain small gains in accuracy and large gains in convergence speed, as well as a significant reduction in model size. The Xception model [23] interprets depthwise separable convolutions as the extreme case of the Inception model presented in the previous section, and hence the name "Xception" which stands for "Extreme Inception". Looking at the inception layer in figure (2.7), the convolution layer is factorized into 3 parts (ignoring the maxpooling path) where a channel projection (1×1 convolution) is performed first, and then the resulting channels are divided into 3 groups and a standard convolution is applied to each of these groups. A Depthwise separable convolutional layer is similar to an inception layer where N channels are divided into N groups (single channel per group), and when the order of operations is reversed (channel projection is applied after spatial convolution). Also, in a depthwise separable convolutional layer the activation function is only applied once at the end after channel projection, while in an Inception layer it is applied after each convolution stage.

Despite having a conceptual connection to the Inception model [23, 36], the Xception network in [26] had a similar structure to the Residual network [1] where a standard convolutional layer is substituted by a depthwise separable convolutional layer.

Depthwise separable convolutions have the potential of significantly reducing computations without sacrificing performance. Even though it was introduced in 2013 [27], it was not widely used until it was reintroduced by the Xception model [26] in 2017. In the meantime, other ways of reducing computations which mainly relied on mathematical tricks such as using FFT to speed up convolutions with large filters [81, 82], or using low rank matrix expansions [83] were investigated. Actually, one of our earlier experiments tried to change the convolution layer in order to significantly reduce computations. Our design tried to use large filters with the same sizes as the input channels, and thus the weight filter will have only one sliding position which significantly reduces the number of multiplications. After a single element-wise multiplication between the input channels and weight filters, the output channels where computed by aggregating a neighboring square of pixels similar to standard convolution. The model was midway between fully connected layers and convolutional layers, and its performance also was in the middle between that of feed-forward networks and convolutional networks. Our abandoned model had a similar speed up to depthwise separable convolutions, but significantly worse performance than standard convolution.

2.3 Related work in the literature

The main results presented in this thesis are divided into 4 main chapters. These results are related to the topics of batch normalization, multi-scale training, hierarchical structures in the training data, and multitasking. Therefore, this review section is also divided into 4 sections to cover the work in the literature that is related to all the results and findings presented in the main chapters. This section mainly covers the theory, implementation, and findings of related work in the literature, and it provides some pointers to how this work is related to our implementations. A full discussion of how our work is related to similar work in the literature is provided later in the "related work" section of each of the main chapters. This section also covers ImageNet [4], the main image dataset and benchmark used in this thesis.

2.3.1 Batch Normalization

As it has been mentioned in the introduction chapter, batch normalization is an integral part of this research, and this section introduces the theory and implementation of BN. The review also looks at other normalization variants of BN that were introduced later to deal with some of the shortcomings of BN, such as applying the normalization to RNNs or to small batch

sizes in feed-forward networks. In BN, the activations are normalized using shared statistics computed using all the examples in the minibatch, and therefore the outputs of one training example are influenced by all the other examples in the batch. In some cases, this is considered a weakness in the implementation of BN, especially when training RNNs or using very small batch sizes. Therefore, all the subsequent normalization methods were designed so that the normalization statistics do not depend on the batch structure. However, BN is still the best performing normalization method for training deep feed-forward CNNs with medium and large batch sizes.

In chapter 3, the results show that such dependencies introduced by BN between the training examples in a single batch can be positive, and it can allow the network to learn a relationship between the members of the batch. The results show that when the training batches were arranged as balanced batches (a single instance from each class) the network uses the structure of the batch to help classify hard images in the batch. A detailed discussion is presented in the "related work" section in chapter three which relates our results and findings with related work in the literature. The remainder of this section presents the implementation details of BN, and a brief introduction to other variants of normalization.

Batch Normalization implementation

For a deep neural network, the input distribution of layer l depends on the parameters of all previous layers, and as parameters change, the input distribution of layer l will also change. Layer l will try to adapt to an input distribution that keeps changing throughout training, and that slows up convergence. This problem is called internal covariance shifts [34], and it gets worse with deeper networks that have many layers. One way to eliminate internal covariance shift is to perform whitening to the inputs of each layer, but complete whitening requires the computation of the covariance matrix $Cov[x] = E[XX^T] - E[X]E[X]^T$ and its inverse square root in the forward pass, and the derivatives of these transformations in the backward pass of the error signal. Batch normalization tries to reduce this problem by performing a simplified version of complete whitening to each layer's inputs. First, batch normalization assumes that input features are independent, and therefore can be normalized independently to have zero mean and unity variance. Second, the means and variances are calculated across the current batch, and not over the entire training data (hence the name batch normalization). For a layer with m inputs x^1, x^2,...,x^m , each input feature x^k is normalized using the equation: -

$$\hat{x}^k = \frac{x^k - E(x^k)}{\sqrt{var(x^k) + \varepsilon}} \tag{2.4}$$

where ε is a small value to prevent dividing by zero, and the expectation is performed over the current batch. These important simplifications make it possible to integrate the normalization step in the stochastic gradient decent algorithm used to update the network parameters. The algorithm also adds a linear transformation after the normalization step to restore the expressive capabilities of the network as follows: -

$$y^k = \gamma \, \hat{x}^k + \beta \tag{2.5}$$

where γ and β are trainable parameters, and if $\gamma = \sqrt{var(x^k)}$ and $\beta = E(x^k)$, then the output y^k can be restored to x^k (the input feature before the normalization step). Both convolution and fully connected layers are implemented as an affine transformation followed by a nonlinear activation function, $z = g(Wu + b)$, and for both types of layers, batch normalization is applied after the affine linear transformation and before the nonlinear activation. Therefore, the inputs to the activation function ($x = Wu + b$) are the ones that are normalized and not the inputs of the layer u. The justification of this implementation is that u, the layer's input, is the output of the previous nonlinear activation function and its shape will change during training, and constraining its first and second moments will not eliminate internal covariance shift. On the other hand, $Wu + b$ will probably have a symmetric Gaussian-like distribution, that if normalized will produce more stable activations. In addition to being done across the current batch, the normalization for convolution layers is also done across the entire output channel to ensure that all activations of the same feature map are normalized using the same mean and variance. The entire implementation of the backpropagation equations is explained in the next section, and it can be found in [2].

The network used in this study is fully convolutional, and therefore batch normalization will only be applied to convolution layers. The batch normalization training algorithm for a convolution layer can be summarized as follows:

- For each output channel and across the current batch a mean and variance is calculated and used to normalize all activations of that channel across all the images in the batch. If the channel size is $h \times w$, and the batch size is m images, and $\acute{m} = m \times h \times w$, then the mean and variance are calculated using:-

$$\mu = \frac{1}{\acute{m}} \sum_{i=1}^{\acute{m}} x_i \tag{2.6}$$

$$\sigma^2 = -\mu^2 + \frac{1}{\acute{m}} \sum_{i=1}^{\acute{m}} x_i^2 \tag{2.7}$$

- Each location of the output channel is normalized using the mean and variance from the previous step using equation (2.4).

$$\hat{x}^k = \frac{x^k - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

- After normalization, the linear transformation of equation (2.5) is applied to each output value using the two parameters γ and β, to restore the expressive capabilities of the network.

$$y^k = \gamma \ \hat{x}^k + \beta$$

- In the backward phase of updating the network parameters using gradient descent, the error signal $\frac{\partial loss}{\partial y_i}$ at the output y_i needs to be propagated back to calculate the error signal $\frac{\partial loss}{\partial x_i}$ at the input x_i, and to calculate the error signals $\frac{\partial loss}{\partial \beta}$ and $\frac{\partial loss}{\partial \gamma}$ needed to update the new parameters γ and β.

$$\frac{\partial loss}{\partial \beta} = \sum_{i=1}^{\acute{m}} \frac{\partial loss}{\partial y_i} \quad (2.8)$$

$$\frac{\partial loss}{\partial \gamma} = \sum_{i=1}^{\acute{m}} \frac{\partial loss}{\partial y_i} \hat{x}_i \quad (2.9)$$

$$\frac{\partial loss}{\partial x_i} = \frac{\gamma}{\sqrt{\sigma^2 + \varepsilon}} \frac{\partial loss}{\partial y_i} - \frac{(x_i - \mu)\gamma(\sigma^2 + \varepsilon)^{\frac{-3}{2}}}{\acute{m}} \sum_{k=1}^{\acute{m}} \frac{\partial loss}{\partial y_k}(x_k - \mu) - \frac{\gamma}{\acute{m}\sqrt{\sigma^2 + \varepsilon}} \sum_{k=1}^{\acute{m}} \frac{\partial loss}{\partial y_k} \quad (2.10)$$

During the training phase, the images are thoroughly shuffled so that a certain image will be presented to the network with a different batch of images each time. This process eliminates any unnecessary correlation that can be inferred by the network if each image is always presented with the same set of images, in all training epochs. Shuffling the images improves the accuracy of the network by about 1% as reported in [2].

Fixed statistics are calculated at each layer by averaging the means and variances of all the training batches, and these statistics are used in the inference stage instead of using the statistics of the test batch itself. This makes the inference deterministic and depend only on the image itself. In our implementation we use the balanced test batch's own means and variances in order to show how the behaviour of the network can be influenced through these shared statistics.

The claims of the authors [2] about BN were investigated by F.Schilling [103] in his masters thesis. F.Schilling investigated the convergence behaviour of batch normalized deep CNNs versus non-normalized networks. He used a VGG model [10] with 10 convolutional layers tested using 4 datasets of small images (MNIST, SVHN, CIFAR10, and CIFAR100). He examined the impact of BN on the convergence speed and classification accuracy of the network, as well as its impact on the selection of the main hyper-parameters. They reached the following conclusions which confirm to some extent with the claims of the authors [2] and with our observations about BN: BN makes the network less sensitive to the choice of the learning rate, where higher learning rates can be used to achieve faster training convergence and better validation and test accuracies. BN makes the network less sensitive to the choice of weight initialization method, although proper variance preserving initialization schemes [19, 21] are preferred. BN allows deep CNNs to use saturating non-linearities (e.g. $tanh(x)$), but their performance is still inferior to ReLUs. Adding BN erases any advantages of using more sophisticated versions of the ReLUs, such as the exponential linear units ELUs [102], which is a behaviour we have noticed with parametric ReLUs. BN does not work well with dropout regularization, and that BN acts as a regularizer and exhibits some of the same properties of dropout without the cost of slowing down the convergence of the network. Finally, they found that the selection of the batch size has an impact of the performance of a batch normalized deep CNN, and he concluded that small sizes work better for all 4 datasets. The limitation of Schilling study (as stated in his thesis [103]) is that as he investigated the impact of BN on the choices of one hyper-parameter, all other hyper-parameters were kept fixed. He stated that this was a practical choice to finish the work in a limited time fame. For example in all our experiments we found that the learning rate should be increased using the same factor used to increase the batch size (similar to the findings in [18]), while Schilling used a fixed learning rate as he tested different batch sizes. The findings of Schilling's thorough investigation agree with our observations about BN, expect on the convergence speed improvements of BN which was small for the MNIST, SVHN, and CIFAR10 small datasets, and somewhat expectable for the CIFAR100 medium dataset. This implies that the convergence speed improvements of BN are more noticeable when training the network using larger datasets made of real sized images such as ImageNet.

When BN is used with RNNs, shared statistics across the current batch need to be calculated at each time step, and that can be tricky especially if the batch is made up of sequences with variable lengths [88]. Also, because standard inference is done using fixed statistics computed from the training sequences, it is tricky to use these fixed statistics if some of the test sequences are longer than the longest training sequence. The performance of BN is also sensitive to the size of the batch. When batch sizes are very small the variance between the statistics used at each training step can be very high which can harm the performance of the network [6, 7]. The following sections show how BN can successfully be applied to RNNs, and also shows alternative

normalization methods that are more suitable for RNNs, and for small batch sizes.

Batch normalization for RNNs

In simple RNNs, the hidden state output of the current time step h_t is an affine transformation of the hidden state of the previous time step h_{t-1}, and of the current input x_t. A RNN can have multiple stacked hidden layers (depth in space) and can be trained on input sequences of a certain maximum length(depth in time). However training such basic networks using SGD is difficult due to the well known problem of vanishing/exploding gradients [84]. More practical implementations such as GRU [87], and LSTM [85] are often used. Equations (2.11, 2.12, 2.13) show the basic implementation of a LSTM cell, where a hidden RNN layer is made up of multiple LSTM cells. A LSTM cell has an additional memory cell state c_t whose update is nearly linear which allows the gradient to flow back through time more easily.

$$\begin{pmatrix} f_t \\ i_t \\ o_t \\ g_t \end{pmatrix} = W_h h_{t-1} + W_x X_t + b \tag{2.11}$$

$$c_t = \sigma(f_t)\ c_{t-1} + \sigma(i_t)\ \tanh(g_t) \tag{2.12}$$

$$h_t = \sigma(o_t)\ \tanh(c_t) \tag{2.13}$$

Earlier attempts [88] to apply batch normalization to RNNs were not successful. Their results show that when BN is applied jointly to both the input-to-hidden transformation and to the hidden-to-hidden transition of the network $BN(W_h h_{t-1} + W_x X_t)$, it hurts the training procedure. On the other hand, when it is only applied vertically to input-to-hidden transformation $BN(W_x X_t)$ it speeds up training but it overfits the training data. Their reasoning behind the failure of applying BN for RNNs is because of vanishing/exploding gradients due to repeated rescaling (using the same adaptive gain γ for all time steps).

Later Cooijmans et al [66] showed that BN can successfully be applied to speed up the training of RNNs by applying BN both vertically to the input-to-hidden transformations and horizontally to the hidden-to-hidden transitions. They showed that it is important to follow these guidelines:- First, apply BN separately to input-to-hidden transformations, and to the hidden-to-hidden transitions $BN_1(W_h h_{t-1}) + BN_2(W_x X_t)$ rather than applying the normalization to the combined output. Second, initialize the adaptive gain parameter γ to a small value (0.1 instead of 1.0), and their experiments show that when a higher value of γ is used the gradient vanishes much quicker as it is propagated back in time. They think that such initialization keeps

the value of the derivatives of the activation function $tanh(x)$ near 1.0 especially in the early stages of training which reduces the problem of vanishing gradients. To deal with variable length sequences they carefully pad shorter ones, and at inference they repeatedly use the statistics of the last time step of the longest training sequence. Their results show the normalized network outperforming the baseline network (without normalization) in a variety of applications.

Layer Normalization

Layer normalization (Ba et al [6]) was inspired by batch normalization and was mainly proposed to be used with recurrent neural networks. Here the means and variances are computed separately for each training example (image, sentence etc.) in the current batch using all the pre-activations of a single hidden layer. Therefore, the output activations of one training example are not influenced by the other training examples in the minibatch. This means that unlike batch normalization, layer normalization doesn't require calculating fixed normalization statistics from the training data to carryout inference, and thus it performs the same computations in the training stage and inference stage. Normalizing all hidden activations using a single mean and variance (per example) might restrict the learning capabilities of the network, and to counter that, layer normalization (like batch normalization) gives each hidden unit (neuron) its own adaptive bias and gain parameters (γ_i, β_i) which are applied after the normalization but before the non-linearity. The second row in table (2.1) summarizes the layer normalization equations used to calculate the means and variances, and to perform the normalization step for a fully-connected hidden layer.

	mean	variance	Normalization
Batch Normalization	$\mu = \frac{1}{B}\sum_{j=1}^{B} a_{ij}$	$\sigma^2 = \frac{1}{B}\sum_{j=1}^{B}(a_{ij} - \mu)^2$	$a_{ij} = \gamma_i \frac{a_{ij}-\mu}{\sigma} + \beta_i$
Layer Normalization	$\mu = \frac{1}{H}\sum_{i=1}^{H} a_{ij}$	$\sigma^2 = \frac{1}{H}\sum_{i=1}^{H}(a_{ij} - \mu)^2$	$a_{ij} = \gamma_i \frac{a_{ij}-\mu}{\sigma} + \beta_i$
Weight Normalization	$\mu = 0$	$\sigma = \|W_i\|_2$	$a_{ij} = \gamma_i \frac{a_{ij}}{\sigma} + \beta_i$

Table 2.1: The equations of Batch, Layer, and Weight normalization for a single fully-connected hidden layer. B is the batch size, H is the hidden layer size, a_{ij} is the pre-activation of ith unit in the hidden layer and for the jth image in the current batch.

Because each training example is normalized separately, it is easier to use layer normalization with RNNs when the network is trained on sequences with variable length, or when carrying out inference on a long sequence that is longer than any of the training sequences. Their results show that layer normalization is effective in stabilizing the hidden state dynamics in recurrent neural networks. The normalization step is normally applied to both input-to-hidden and

hidden-to-hidden transformations of the LSTM or GRU cell (eq.(2.11)), but they also applied it to parts of the gating mechanism that updates the cell state (eqs(2.12, 2.13)). Their results show layer normalization outperforming batch normalization [2] and weight normalization [5] in some tasks that require training RNNs.

The results also show layer normalization being competitive to batch normalization when training fully connected feed-forward neural networks (outperforms BN when using very small batch sizes e.g. 4) but failing to perform when applied to convolution neural networks (the focus of this research). Their reasoning behind this behavior is that in fully connected layers, all the hidden units tend to make similar contributions to the final prediction and re-centering and rescaling the pre-activations to a hidden layer works well. However, the assumption of similar contributions is no longer true for convolutional neural networks. The large number of the hidden units whose receptive fields lie near the boundary of the image are rarely turned on and thus have very different statistics from the rest of the hidden units within the same layer.

Weight Normalization

Weight normalization is another normalization method that was inspired by batch normalization but like layer normalization it does not introduce any dependencies between the examples in a minibatch. This means that it can easily be applied to recurrent neural networks. In weight normalization the pre-activation output of each hidden neuron is divided by the L2 norm of the input weights to that neuron and then it is multiplied by an adaptive gain and then it is added to an adaptive bias. This is similar to normalizing the signal using $\mu = 0$, and $\sigma = \|W\|_2$ (third row in table(2.1)). Because the input weights are used to normalize the pre-activation signal instead of using the statistics of the pre-activation signal itself, weight normalization does not guarantee fixing the distribution of the pre-activations to have zero mean and unity variance, and therefore it is more sensitive to parameter initialization.

The results [5] show that when weight normalization is applied to RNNs and generative models it always speeds up training and produces better results compared to the baseline model without normalization. The results also show that weight normalization provides most of the speed up provided by batch normalization when training CNNs but with a lower computation cost as the weight filter sizes of convolutional layers are much smaller compared to the sizes of the convolutional layer outputs. Therefore, weight normalization can be considered as a cheaper alternative to batch normalization with a small loss of accuracy when training deep CNNs. Finally, their results show that weight normalization works well with noise-sensitive applications such as deep reinforcement learning, and their reasoning behind that is because using the input weights (which change very gradually during training) for normalization is less stochastic and it introduces less noise than using the statistics of the signal itself.

Another normalization method which is similar to weight normalization is called normalization propagation [65]. The idea behind it is, that once the input signal is normalized, that normalization can be propagated throughout all the hidden layers without applying an expensive normalization such as batch normalization. However, the end product is a slightly more complex version of weight normalization, which can be effectively used to train CNNs.

Batch Renormalization

This normalization method [7] is an update to the original BN [2], where the authors try to broaden the application of BN, so that it can be used with smaller batch sizes and with non-random (non-i.i.d.) batches. Training the networks on non-random batches is very closely related to the results about using balanced batches, and it will be discussed in more detail in the “related work” section in chapter three. In the forward pass, the new version of BN uses the running average of the means and variances (used to measure the validation error) to normalize the pre-activations instead of using the means and variances of the batch itself. In the backward pass of the error signal, only the contributions that correspond to the means and variances of the batch itself are updated. The authors say that this inconsistency between the forward pass and the backward pass of the SGD does not disturb the convergence of the training process because the expectations of the batch's means and variances converge to the running averages of these statistics. If the running average statistics are μ and σ , and the current batch's own statistics are μ_B and σ_B, then the pre-activation signals are normalized as follows:-

$$\hat{x_i} = \frac{x_i - \mu}{\sigma} = \frac{x_i - \mu_B}{\sigma_B} \cdot r + d \quad where \;\; r = \frac{\sigma_B}{\sigma} \;\; and \;\; d = \frac{\mu_B - \mu}{\sigma}$$

In the forward pass the variables r and d will be used in the normalization, but in the backward pass they will be considered as constants, and only the batch's own statistics μ_B and σ_B will be updated. r and d act as correction terms that bridge the gap between the batch statistics and the population statistics. However, they find it necessary to start the training with just BN without correction ($r = 1$ and $d = 0$), and then gradually ramp up the amount of allowed correction. The resulting normalization method, has a similar performance to BN when used to train deep CNNs with medium and large batch sizes, and performs better than BN when used with very small batch sizes. Because the normalization effectively relies on the running average statistics and not on the batch statistics, its performance is not affected by how training batches are structured, and it performs the same when trained on random batches or on non random batches (this is related to our results and there is more discussion about it in chapter three). Finally, when used with RNNs, shared statistics (μ_B, σ_B)are still needed at each time step, and therefore the new implementation still suffers from the same limitations as BN when applied to RNNs.

2.3.2 Multi-Scale Training

One of the interesting properties of convolutional layers is that the number of parameters in these layer is independent of the size of the input. This allows the last convolution layer of a convolutional network trained on a certain input size to produce very good high-level features when tested on a different size. In practice however, deep CNNs are trained using a fixed input size (e.g. 224×224), because these networks use fully connected layers in the latest stages. Therefore, if there is a way to keep the size of the input to fully connected layers constant, while allowing the size of the input to the convolutional layers to vary, then the network can be trained on variable input sizes. One such implementation is presented in chapter 4, which shows a significant improvement in the single view (crop) performance. In this section we review a similar implementation in the literature called Spatial Pyramid Pooling [67], and in chapter 4 a discussion about the differences between the two approaches is also presented.

Spatial Pyramid Pooling

In order to keep the size of the input to the fully connected layers constant, He et al [67] used a pyramid pooling stage after the last convolution layer. In their implementation the pyramid pooling stage will always produce a fixed-size output vector regardless of the size of the input. This allowed the network to be trained using variable input-channel sizes, while keeping the number of parameters in the fully connected layers (therefore in the network) constant. They used a 4-level spatial pyramid with the number of bins equal to 1×1, 2×2, 3×3, and 6×6. At each of these levels a max-pooling operator is used to produce a single output per bin. Therefore, each of the output channels of the last convolution layer will produce 50 outputs (after it passes the pyramid pooling stage) regardless of its size. This is simply done by keeping the number of bins constant at each level and allowing the size of the bin to vary according to the size of the channel.

For ImageNet classification the authors [67] used AlexNet [9], Zeiler and Fergus [48], and Overfeat [57] netwoks as a baseline models. All these models have 3 fully connected layers, and the pyramid pooling stage was added just before the first hidden fully connected layer replacing the last max-pooling in these models. The new model is called SPP-Network. The SPP-Network is trained as a standard network using a fixed input-channel size of 224×224, and also trained using two sizes of 180×180 and 224×224. In both cases the SPP-network improved the performance of the baseline network. The reason why the SPP-network trained using a standard single input-channel size outperformed the baseline network (which also trained using the same fixed size) is because of pyramid pooling. Pyramid pooling pools the feature maps at different scales which by itself makes the network more robust to object deformation. The performance of the SPP-network is further improved when the network is trained using multiple

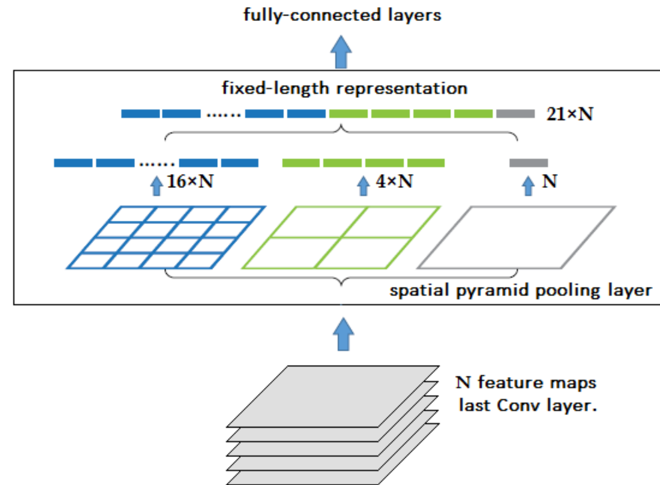

Figure 2.11: adopted from [67], Spatial Pyramid Pooling layer inserted between convolution and fully-connected layers.

input sizes (2 sizes). Our model presented in chapter 4 uses a global average pooling after the last convolution layer to keep the number of parameters constant when the network is trained using variable input-channel sizes. In our implementation however 6 different scales were used, and the residual network by [1] is used as a baseline network. The "related work" section in chapter 4 compares the two methods.

2.3.3 ImageNet

ImageNet [4] is a largescale collection of images built upon the large lexical database of English called WordNet [89]. ImageNet uses the hierarchical structure of WordNet, where each meaningful concept (possibly described by multiple words or word phrases) is called a "synonym set" or "synset". There are around 80,000 noun synsets in WordNet, and currently ImageNet populates 21841 of those noun synsets, with a total of 14,197,122 images. The aim of ImageNet is to provide the most comprehensive and diverse coverage of the image world. Because of its coverage and size, ImageNet has one of the most densely semantic hierarchies, where for example 147 categories of dogs are covered.

Images of each concept (synset) are quality-controlled and human-annotated to achieve a very high accuracy in the labeling process, which can be tricky for the more specific and accurate concepts that lie in the leaf nodes of the tree structure (e.g. Siamese cat versus Burmese cat). Also, ImageNet was constructed so that objects in images should have variable appearances, positions, viewpoints, and poses as well as background clutter and occlusions. Such diversity will lead to better recognition systems.

The first stage of constructing ImageNet involves collecting candidate images for each synset. Images were collected from the internet by querying several image search engines using all the synonyms of a synset. To obtain as many images as possible queries were expanded by using words from parent synsets, and by translating the queries into other languages. Because the average accuracy of image search results from the Internet is low, a large set of candidate images were collected. After intra-synset duplicate removal, each synset has over 10K candidate images on average. To obtain a highly accurate dataset, candidate images are filtered manually by multiple individuals through the Amazon Mechanical Turk (AMT), an online platform on which one can put up tasks for users to complete and to get paid. For each synset, users were presented with a set of candidate images and the definition of the target synset (including a link to Wikipedia), and asked to verify whether each image contains objects of the synset. An image is considered positive only if it gets a convincing majority of the votes, where harder synsets require larger number of users to get an accurate decision.

The dataset that is usually referred to as ImageNet and is used as a benchmark in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is only a small part of the whole ImageNet. This 1k ImageNet dataset is constructed by randomly sampling 1000 synsets (categories) from the leaf synsets of ImageNet. Sampling only leaf nodes assures that there will be no overlap between internal nodes (e.g. vehicle) and their decedent leaf nodes (e.g. bus). 891 out of the 1000 chosen leaf categories had 1300 training images, while the remaining 123 had between 834 and 1300 images. The selection process was biased toward choosing categories with comparable number of images to make the dataset more effective in training image recognition systems. Because the 1k ImageNet dataset was constructed using only leaf nodes, the structural hierarchy of ImageNet is rarely used. However, because the number of chosen categories is still high (1000), the hierarchical structure or part of it is still there. By visually inspecting the 1k ImageNet dataset, it is obvious that many fine categories can be grouped into coarse and then coarser categories. The mean experiment in chapter five was constructed by selecting 100 fine categories that can naturally be grouped into 10 coarse categories. The results in chapter 5 show that a deep CNN can learn to separate different coarse categories without the need of using the hierarchical structure of the fine leaf categories. Most of the experiments presented in this study were implemented using datasets sampled from this 1k ImageNet dataset.

2.3.4 Discovering hierarchical structure in the training data

Deep CNN are effective in solving large recognition problems with large numbers of class categories, and have won the ILSVRC ImageNet challenge every year since 2012. The results presented in chapter 5 show that the classification error increases at a much slower rate compared to the increase in the number of classes learned by the network. One of the reasons that might

explain such resilience toward solving large recognition problems with very large numbers of classes is transfer learning, where the network learns better feature extractors if it was trained using larger and more diverse datasets. The main results in chapter five show another aspect of these networks, which might also explain why deep CNNs are effective in solving large problems. The results show the ability of these networks to discover the hierarchical structure of the training data without any modification to the plain implementation of convolutional networks. These results may explain why there is limited interest in the literature to change these networks to incorporate the hierarchical structure of the data. Here we look at how the hierarchical structure of the data was used before the era of deep learning, and then review some attempts that try to incorporate it in the design and training of deep convolution networks. A discussion of how our results and finding are related to similar findings in the literature is presented in the "related work" section of chapter five.

Hierarchical classification before deep learning

Before deep learning, recognition systems were built by applying shallow classifiers (e.g. SVMs, k-NN) to hand crafted features (e.g. SIFT, HOG). When using SVMs for image classification to classify a dataset with multiple classes, a one-vs-all binary classifier is needed for each class category, and that is inefficient when dealing with a large number of classes. Therefore, the hierarchical structure of the data can be exploited to only compare similar classes rather than testing each class against all the other classes in the dataset. This can result in a better performance [72], better behavior (making reasonable mistakes) [70], or a speedup in training and inference [71].

Two common methods are used to construct the hierarchical structure of the data. The first option is to use a fixed structure, such as the popular lexical sematic network WordNet [89]. The other option is to use the confusion matrix produced by the classifier itself, and then apply a clustering algorithm to group similar classes (usually being confused with each other by the classifier) into coarser categories. Once the hierarchical tree is constructed, a classifier is trained at each node to decide which path to follow to the appropriate leaf node which will make the final decision about the identity of the image (for a binary tree this will be similar to a binary search).

Marszalek et al. [72] used the WordNet lexical network to construct the hierarchical tree for the PASCAL VOC'06 image dataset which contains 10 classes. In addition to the hypernymy/hyponymy relationship which is usually used to construct the hierarchy tree, they also used the holonymy/meronymy (PARTOF and MEMBEROF relationships), and they say it permits much richer reasoning. At each node N SVM classifiers were trained, where N equals the number of descendent nodes. Speeding up the computations was not a concern here as the

number of classes was small, and using an elaborate hierarchy structure (that incorporated the PARTOF/MEMBEROF relationships) resulted in performance gains.

Griffin et al. [71] applied hierarchical classification to the Caltech-256 image dataset, which has 256 class categories. A standard implementation requires 256 one-vs-all SVM classifiers. Their goal was to speed up computations 5 to 20 fold while enduring a small drop in the performance. The hierarchical tree was constructed using the confusion matrix of a standard classifier. To make it fast and less stochastic, the estimation of the confusion matrix is done using only 10 training images per class and a leave-one-out validation procedure. Then Self-Tuning Spectral Clustering [90] is used to split the confusion matrix into two groups, so that the confusion is minimum between the resulting groups. This splitting procedure is repeated until each group represents a single class, resulting in a binary tree structure. Once the hierarchy structure is constructed, a single binary SVM classifier is trained at each node using only a small fraction of the training images.

Deng et al. [70] show the computation and performance limitations of such shallow classifiers (e.g. k-NNs, SVMs) when used with very large datasets, and showed how hierarchical classification can speed up computations. They experimented with the 10k ImageNet (10184 classes, the whole ImageNet dataset at that time, Fall 2009 release). They also experimented with smaller datasets (between 134 and 262 classes). For the 10K ImageNet, 10184 one-vs-all SVM classifiers were needed, with 1 hour to train a single classifier (more than 1 year to train all of them) using a single machine (2.66GHz Intel Xeon CPU). They also found that while these one-vs-all SVMs outperform simpler classifiers such as k-NN on smaller datasets, they significantly underperform k-NN when used with the large ImageNet dataset. The brute force linear scan k-NN also needs about a year to run through all test images for the 10k ImageNet. This shows how inadequate these methods are in dealing with large datasets such as ImageNet compared to deep CNNs. They used WordNet to construct a hierarchical classifier that achieved speed up similar to that achieved by Griffin et al [71].

Hierarchical classification with deep CNNs

One advantage of using CNNs for image recognition compared to one-vs-all SVMs, is that the size of the classifier (convolutional network) is almost the same regardless of the number of class categories. Adding extra classes requires only adding extra neurons in the output layer. Therefore, hierarchical classification cannot be seen as a way of speeding up computations when used with deep CNNs as is the case with one-vs-all linear classifiers. Adding to this reason, the findings presented in chapter five which show the ability of a plain convolutional network to discover such a hierarchy by itself, and the difficulty of incorporating it with CNNs, might explain the reduced of interest in the literature of doing hierarchical classification with CNNs.

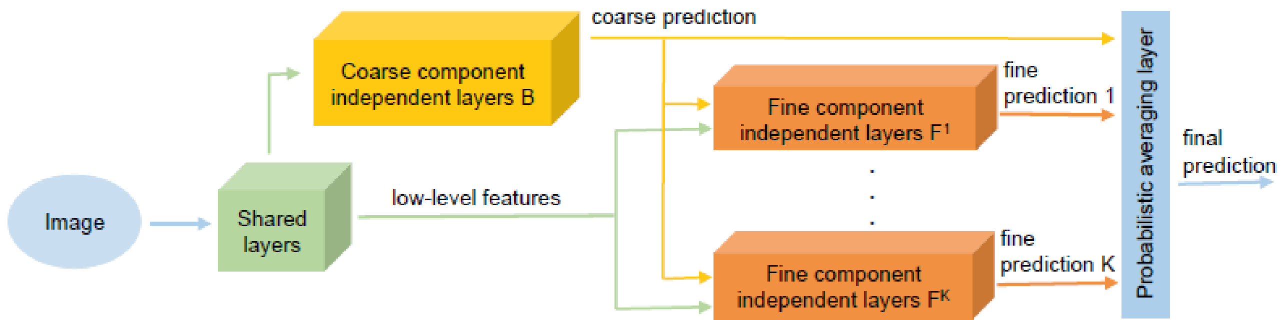

Figure 2.12: adopted from [69], shows the layout of a two-level (coarse-fine) Hierarchical convolutional network.

Here we review two attempts in the literature [68, 69] to incorporate hierarchical classification with CNNs.

It is impractical to train a separate CNN at each internal node of a hierarchical tree, and therefore the hierarchical model used with linear classifiers can not be applied to deep CNNs. Yan et al [69] used a two-level hierarchy structure, where in the first stage images are divided into N coarse categories, and in the second stage a dedicated classifier is used for each of the N coarse categories to find the class identity of the image. Both the coarse category classifier, and the N fine category classifiers were implemented using CNNs. To make such an implementation practical, all classifiers share most of the convolution layers (early layers), while each classifier has only a few dedicated layers. They experimented on the CIFAR100 [3] dataset and on 1k standard ImageNet dataset. In order to divide the dataset into coarse categories they used the confusion matrix produced by a plain CNN and applied spectral clustering to divide the data into N coarse categories, where N is a hyperparameter tuned using held-out training images. The coarse category classifier is trained on all the data, while the N fine category classifiers were trained only on the classes that belong to the corresponding coarse category. If the coarse classifier directs the test image to the wrong fine category classifier, then the test image will be misclassified. To reduce the impact of the mistakes made by the coarse category classifier, they allow fine categories to belong to multiple coarse categories based on an overlapping hyperparameter γ. With overlapping identity, all coarse categories with a confidence score exceeding a predefined threshold β (another hyperparameter) are considered, which significantly increases inference time. As a backbone convolutional network they used the 4-layers Network-In-Network model [95], and the 16-layers VGG model [10]. Their results show some improvement in performance, however with significant increase in computations.

Another study [68] titled "Do Convolutional Neural Networks Learn Class Hierarchy" reaches similar conclusions to our results presented in chapter 5. After training a CNN on the 1k ImageNet dataset, they reordered the confusion matrix according to the WordNet hierarchy. They found that the reordered confusion matrix forms non-overlapping square blocks with variable sizes

along the diagonal, which implies that the hierarchical pattern discovered by the network is in tune with the hierarchical structure used to construct ImageNet in the first place which is WordNet. Using spectral clustering also produces blocks along the diagonal which implies that confusion between the classes follows a hierarchical pattern. The other observation is that earlier stages in the network develop feature detectors that separate between large categories, while later stages develop specialized feature detectors that can separate between individual classes. They observed that by training linear classifiers on the outputs of different layers at different depths in the network and then investigating the confusion matrices produced by these classifiers. The results in chapter 5 reach similar conclusions, although using more direct t-SNE visualization on the features of individual images. They used this observation to add auxiliary unexpensive fully connected output layers at different stages of the network to produce coarse category labels. The number of these categories is chosen based on the analysis of the classification power the of corresponding convolution layers. Therefore, the hierarchical structure of the data is utilized by augmenting the main classifier rather than training separate classifiers for different parts of the data. This implementation improves the results of the AlexNet [9] by about 3%.

2.3.5 Multitask learning with deep CNN

One of the main results presented in this research is about multitask learning, where a single deep convolutional network was able to learn tasks and significantly outperforming the single task networks. With BN the network can learn multiple tasks by circulating through them in a round-robin fashion. Also, the results show that the amount of gains obtained from the multitasking network increase as the number of tasks trained on the network increases, which points to added transfer learning between the tasks. In this setup tasks are made of datasets with equal number of classes and were randomly sampled from ImageNet. Therefore, these tasks are homogenous, with similar types of inputs and outputs. Also, each task has its own dataset, which is similar to the setup used by Multilinear Relationship Networks [76], and is different from the model used by Caruana, where the inputs of a single dataset are augmented with multiple labels. In this section we review different implementations of multitask networks in the literature, and in the "Related Work" section in chapter 6, we discuss the differences between these models and our model, and also discuss the relationship to transfer learning.

Caruana's Model

One of the early models where multiple tasks are learned concurrently using a single neural network is the Caruana model [30] 1998. Caruana found that learning multiple auxiliary tasks in addition to the main task helps the network to improve its performance on the main task.

In his multitasking implementation each input data point (image, speech, words, etc.) has multiple labels, one for each task, and the combined error signal propagated back from all labels is used to update the network weights. Therefore, in Caruana's model there is only one dataset, and that dataset is augmented with multiple labels for multiple tasks. In our model as in the Multilinear Relationship Networks [76] discussed later, each task has its own dataset.

The training signals of the extra tasks serve as an inductive bias, that makes the network prefer shared representations that work for all tasks. Adding these tasks allows the network to utilize rich sources of information available in the domain to solve real-world problems. One of the cases where using auxiliary tasks is a good choice is when some of the input features are not available at decision time. One of the experiments by Caruana [30] that explains this scenario is related to Pneumonia Prediction. The task is to decide based on preliminary tests if a patient needs hospitalization or not. Caruana used the Medis Pneumonia Database [91] which contains 14,199 pneumonia cases collected from 78 hospitals in 1989. This dataset contains 35 extra lab results that are only taken once the patient is admitted. Therefore, these lab results are not available when the decision about admitting a patient needs to be made, and therefore cannot be used as extra inputs. Caruana used these extra lab results as extra auxiliary tasks to improve the performance of the network.

Various adaptations of Caruana's model have been used by others [31, 32, 33] where the differences reflect the specific application of the network, and how the structure is tuned and divided between task specific (output layers and maybe late hidden layers) and shared layers (early hidden layers). In the field of deep convolution networks, models used for object detection such as the R-CNN family [92, 93, 13] estimate the class identity of each object, and also the coordinates of the bounding box that surrounds that object. Therefore, each image is augmented by the class labels, and bounding box coordinates. Models such as the Mask R-CNN [15] estimate the class identity, the bounding box coordinates, and also the segmentation mask for each object in the image.

Cross stitch networks

The standard way of implementing multitasking in convolution networks is to share earlier layers between all tasks and make later layers task specific. At which layer should tasks diverge depends on the tasks. However, in practice most of the layers are shared and only few later stages are task specific [30, 31, 32, 33]. The cross-stitch network [75] takes a different approach, where each task has its own complete baseline network, but the final output of each task is a linear combination of all the baseline networks. The authors [75] only experimented with two tasks and used AlexNet as a baseline the network [9]. The two baseline networks were cross stitched after each of the 3 pooling layers and the 2 hidden fully connected layers as shown

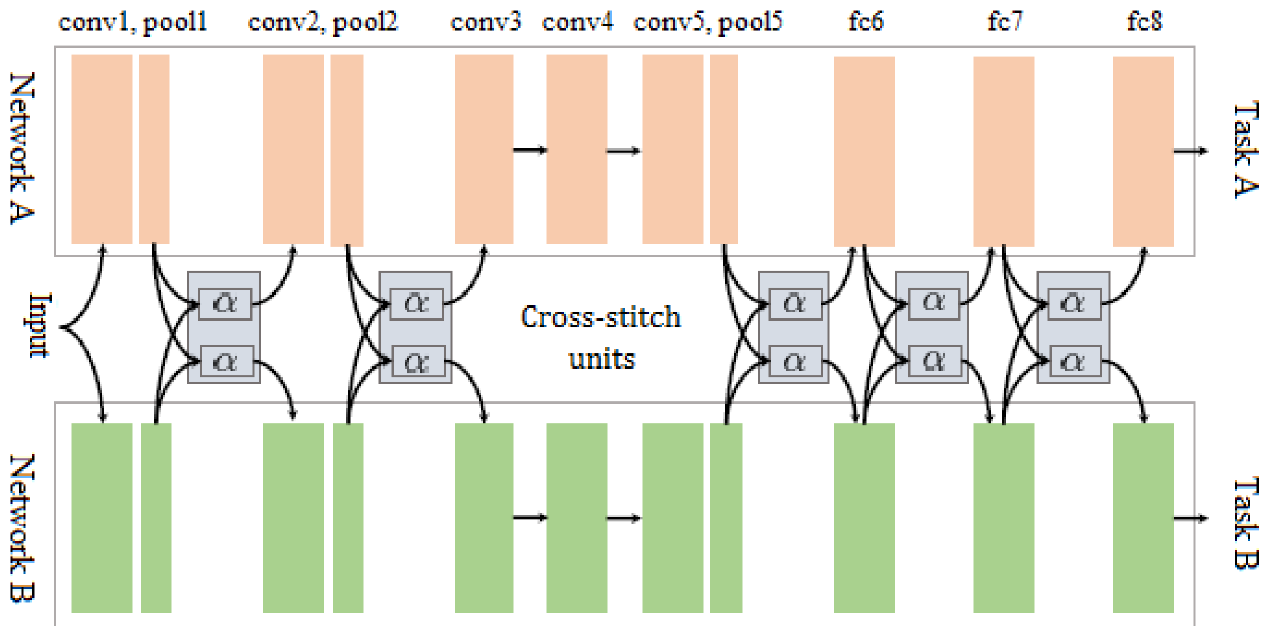

Figure 2.13: adopted from [75], using AlexNet [9] as baseline. Cross-stitch units linearly combine the outputs of corresponding layers from two networks using adaptive α parameters.

in figure (2.13). If the outputs of layer l for tasks A and B are X_A and X_B then the cross stitched output of task A is $\alpha_{AA}\ X_A + \alpha_{AB}\ X_B$ and the cross stitched output of task B is $\alpha_{BB}\ X_B + \alpha_{BA}\ X_A$, where the α parameters are initialized reasonably (α_{AA} and $\alpha_{BB} = 0.9$, while α_{AB} and $\alpha_{BA} = 0.1$) and updated during training. This allows the learning process to adjust the amount of sharing based on the nature of the two tasks. They trained a two-task network on the NYU-v2 dataset for the tasks of semantic segmentation and surface normal prediction, and another two-task network on the PASCAL VOC 2008 dataset for the tasks of object detection and attribute prediction. Therefore, each multitask network is trained on a single dataset, where each input image is annotated with multiple labels, which is similar to Caruana's setup. Their results show the cross-stitch network outperforming the standard implementation of using shared and task-specific layers. The drawback of this method is that it is hard to scale it up to more tasks, as a whole new baseline convolutional network is needed for each new task. The second drawback is that each baseline network needs to be pretrained separately using the corresponding task, which makes the training time increase linearly as the number of tasks increases.

Relationship Networks

Relationship Networks (MRN) [76] try to discover the relationship between the tasks by assuming tensor normal priors over the parameters of task-specific layers in deep convolutional networks. Standard multi-task deep CNNs [30] learn shared representations in the shared feature layers and

multiple independent classifiers in the task-specific layers without inferring the task relationships. This may result in under-transfer in the task-specific layers, and negative-transfer in the shared layers if tasks are not related and modeling the relationship between the tasks can elevate these problems. The Relationship Network [76] used AlexNet [9] as a baseline network where all layers are shared except the last hidden layer and the output layer ($fc7$, $fc8$) were task specific (figure (2.14). For task specific layer l, each task t has a weight matrix $W^{t,l} \in \mathbb{R}^{D_1^l \times D_2^l}$, where D_1^l is the number of input features, and D_2^l is the number of outputs of layer l. The combined weight matrix for all T tasks at layer l is $W^l \in \mathbb{R}^{D_1^l \times D_2^l \times T}$. A multivariate normal distribution for W^l will have a covariance matrix Σ with size $\left[D_1^l \times D_2^l \times T\right]^2$ which is intractable. The authors made the choice of using a tensor normal distribution which corresponds to multivariate normal distribution with Kronecker decomposable covariance structure, where the covaraince matrix is decomposed in terms of input features, outputs, and tasks (Σ_1 with size $\left[D_1^l\right]^2$, Σ_2 with size $\left[D_2^l\right]^2$ and Σ_3 with size $[T]^2$). By learning these covaraince matrices, the network can model a task-relationship, feature-relationship, and in the case of the output layer class-relationship. The Maximum a Posteriori (MAP) estimation $p(W)\ p(Y|X,W)$ of network parameters W is used to update the network weights and the covariance matrices. The maximum likelihood estimation $p(Y|X,W)$ is modeled by a deep CNN, and the prior estimation $p(W)$ is the tensor normal priors imposed on each task-specific layer.

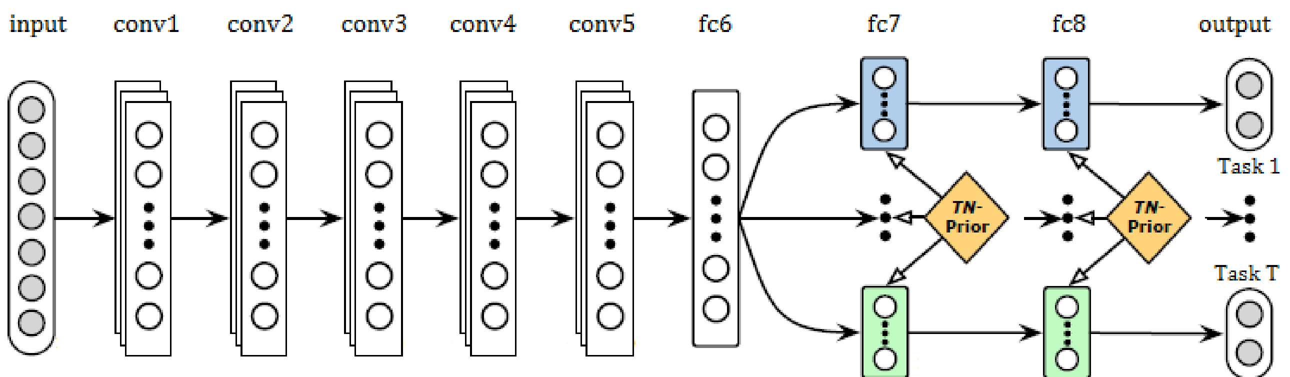

Figure 2.14: adopted from [76] using AlexNet [9] as baseline. The parameters of convolutional layers conv1–conv5 and fully-connected layer fc6 are shared across tasks. The parameters of the last two fully-connected layers fc7–fc8 are modeled by tensor normal priors for learning multilinear relationships of features, classes and tasks.

The experiments were carried out on domain-adaption datasets including Office-Caltech, Office-Home, and ImageCLEF-DA datasets. Each one of these datasets is made of multiple datasets that have the same classes, but the images in each dataset belongs to a different domain (for the Office-Home dataset the images are from 4 different domains: Artistic images (A), Clip Art (C), Product images (P) and Real-World images (R)). Therefore, the setup used here is

similar to our setup where each task has its own inputs and outputs (full dataset), which is different from Caruana's model [30], where each image has multiple labels (one for each task). Because these are small datasets, the CNN were first pretrained on ImageNet. The results show better performance compared to the baseline single task network, and compared to other multitask models.

The tensor normal priors are applied to fully connected layers, and the baseline CNN used in these experiments is AlexNet [9] of 2012 which has 2 hidden fully connected layers. The VGG model [10] also has 2 hidden fully connected layers, and therefore substituting the VGG model for AlexNet is straight forward. However, newer models, such as the Residual models [1, 12, 94], and the Inception models [23, 36, 37] have no hidden fully connected layers, and therefore it is not clear how to apply the priors to convolution layers. Also applying the priors to the output layer assumes that all tasks have the same number of classes, and if tasks are not homogeneous, the priors can not be applied to the output layer.

Multi-Model Multitasking Networks

All multitasking networks presented so far are a single-domain networks (e.g. all tasks are image recognition tasks). Kaiser et al[77] introduced a multitasking model that learns tasks from multiple domains. Their model was trained on ImageNet, multiple translation tasks, image captioning (COCO dataset), a speech recognition corpus, and an English parsing task. The unified structure incorporates building blocks from multiple domains. The model contains convolutional layers which are important for image recognition, and sparsely-gated mixture-of-experts and an attention mechanism which are important for language related tasks. They found that adding a block that is crucial for one task never hurts the performance on other tasks but sometimes it improves them slightly. To train a single model on input data of widely different sizes and dimensions, such as images, sound waves and text, these inputs need to be converted into a unified representation space. A small sub-network called the modality net is used for each domain to translate the input of that domain to the unified input representation. Most of these modality nets apply convolutions and max-pooling to rapidly transform the domain-specific inputs to the unified representation.

The results show that this model can learn multiple tasks from different domains, but it was not meant to achieve state of the art results. However, when comparing the results achieved by training the model on all tasks with the results achieved by training the model on each task separately, for some tasks the multitasking model achieved better results. This shows signs of transfer learning between tasks that belong to different domains. In one of the experiments, a model trained on both the ImageNet and a parsing task (with limited data), showed a better performance on the parsing task, than a single task model trained only on the parsing task.

CHAPTER 3

Encoding Batch Structure through the normalization statistics of BN

3.1 Introduction

Batch normalization [2] was introduced in 2015 to speed up training of deep convolution networks, and to achieve better accuracy. Batch normalization is implemented in all hidden layers of a deep convolutional network. In FC layers BN normalizes each neuron pre-activation output across the current batch of images to have zero mean, and unity variance. In convolutional layers BN normalizes all pre-activation outputs of a single output channel across the current batch to have zero mean, and unity variance. The normalization is applied after the linear transformation of the weight layer and before applying the activation function. Batch normalization allows for faster and more stable training by allowing the network to use a much higher learning rate, and by being less sensitive to weight initialization methods.

The experiments and results presented here show that when a batch normalized network is trained using structured non.i.i.d batches, the behaviour and performance of the network is influenced by structure of the training batches. Because activations are normalized using means and variances that are shared across all images in the current batch, the output activations of one image are influenced by all the other images in the batch. This means that the behaviour of the network is not only receptive to the individual images but also to the total makeup of the training batches. The batch structure examined in our experiments is balanced batches. A balanced batch contains a single instance per class and its size is equal to the number of classes. The results presented in this chapter show that if the network is trained only on balanced batches, and inference is also carried out on balanced test batches, the conditional results improve considerably. Further experiments and analysis show that these big conditional improvements were achieved because the network used the shared statistics of BN to learn an extra logic based on the structure of balanced batches.

The main section of this chapter includes two experiments where the first experiment is carried out using the CIFAR10 [3] dataset, and the second experiment is carried out using three datasets with different number of classes that were randomly and uniformly sampled from ImageNet [4]. The performance of the network trained using balanced batches is compared to the performance of the same network trained using randomly constructed batches. Standard inference is carried out on individual images where fixed batch normalization statistics are used to normalize each individual image. For the network trained on balanced batches, the performance is measured twice, first using standard inference, and then repeated using balanced test batches. The results show that when the network trained using balanced batches is tested using standard inference, its performance doesn't change (similar to that achieved using random batches), while if tested using balanced test batches its performance improves significantly. This suggests that the network is incorporating the structure of balanced batches in the decision-making process. Balancing the test batches requires the labels of the test images, which are not available in

practice, however further investigation can be done using batch structures that are less strict and might not require the test image labels. The conditional results show the error rate almost reduced to zero for nontrivial datasets with small number of classes such as the CIFAR10.

To further investigate this behaviour, a visualization experiment was carried out, where the outputs produced using fixed normalization statistics (standard inference) were compared to the outputs that were produced using the balanced test batch own statistics at different layers in the network. This t-SNE visualization shows a gradual departure between the two outputs that accelerates in the latest stages, where the outputs produced using the balanced batch own statistics lead to correct classification, while the standard outputs fail. This gradual departure between the two outputs indicates that the network is using the structure of balanced batches through the shared normalization statistics. A side result of our investigation of the normalization statistics of BN led to an interesting finding which shows a correlation between the position of the residual connections and the magnitude of these statistics, and these are two different independent components. These results are presented at the end of Appendix A, as we were unable to use them to differentiate between balanced batches and random batches. Finally, an attempt to overcome the condition of balancing the test batches is presented, where the labels generated by the network using standard inference are used to balance the test batches.

3.2 Related Work

Batch normalization [2] was introduced to speed up training of deep neural networks by reducing the effect of covariance shifts; the change or shift in the distribution of a hidden layer input caused by the continuous change of the parameters of the previous layers. BN normalizes each hidden-layer input component independently across the current batch to have a zero mean and unity variance. However, because the normalization is carried out across the current batch, it makes the normalization statistics (μ, σ) depend on the makeup and size of the batch. Such dependency is considered one of the shortcomings of batch normalization when training RNNs, or when using very small batch sizes (or online training with a batch size of one). Most of the normalization methods introduced after BN where designed so that the normalization statistics do not depend on the structure or size of the batch. Weight normalization [5] uses the L2 norm of the incoming weights to normalize the output of that neuron, and in layer normalization [6] each training example in the current batch is normalized independently across all hidden units. Both methods work well with RNNs, and with small batch sizes in feed-forward neural networks. While these alternative methods to BN were designed to eliminate the dependency of the normalization statistics on the structure of the batch, the results and findings presented in this chapter are based on how such dependency can influence the behavior of the network. Therefore, the conclusions presented in this chapter are different from those related to other

normalization methods such as weight normalization [5] and layer normalization [6].

The experiments presented in this chapter show that if the training batches are structured as balanced batches and inference is also carried out using balanced test batches, the conditional results (conditioned on balancing the test batches) improve significantly. The network uses the shared statistics (μ, σ) between the images in the same batch to help classify hard images in the batch. If such a network is tested on individual images (standard inference) rather than on balanced test batches, then its performance will not improve and will be similar to that achieved by training the network on random batches (standard implementation). These conditional gains are hard to obtain as they require structuring the test batches, which in the case of balanced batches requires the labels of the test images. However, these gains are very substantial especially for problems with small and medium number of classes. Therefore, we think it is worth reporting and further investigating these results, and it may be possible to find a special use for them when batches are inherently balanced.

The authors of BN [2] have made an update to the original implementation of BN, and called the new method Batch Renormalization [7], where the new implementation tries to reduce the dependency of the normalization statistics on the size and structure of the batch. In the new implementation, the normalization is done using the running averages of the means and variances and not the means and variances of the batch itself, but when propagating back the error signal, only the contribution that corresponds to the means and variances of the batch itself are considered. The reason for this update is to broaden the application of BN. First, make it more effective with small batch sizes. The second reason for this update, which is more related to the work presented in this chapter, is to efficiently train networks using non-random (non-i.i.d.) batches.

Non-i.i.d batches are batches that are not structured randomly but rather constructed to have a certain shape (in our case balanced batches). And based on the results of one of their experiments [7] which tested a certain non-i.i.d batch structure, they reached a conclusion that is similar to ours. They concluded that a batch normalized network trained using non.i.i.d batches (structured batches) will encode the structure of the batch and incorporate that structure in the decision making process. In other words, when inference is carried out using the same batch structure that is used in the training phase, the results will be superior to those achived when inference is carried out using individual images or random batches. The similarity between their findings and ours is that the structure of non.i.i.d batches can impact the performance of the network that implements BN. The difference between their findings and ours is that they gave a negative example when the performance of the network suffers if BN is used with non-random batches, while here we give a positive example when the performance of the network improves if BN is used with non-random batches. They refer to the batch structure used in FaceNet [8] for face recognition when batches are structured in multiple triplets, where each triplet is made

of an anchor, positive, and negative examples (each batch should contain 2 images per person, an anchor image and a positive image). They simulated such structure using ImageNet, where 16 out of 1000 classes are randomly chosen, and then two instances per class were randomly sampled to create batches of 32 images. Their results show that the performance of a batch normalized network trained using this batch structure is inferior to the performance of the same network trained using random batches. Their results also show that the performance of the network trained using this batch structure improves significantly when inference is also carried out using the same batch structure (error decreases from 33% to 23.5%) which implies that the network is incorporating the structure of the batch in the decision-making process. However, even when inference is carried out using the same non.i.i.d batch structure, the performance of the network was still inferior to that obtained by training the network using random batches (error rate 21.7%). Based on these results they concluded that using BN with non.i.i.d batches is not optimal, and when their new update (batch renormalization) [7] is used instead the performance of the network was not affected by the structure of the batch, and it matched the performance achieved using random training batches.

The experiments presented in this chapter using a different non-i.i.d batch structure show different results to those in [7]. Our results achieved using balanced batches show that the performance of the network improves significantly if measured using balanced test batches (from 3.89% to only 0.69% for CIFAR10 dataset), and it stays the same if measured using standard inference. Therefore, and to the best of our knowledge, our results are the first to show that the network can use the structure of the training batches through the shared normalization statistics of BN to improve its performance for specific batch structures.

3.2.1 Standard Inference with Batch Normalization

In the training phase, the means and variances used for normalization are calculated at each layer using the activations of all the images in the current batch, and therefore the outputs of one image are influenced by all the other images in the current batch. In the inference stage the standard implementation of BN carries out inference on individual images, which makes inference deterministic, and only depend on the image itself. In order to normalize the activations of those individual images BN uses fixed normalization statistics (fixed means and variances). These fixed statistics are calculated by averaging the statistics of all the batches in the training data. For a convolutional layer with output channels of size $h \times w$, and a batch size of m images, where $\acute{m} = m \times h \times w$, the fixed means and variances are calculated for all batches contained in the training dataset of size N images, as follows: -

$$\mu_{inf} = \frac{1}{\acute{n}} \sum_{i=1}^{\acute{n}} \mu_i \tag{3.1}$$

$$\sigma^2_{inf} = \frac{\acute{m}}{(\acute{m}-1)} \frac{1}{\acute{n}} \sum_{i=1}^{\acute{n}} \sigma^2_i \tag{3.2}$$

where $\acute{n} = \frac{N}{m}$ is the total number of batches in the training dataset.

The results presented in this chapter are about the impact of batch structure on the performance of a batch normalized network. In order to measure such impact, inference has to be carried out using the same batch structure that is used in the training stage (assuming we have a test set that can be used to construct such batches). Therefore, when the network is trained using balanced batches, inference is also carried out using balanced test batches where normalization is carried out using the statistics of the batch itself. In the remainder of this chapter, standard inference means carrying out inference on individual images using the fixed normalization statistics obtained using equations 2.1 and 2.2, while the other type of inference is carried out on batches that have the same batch structure used in the training phase and using the normalization statistics of the batch itself (also called inference using balanced test batches). Also, the phrase ”normalization statistics” means the ”means and variances” used to normalize the activations.

3.2.2 Balanced Batches

A balanced batch contains a single instance from each class, and therefore the size of the batch will always be equal to the number of classes. With balanced batches, the standard batch normalization algorithm will be used with the following considerations at the training and inference stages.

- **Training: -** instead of being constructed randomly, training batches are created to be balanced, and to contain a single instance from each class. If training images are to be shuffled to prevent an image from always appearing in the same batch (to improve performance), then the shuffling subroutine needs to be changed so that all generated batches are balanced batches.

- **Inference: -** inference will be carried out twice using standard inference and also using balanced test batches. When comparing the results measured using standard inference with the results measured using balanced test batches, the discrepancy between the two results shows the impact of batch structure on the performance of the network.

3.3 Model selection

All the experiments carried out in this chapter and all other chapters use a deep forward convolution network for image classification. The structure of many of these networks were introduced in chapter two. The design of a convolutional network in general follows certain principles that are fundamental for the success of these networks. First, the network starts with convolutional layers and the last few layers can be implemented as fully connected feed-forward layers. The reason for starting with convolutional layers is because these layers develop filters that act as feature detectors which are transferable across many image recognition tasks. Second, the resolution of the output channels of these convolutional layers should be reduced gradually throughout the network using pooling or other methods. The size of the receptive field of a single convolutional layer is equivalent to the square size of its filter, and early convolutional layers have very small receptive fields in the input image. Therefore, reducing channel resolutions through pooling increases the receptive fields of subsequent convolutional layers, which gradually transfers the features extracted by these layers from local (covers a small area in the image) to global (covers most of the image), and therefore making them semantically more meaningful in solving the recognition task. Third, using the rectified linear function $max(x, 0)$ as an activation function instead of saturating nonlinearities significantly improves the performance of the network. Forth, using a normalization method such as batch normalization makes the learning process more stable and less sensitive to the choice of parameter initialization, which results in a faster training and better final results. Fifth, when using many layers (up to hundreds), there should be shortcut paths that allow for easier backpropagation to the error signal. For this purpose, residual networks use residual connections (identity bypass connections), and inception models use inception layers that have very short paths in them. These are the ingredients required to implement a successful model, and following these principles one can design a convolutional network from scratch. However, it is better to choose one of the latest and most successful models, as they have already found a successful way of implementing these principles.

Convolution networks have evolved significantly throughout the period of this research, and earlier trails and experiments which were carried out in the first half of 2015 used earlier model like AlexNet [9] and the VGG model [10]. The structure of these models was discussed in more detail in chapter two. At the end of 2015 a much deeper and more powerful model was introduced which is the residual network model [1]. This model has incorporated all the required ingredients we have mentioned in this section, including the identity pypass connections called residual connections and the normalization method called Batch Normalization [2]. Although, a number of additions and improvements have been proposed [11, 12], it is still widely used in various image recognition tasks [13, 14, 15]. Therefore, the deep residual network model [1] was a reasonable choice to carry out the experiments in this research.

The structure of the residual model is outlined in chapter two. Here we expand on that discussion in terms of the fine details of the model needed to carry out the experiments in this chapter as follows: -

- For experiments carried out using ImageNet, He et al. [1] used different residual model sizes with different number of layers. These models had 18, 34, 50, 101, and 152 layers. We only experimented with the 18 and 34-layer models, and at the end all the experiments carried out using datasets sampled from ImageNet used the 34-layer model(figure(3.1, right)). The 50-layer model improves the ImageNet top-5 error only by about 0.7% compared to the 34-layer model, while it increases the required GPU memory and computation time significantly. Therefore, using the 34-layer model allows for doing more experiments, and for using bigger batch sizes while achieving comparable results to bigger models. For the experiments carried out using the CIFAR10 dataset, He et al. [1] used residual models with 20, 32, 44, 56, 110, and 1202 layers. Our results indicate that these models were very thin, and by increasing the number of channels in all convolutional layers, the performance of these models improved significantly (similar results were reported in [16]). A wider version of the 20-layer and 44-layer models in [1] were used in this chapter with 4 times the number of channels in each convolutional layer ((figure(3.1, left)).

- The networks only use small convolution filters of size 1×1 and 3×3 which allows the network to use many convolutional layers without increasing the number of parameters and computations significantly. The network used the residual block in figure (2.5) as a building block, where all convolutional layers used 3×3 filters. The 1×1 convolution filters are used only in the residual connections.

- The number of output channels is doubled after each resolution reduction to count for the loss of information. The residual model in [1] uses a convolution with a stride of 2 to reduce channel resolution. For smaller datasets (CIFAR10, and datasets sampled from ImageNet) we found that using max-pooling with a pooling size of 3×3 and stride of 2×2 improves the results slightly. For the 34-layer model (used for ImageNet related datasets) the input resolution is 224×224 pixels which is reduced 4 times throughout the network to 56×56, 28×28, 14×14, and 7×7. The number of output channels at each of these resolutions is 64, 64, 128, 256, 512 channels. The number of channels was not doubled after the first resolution reduction in order to reduce computations. For the 20-layer and 44-layer models used for the CIFAR10 dataset the input resolution is 36×36 which is reduced twice to 18×18 and then to 9×9. The number of channels at each of these resolutions is 64, 128, and 256 channels (4 times the width used in [1]). Channel resolution was not reduced after the first convolutional layer because the input resolution is small (36×36 pixels). The 36×36 input resolution used in our CIFAR10 experiments is

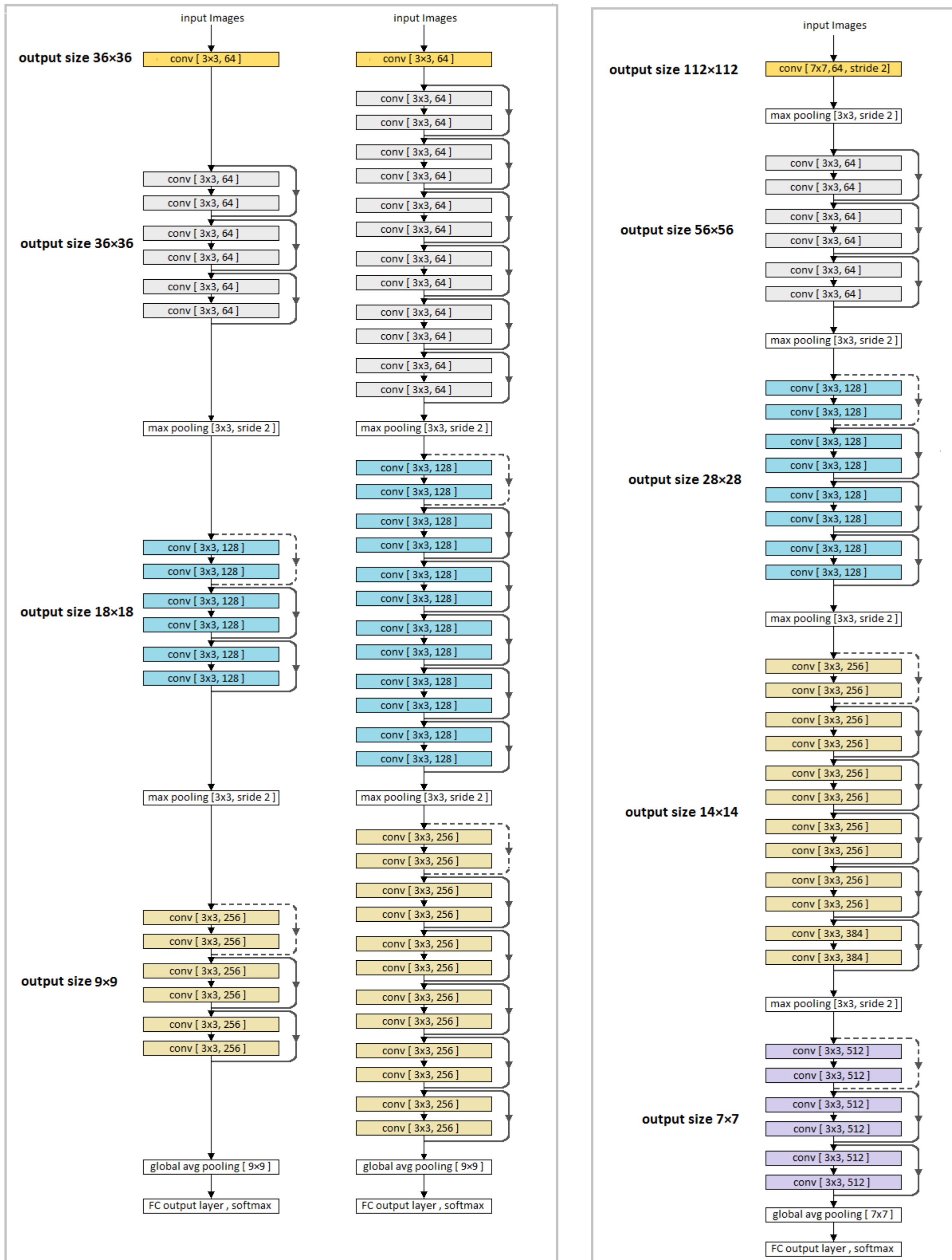

Figure 3.1: **Left:** 20 and 44 layer residual convolution network used with the CIFAR10 dataset. **Right:** 34 layer residual convolution network used with datasets sampled from ImageNet.

bigger than the 32×32 used by He et al. [1]. We found that using bigger input resolution allows for better scale augmentation which leads to better results.

- The positions of reducing channel resolutions for both sets of models follow the designs in [1]. For the 34-layer model max-pooling was inserted after convolutional layers number 1, 7, 15, and 27. For the 44-layer model max pooling was inserted after convolutional layers number 15 and 29. The number here reflects the layer depth. We found that changing the positions of max pooling doesn't affect the results much as long as there are enough convolutional layers between each two consecutive pooling stages.

These principles indicate that the 34-layer residual network in figure (3.1) is a reasonable choice to carry out experiments on datasets constructed from ImageNet. Later chapters use different versions of this network (with different window sizes, different layer widths etc.), where those variations depend on the application of the network.

3.4 Statistical significance of the results

For all experiments in this thesis a two-sample T test is used to test the hypothesis that the data in two vectors (the results of our model vs standard model) comes from independent random samples and from normal distributions with unequal means and equal but unknown variances. In other words, we will test the hypothesis that the difference between our new results and the standard results (which we are trying to improve) is statistically significant. In all experiments we report the p-value and will consider the improvements of the proposed model to be statistically significant if such values are smaller than 0.05. The sample size is equal to the number of runs, and a single run is a complete cycle of training and inference to produce a single result. To finish the experiments in a reasonable time frame, the number of runs was set to 5 for small and medium datasets, and 3 for large datasets, throughout the thesis. Small and medium datasets include the CIFAR10 dataset, and all datasets sampled from ImageNet when the number of sampled classes is smaller than or equal to 100 classes. Large datasets include the ImageNet dataset, and all datasets sampled from ImageNet when the number of sampled classes is greater than or equal to 500 classes. In multi-task learning, the term number of classes means the total number of classes in all tasks.

For the experiments in chapters five and six which compares the performance of a single network trained on multiple tasks or multiple coarse categories, and the overall performance of multiple networks each trained on a single task or single coarse category, a two-tailed paired-t-test is used instead. Each task or category has a pair of measurements one for the single task (category) network and one for the multi task(category) network.

3.5 CIFAR10 experiments and Results

CIFAR10 is a 10-class dataset with 50000 images for training and validation, and 10000 test images. The images are distributed evenly between the 10 classes (airplanes, birds, cars, cats, deer, dogs, frogs, horses, ships, and trucks) with 5000 training images and 1000 test images per class. This dataset is made up of small (32×32 pixels) images, and it is still being used because it is a well-known and widely reported benchmark, and it also has a nontrivial error rate.

The structure of the residual network used in this experiment was outlined in the "model selection" section. Two models with 20-layers and 44-layers are used with 4 times the width (number of channels) used in [1]. Here we look at the setup of the experiment such as choosing the hyper-parameters of the network, and the data augmentation method applied on the input images.

3.5.1 Hyper-parameter tuning

The CIFAR10 training dataset has 50000 images, and the choice of the hyper-parameters was validated by setting aside 10000 images as a validation set. Once the hyper-parameters are chosen, the entire 50000 images of the training set are used to train the network. For the CIFAR10 dataset we found that using simple stochastic gradient decent with weight decay produces comparable results to using gradient decent with momentum or using RMSprop. Weight decay improves the results by about 0.8% for networks trained on random batches and by about 1% for networks trained using balanced batches. However, we found that the performance of the network is not sensitive to the value of the weight decay hyper-parameter. Both models (the 20-layer and 44-layer) for both tasks (random vs balanced batches) perform similarly for values between 0.005 and 0.00005, and the weight decay hyper-parameters were set to 0.0005.

The residual networks used in this experiment implement BN, which allows for higher learning rates. In all our experiments in this chapter and other chapters, we found that when the batch size is increased the learning rate should also be increased by the same factor. This is not widely reported in the literature as experiments tend to use similar batch sizes for each well known dataset (e.g. CIFAR10, ImageNet). Looking at the weight update equation (3.3), weight k is updated using the average contribution of all the training samples in the current batch. Therefore, the learning rate α will effectively be divided by the size of the batch n, and the effective learning rate becomes $\frac{\alpha}{n}$. Thus, when increaseing the batch size (n), the learning rate (α) should be increased by the same factor to maintain the same effective learning rate. In 2014 Alex Krizhevsky [17] experimented using large batches (1024 images), and as the batch size is increased by a factor of n he increased the learning rate by a factor of $\sqrt(n)$. In 2018, Goyal et al [18] also found out that increasing the learning rate by the same factor used to increase

the batch size allows the network to be trained on batches as large as 8000 images without degrading the performance of the network.

$$\Delta w_k = -\alpha \, \frac{1}{n} \, \sum_{i}^{n} g_{ik} - \alpha \; \lambda \; w_k \tag{3.3}$$

In our experiments on the CIFAR10 dataset we found that the performance of the network trained using random batches is more sensitive to the choice of batch size than it is to the choice of the learning rate. We found that using a small batch size of 10 images (equal to the number of classes) produced better results than using larger batch sizes (128 is usually used). For the experiments carried out using balanced batches, the batch size is 10 images by default because it has to be equal to the number of classes. For the batch size of 10 images, a learning rate between 0.01 and 0.05 produced similar results (0.03 was used in this experiment for both models and both tasks). Training is continued for 160 epochs in which the learning rate was decayed 6 times by multiplying it by 0.5. The initialization method by He et al [19] is used to initialize the network weights, however we found that with BN the network produces similar results even when older initialization methods [20, 21] were used. Table(3.1) summarizes the main hyper-parameters. The **Tolerance Range** column in table(3.1) shows the range of the hyper-parameter that is tolerated by the network, where the performance does not drop significantly by choosing a value in that range. For categorical parameters such as the "optimizer", the Tolerance Range field shows a list of tried methods, it is by no means an exhaustive list.

Parameter	Tolerance Range	Used Value	Impact
Learning Rate	(0.01 - 0.05)	0.03	significant
L2 reg parameter	(0.005 - 0.00005)	0.0005	mild
Batch Size	(10 - 20)[1]	10	significant
Weight Initialization	lecun [20], glorot [21], kaiming [19]	kaiming [19]	mild
Optimizer	(SGD, RMSprop, Adam) + L2 reg	SGD + L2 reg	mild
Learning Rate Decay	(Step, Exponential, Schedule) Decay	Step Decay by 0.5 6 times	mild
Epochs	(100 - 160) epochs	160 epochs	mild

Table 3.1: the tolerance range for **weight initialization**, **optimization**, and **weight decay** methods, are based on tried methods in our experiments, they are not exhaustive lists. The **Impact** column shows how important the choice and tuning of a certain parameter on the performance of the network.

[1] for balanced batches the batch size is 10 which is equal to the number of classes

3.5.2 Data Augmentation

The main stream approach in dealing with the CIFAR10 dataset uses cropping without scaling because the images are very small, 32×32 pixels. However, we found that using scaling has improved the results despite the images being very small. So here, the side of the square image is scaled to be between 40 and 80, and then a random square of size 36×36 pixels (or its horizontal reflection) is cropped and fed to the network. Also, the standard colour augmentation method by Krizhevsky et al [9] is used which adds a small random value to each of the RGB channels of the input images.

3.5.3 Results

Two network models (20-layer and 44-layer) are trained using balanced batches, where inference is carried out using balanced test batches, and then repeated using standard inference. To provide perspective, the experiment is also carried out using a standard implementation where training is carried out using random batches, and standard inference is used. Table (3.2) shows the results which are averaged over 5 runs, with the standard deviation included. The table shows that the network trained using balanced batches produces the same results as the network trained using random batches if inference is carried out on individual images (standard inference). But when the inference for the network trained using balanced batches is also carried out using balanced test batches, the performance of the network improves significantly compared to standard inference (table(3.3)). This shows that when the same batch structure is used in the inference stage, the network uses that structure in the decision-making process to improve the performance of the network. Unfortunately, these significant results cannot be achieved in practice because balancing the test batches requires the labels of the test images. However, for the special case when batches are inherently balanced (if such case exist) the error rate can be reduced significantly especially for problems with small or medium number of classes. As table (3.2) shows the error rate for the nontrivial CIFAR10 dataset was reduced by about 80% for both 20-layer and 44-layer models. The results are reported for two network models with different depths to show that these improvements in the results are orthogonal to the improvements obtained from increasing the network depth.

Training	Inference	20 Layers	44 Layers
standard	standard	4.45% ± 0.249	3.89% ± 0.232
Balanced Batches	standard	4.47% ± 0.207	3.9% ± 0.22
Balanced Batches	Balanced Batches	0.97% ± 1.252	0.69% ± 0.10

Table 3.2: CIFAR10 Results, training using random vs balanced batches, inference on individual images vs balanced batches.

	p-value from t-test, sample-size = 5
20 Layers Network	1.73×10^{-7}
44 Layers Network	3.96×10^{-7}

Table 3.3: This table shows the p-values that compares the results obtained using balanced batches and those obtained using standard implementation (random batches). Both values for the 20-Layer and 44-Layer networks show a p-value that is much smaller than 0.05 and therefore the difference between the results is statistically significant. The p-values were obtained using a 2-tailed t-test with a sample size (number of runs) of 5.

Figure (3.2) shows the error curves measured on the central crop of the test set images as training progresses for the 44-layer network. The blue curve is obtained from the standard network trained on random batches. The red curve is obtained from the network trained on balanced batches but because its performance was measured using standard inference it shows similar speed up to the blue curve obtained using standard network. The black curve is obtained from the network trained on balanced batches and also tested using balanced test batches. The black curve shows a significant speed up in convergence throughout training compared to the standard network. This figure agrees with and further validates the final results shown in table (3.2).

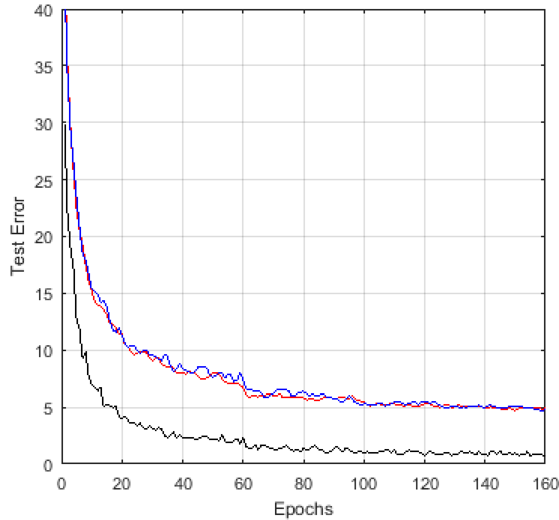

Figure 3.2: validation (test) error curves as training progresses for the CIFAR10 dataset. **blue curve:** standard training and inference, **red curve:** training using balanced batches, **black curve:** both training and inference using balanced batches.

3.6 Experiments and results for the sub-ImageNet datasets

Three datasets with 10, 50, and 100 classes were randomly sampled from ImageNet ILSVRC 2015, and will be used here. For ImageNet, the training images of each class are stored in a separate folder, and the folder names of the classes that make up these three datasets are listed in several tables in Appendix A. Part 1 (first row) of table (B.5) contains the folder names of the classes that make up the 10-class dataset, part 1 (first 5 rows) of table (B.7) contains the names of the classes of the 50-class dataset, and the whole of table (B.5) contains the names of the classes of the 100-class dataset. The make up of these data sets is outlined in Appendix B to give the ability to reproduce the results. The reason for not using the entire ImageNet dataset is because it has 1000 classes, and using balanced batches will require a batch size of 1000 images, which can not be supported by our hardware. The reason for sampling three datasets instead of just one is to measure the impact of batch size on the results obtained using balanced batches. All sampled classes have 1300 training images, and 50 test images. Unfortunately, the ImageNet test set can not be used when only a subset of the whole 1000 classes are used, because labels are not provided for the test images, and are only available through the test server. Therefore, the validation set will be used to sample the three test sets (using the validation set of ImageNet to measure performance is a common practice, and more practical when the experiment is repeated multiple times). Table (3.4) shows the number of training and test images for each of the three sampled datasets.

	10 Class dataset	50 Class dataset	100 Class dataset
Numberof TrainingImages	13000 images	65000 images	130000 images
Numberof TestImages	500 images	2500 images	5000 images

Table 3.4: the number of training images and test images for the 10-class dataset, the 50-class dataset, and the 100-class dataset.

The structure of the residual network used in this experiment is outlined in the "model selection" section. Two models with 18 layers and 34 layers were initially tested and both show significant gains when using balanced batches in both the training and inference stages (although the 34-layer model (figure(3.1, right) outperformed the 18-layer for all three datasets). To focus the analysis on the impact of batch size, only the results of the 34-layer model are reported.

3.6.1 Hyper-parameter tuning

For all 3 datasets, each class has 1300 training images, and 100 of them were set aside as a validation set for hyper-parameter tuning. Once the hyper-parameters are chosen, all 1300 training images are used to train the network. For these datasets which are made of large images, the RMSprop version of SGD significantly outperformed simple gradient decent with weight decay, and therefore it is used to carry out these experiments. RMSprop is covered in appendeix A. At the time of carrying out the experiments, there was not a published work about the implementation of RMSprop that details the proper initialization of the its parameters. We found that using the initialization method used with the AdaM [22] algorithm allows for choosing a better optimal value for the decay parameter used to calculate the running average of the square of the derivative per parameter. This allowed the RMSprop to produce similar results to AdaM while using a single running average per parameter instead of 2. The optimal value of the decay parameter depends slightly on the batch size, and a value of 0.999 works well with batch sizes between 10 and 100 images.

For all three datasets a weight decay value between 0.0001 and 0.001 performs well and therefore it was set to 0.0005. For the network trained using balanced batches the batch size was set by default to 10, 50 and 100 images for the datasets with 10, 50 and 100 classes respectively. For the network trained in the standard way using randomly constructed batches, the impact of the batch size is different between these datasets. For the smallest dataset which has 10 classes, a small batch size of 10 or 20 images produced the best results, while for the bigger datasets with 50 and 100 classes the results were less sensitive to the choice of the batch size. For the middle dataset with 50 classes a batch size between 10 and 50 images produced similar results, while for the biggest dataset with 100 classes a batch size between 25 and 100 images produced similar results. However, we found that using smaller batch sizes makes training less sensitive to the choice of the learning rate. For example, for the dataset with 100 classes, using a batch size of 25 images allows the optimal learning rate to be set to any value between 0.0002 and 0.0016, while using a batch size of 100 images restricts the optimal learning rate between 0.001 and 0.0016. As it has been mentioned in the "CIFAR10 experiments" section, when increasing the batch size, the learning is increased with the same factor, therefore these learning rate values mentioned here are the normalized values (divided by batch size). A (normalized) learning rate of 0.001 is used for all three datasets as it works well with all used batch sizes. Training is continued for 100 epochs in which the learning rate was decayed 5 times by multiplying it by 0.5. This number of epochs is not the optimal choice as adding more epochs improves the results, however it is a reasonable compromise to keep a reasonable training time while achieving good results. Also, the purpose of the experiment is to compare the performance of the network trained using balanced batches with that achieved using random batches, and is

not to achieve the best possible results. The initialization method by He et al [19] is used to initialize the network weights, however we found that with BN the network produces similar results even when older initialization methods [20, 21] were used. Table(3.5) summarizes the main hyper-parameters.

Parameter	Tolerance Range	Used Value	Impact
Learning Rate	(0.001 - 0.0016)	0.001	significant
L2 reg parameter	(0.001 - 0.0001)	0.0005	mild
Batch Size (10 classes dataset	(10 - 20)[2]	10	significant
Batch Size (50 classes dataset	(10 - 50)	50	significant
Batch Size (100 classes dataset	(25 - 100)	100	significant
Weight Initialization	lecun [20], glorot [21], kaiming [19]	kaiming [19]	mild
Optimizer	(SGD, RMSprop, Adam) + L2 reg	RMSprop + L2 reg	significant
RMSprop Decay Parameter	(0.99-0.9995)	0.999	mild
Learning Rate Decay	(Step, Exponential, Schedule) Decay	Step Decay by 0.5 5 times	mild
Epochs	(60 - 100) epochs	100 epochs	mild

Table 3.5: All three datasets use similar hyper-parameters except the batch size. The RMSprop produced significantly better results than simple SGD, and its additional decay parameter requires additional tuning.

3.6.2 Data Augmentation

Instead of using the standard data augmentation technique [10] that is usually used with deep residual networks, the more aggressive augmentation method (usually used with the inception model [23]) is used here. The standard method scales the shorter side of the image to a random value between 256 and 512, and then a square of size 224×224 (or its horizontal reflection) is randomly cropped from the scaled image and fed to the network. In the more aggressive augmentation method used here, the size of the cropped square is chosen randomly to be between 8% and 100% of the size of the maximum square in the image, and the aspect ratio of the cropped square (or its horizontal reflection) is changed randomly to be between 3/4 and 4/3. This aggressive method uses bigger scale augmentation and adds aspect-ratio augmentation which make it more effective in compacting overfitting in smaller datasets (our results showed slight advantage for the aggressive method). The standard colour augmentation method by Krizhevsky et al [9] is used to simulate variance in illumination and intensity that exists in natural images.

[2]for balanced batches the batch size by default is equal to 10, 50, 100 for the datasets with 10, 50, and 100 classes respectively.

3.6.3 Results

The 34-layer model is trained (on all 3 datasets with 10, 50 and 100 classes) using balanced batches, where inference is carried out using balanced test batches, and then repeated using standard inference. To provide perspective, the experiment is also carried out using standard implementation where training is carried out using random batches, and standard inference is used. Table (3.6) shows the results for the three datasets (which are averaged over 5 runs) . As with the CIFAR10, training the network using balanced batches doesn't change the results if inference is carried out on individual images (standard inference). However, if networks trained on balanced batches, are also tested using balanced test batches, the error rate is reduced significantly for all three datasets (table(3.7)).

When using balanced batches, the batch size is equal to the number of classes, and therefore, it is fixed (it is not a hyper-parameter) and its impact on the performance of the network cannot be investigated when using a single dataset. This is the reason for repeating the experiment in this section with three datasets that have different number of classes. The batch sizes used here are 10, 50, and 100 images, for the datasets with 10, 50, and 100 classes respectively. From table (3.6) The relative reduction in the error (when using balanced batches in both training and inference) is very dependent on the batch size. For the 10-class dataset the reduction was 81%, for the 50-class dataset it was 54%, and for the 100-class dataset it was 33%. This is not surprising because balancing the test batches means that the real task of the network is not really classifying the image but rather finding the correct identity of each image in the balanced batch. It makes sense that this task gets harder as the size of the batch gets bigger. As the batch size (thus the number of classes) goes to infinity (large number) the performance measured using balanced batches converges to the performance measured on individual images. It is interesting to notice that the 81% conditional gains for the 10-class dataset is very close to the 80% conditional gains obtained with the CIFAR10 data sets, which also has 10 classes.

Training	Inference	10−Class dataset	50−Class dataset	100−Class dataset
standard	standard	5.97% ± 0.21	7.3% ± 0.13	10.1% ± 0.11
Balanced Batches	standard	5.9% ± 0.19	7.34% ± 0.144	10.14% ± 0.08
Balanced Batches	Balanced Batches	1.1% ± 0.3	3.32% ± 0.18	6.74% ± 0.14

Table 3.6: results for 3 datasets, when training is done using random vs balanced batches, and inference is done on individual images vs balanced batches.

	p-value from t-test, sample-size = 5
10-classes dataset	6.02×10^{-9}
50-classes dataset	6.78×10^{-10}
100-classes dataset	3.03×10^{-10}

Table 3.7: This table shows the p-values that compares the results obtained using balanced batches and those obtained using standard implementation (random batches). For all 3 datasets the p-value is much smaller than 0.05 and therefore the difference between the results is statistically significant. The p-values were obtained using a 2-tailed t-test with a sample size (number of runs) of 5.

Figure (3.3) shows the error curves measured on the central crop for all three datasets as training progresses. Again, the black curves which are obtained when balanced batches are used both in training and inference, show significant speed up throughout training compared to standard implementation (blue curves). The red curves show that training the network using balanced batches does not improve the results if the performance was measured on individual images (standard inference). These learning curves agree with the final results in table (3.6), and show that those final results reflect a behavior that can be measured throughout training.

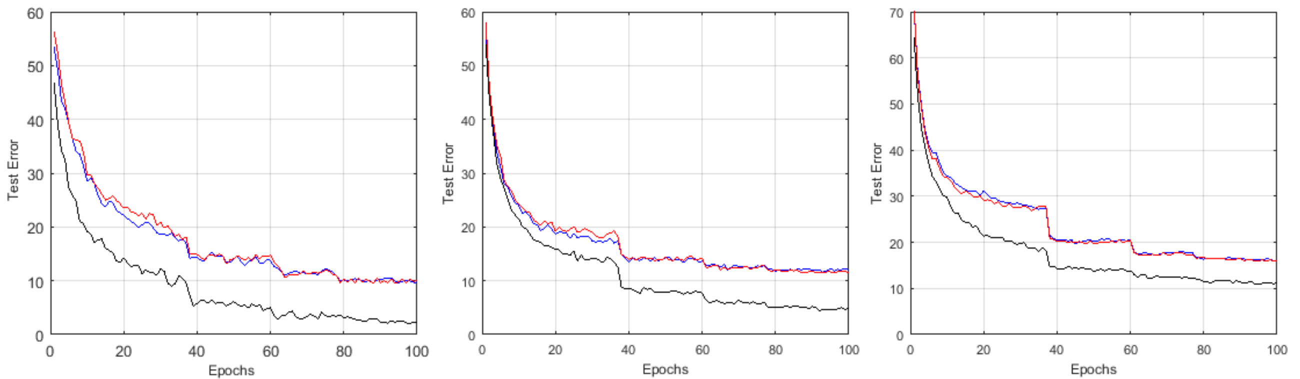

Figure 3.3: the validation (test) error curves as training progresses, blue curve using standard training and inference, red curve training using balanced batches, black curve both training and inference using balanced batches. **left:** for the 10-class dataset, **middle:** for the 50-class dataset, **right:** for the 100-class dataset.

3.6.4 Shuffling the balanced test batches

In the training phase shuffling the training images improves the results if batch normalization is implemented in the network. This experiment shows that if inference is carried out using balanced test batches, and these balanced test batches are also shuffled, the results improve even further (assuming the test image labels are available). First, let's explain what shuffling the test batches mean. When inference is carried out, the prediction of a test image is obtained by averaging the scores of multiple crops from that image. Shuffling the test batches means that each crop of a test image is evaluated with a different batch rather than evaluating all the crops of that image with the same batch. This shuffling matters when inference is carried out using balanced test batches, because the shared normalization statistics depend on the makeup of the batch (all images (crops)).

Table (3.8) shows the results for the three datasets sampled from ImageNet, and it shows that shuffling the balanced test batches will further improve the conditional results, where the relative reduction in the error rate is 96.6%, 72%, and 45.5% for the 10, 50, and 100 class datasets respectively, in comparison to the results achieved by standard inference. For the 10 classes dataset that uses a batch size of 10, the error rate was almost eliminated. Out of the 500 test images, there was only one misclassified image on average. It seems that there is an advantage in presenting the different crops of a test image with different batches rather than presenting all of them with the same batch. One possible explanation is that if images are not shuffled, the current balanced batch may contain multiple similar images that belong to different classes, and that may confuse the network. By shuffling the images at inference time, the chances of presenting all the crops of an image with another similar image from a different class will be reduced. However, to produce shuffled balanced batches, the labels of test images must be provided, and even having a special application when batches are inherently balanced is not enough.

Training & Inference	10−Class dataset	50−Class dataset	100−Class dataset
Balanced Batches	0.2% ±0.178	2.04% ±0.114	5.5% ±0.093

Table 3.8: Results obtained by shuffling balanced test batches

3.7 t-SNE Visualization

The results presented in this chapter so far show that the performance of a network trained using balanced batches depends on the kind of normalization statistics used in the inference stage. If fixed statistics are used (standard inference) then the performance of the network does not improve, while if the balanced test batch's own statistics are used the performance of the network improves significantly. This suggests that the network has learned to incorporate the structure of the test batch in the decision-making process through the batch's shared statistics. This visualization experiment tries to look inside the network and see how it behaves differently based on the kind of normalization statistics used in the inference stage. To do that the outputs of certain test images are visualized at different stages of the network. The test images that are selected for visualization are the ones that were misclassified when using fixed statistics (standard inference) but were correctly classified when using the balanced test batch's own statistics. For these images inference is carried out twice, and in both cases the outputs are generated using the same network weights (trained using balanced batches), and the only difference is in the normalization statistics. A t-SNE visualization is used to project the output of the convolutional layers at different depths in the network. By comparing the two outputs for each image, this visualization should show what happens throughout the network that makes the same image produce two different outputs when using different normalization statistics.

The 10-class dataset sampled from ImageNet and used in the previous section is used here but with different settings. The test set for this dataset has only 500 images, and therefore there are not enough test images to project (test images that are misclassified when using fixed statistics and correctly classified when using the batch's own statistics). To solve this problem the training dataset is divided into two equal parts with 6500 images used as a training set, and 6500 images used as a test set. Using these settings, the network was trained using balanced batches with the same hyper-parameter settings used in section (2.5.1). When the performance of this network was measured using standard inference (fixed statistics) the error rate was 10.8%, while when it was measured using the balanced test batch's own statistics the error rate was 2.9%. Looking at these outputs there were 514 test images that fit the projection criteria. Figure (3.4) shows the t-SNE projection of the outputs of these 514 test images at different layers for both inference methods. For each image the projection is for a single central crop, and therefore the error rates in this experiment are measured using a single crop. Here, the entire output of the convolutional layer is projected using PCA into a lower demission (50 to 100) to eliminate noise, and then t-SNE is used to project this PCA output into a 2D ouput. The green dots are generated using fixed normalization statistics (standard inference), and the red dots are generated using the balanced test batch's own statistics, where a single dot represents a single test image.

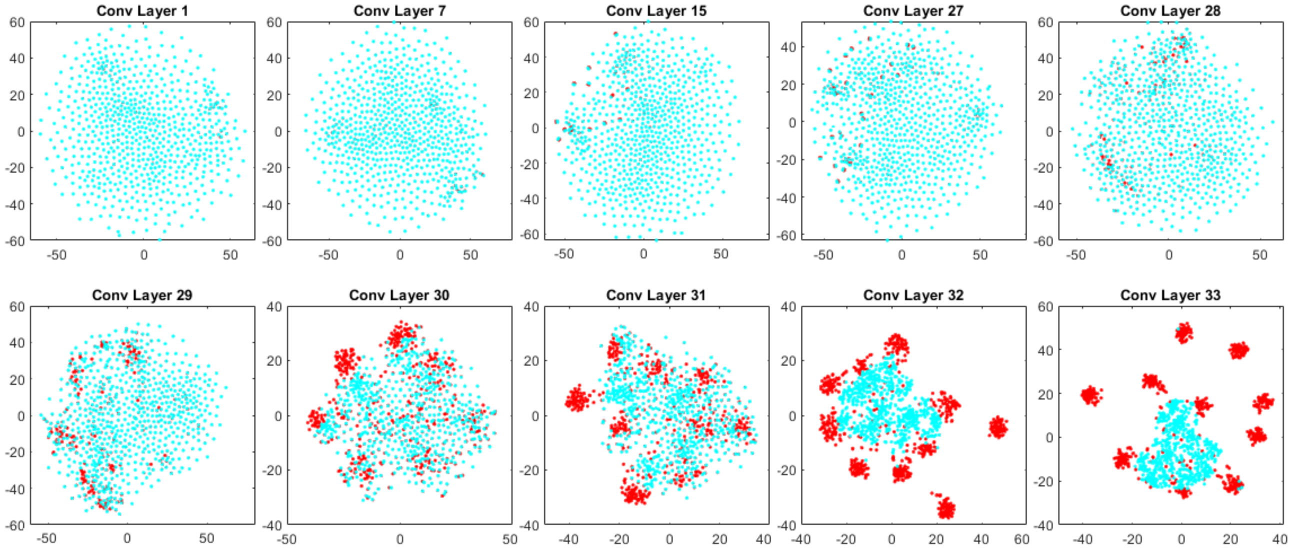

Figure 3.4: a t-SNE 2D visualization of the outputs of 10 out of the 33 convolution layers. These are the outputs of the images that were misclassified by standard inference, and were correctly classified using balanced test batches. They start with similar outputs in the early stages (green for standard inference, red for balanced batch inference) and then gradually depart were standard inference fails, and inference using balanced batches succeeds.

Figure (3.4) shows that for early convolutional layers the outputs generated by the two inference methods for each image were almost identical and that is why the green dots cover the red dots. This shows that using fixed means and variances for normalization has no noticeable difference from using the balanced batch's own means and variances in the early layers of the network. However, as the figure shows this subtle difference starts to build up gradually and it significantly increases in the later stages starting from convolutional layer 27 onward. At the later convolutional layers the outputs generated using the batch's own statistics start to divide into different clusters, and start to divert from the outputs generated using fixed normalization statistics which fail to divide into different clusters. For the last two convolutional layers, the outputs generated using the batch's own statistics divide into 10 clusters, which is equal to the number of classes in the dataset, while the outputs generated using fixed statistics stay as a single big cluster. This clearly shows that for hard images (those misclassified using standard inference) the network is using the shared normalization statistics to gradually guide the network toward producing the right classification.

To complete the picture, Figure (3.5) shows the projection of another 500 test images that were correctly classified using both inference methods. It is clear that both methods have succeeded in separating the test images into multiple clusters (10 cluster for 10 classes). Also, the separation was achieved at an earlier stage compared to figure(3.4), which is expected because these are easier images that were correctly classified using standard inference.

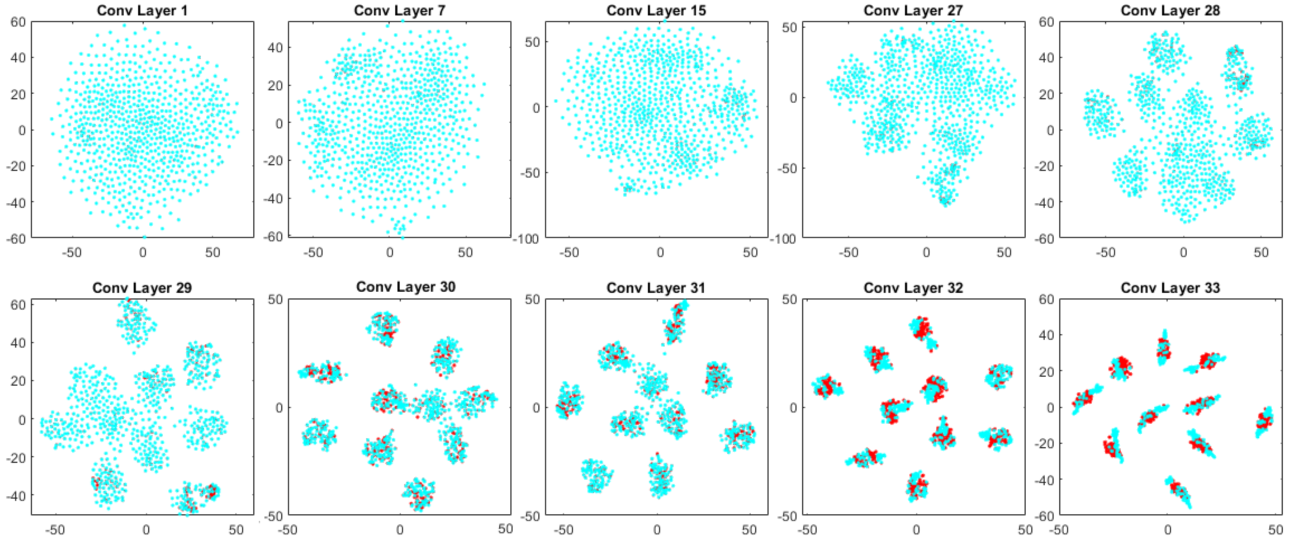

Figure 3.5: a t-SNE 2D visualization of the outputs of 10 out of the 33 convolution layers. Contrary to the projection in figure (3.4), this visualization is done for a subset of the test images that were correctly classified by both inference procedures.

3.8 Inference using the balanced batch's own statistics

The results and visualizations presented in this chapter so far show that the network incorporates the structure of balanced batches in the decision-making process if inference is carried out using balanced test batches. We know that the network is doing that through the shared normalization statistics of the test batch, which are calculated using the output channels of all the images in the current batch. But what kind of logic is the network inferring based on the structure of balanced batches? The experiment presented in this section tries to answer this question.

In this experiment the 20-layer network in section (2.4) which was trained on the CIFAR10 dataset using balanced batches is used. Then an inference cycle that classifies a single batch is repeated many times using this trained network. In each cycle, 10 images from the test set are selected to construct a balanced batch, and another image is randomly chosen to dilute the balanced batch (to randomly replace one of its images). The 10 images used to construct the balanced batches are only chosen from the test images that were correctly classified by standard inference, and therefore the network is more confident about their identity. On the other hand, the images used to dilute the balanced batches are only chosen from the test images that were misclassified by standard inference, and therefore the network is less certain about their identity. Inference is carried out using the test batch's own normalization statistics. This process is repeated many times (in this case 10000 times), and each time using a different balanced batch and a different image to dilute the balanced batch. The idea is to see how the network will use the shared normalization statistics of these batches to classify the image used to dilute the

balanced batch. Because a balanced batch is made of 10 images (one from each class), the chances of keeping a balanced batch when replacing one image randomly is 10%. However, when looking at the results of this experiment, 74.2% of these diluted batches were classified by the network as balanced batches. This means that in 74.2% of the time the diluting image was given the same identity as the replaced image, while on average only 10% of the diluting images should have the same identity as the replaced images in the balanced batches.

To get a more generalized view, the experiment was repeated when the balanced batches were diluted by randomly replacing 2, 3, 4 and 5 images out of the 10 images that make the balanced batch. When randomly replacing 2, 3, 4, and 5 images, the chances of keeping a balanced batch is 2%, 0.6%, 0.24%, and 0.12% respectively. On the other hand, looking at the results produced by the network, the ratio of diluted batches that were classified as balanced batches was 56.6%, 43.8%, 33.7% and 25.3% respectively. So, again the network is interpreting the identity of the confusing images used to dilute the balanced batch to match the identity of the replaced images.

Using a less restrictive measurement than a fully balanced batch, the network produces more consistent ratios across all five cases. In this new measurement, we measure the ratio of diluted batches where the predicted identities of the diluting images are restricted by the network to match one of the replaced images. Such batches are not necessarily fully balanced (e.g. if two replaced images belong to classes 3 and 7, then the two images replacing them can be interpreted as (3,7), (7,3), (3,3), or (7,7)). The results show that when balanced batches were diluted by randomly replacing 1, 2, 3, 4 and 5 images, the ratio of diluted batches that were classified by the network to satisfy this measurement were 74.2%, 73%, 72.1%, 71.4%, and 70.6% respectively. Table (3.9) summarizes all the results, and figure (3.6) draws all the results for easier visualization. These are more consistent ratios than the ratios of predicting fully balanced batches. Based on these consistent ratios the network is using the structure of balanced batches (more than 70% of the time) as explained by equation (3.4).

$$(\mathbf{m} \textit{ out of } \mathbf{n} \textit{ images identified}) \;\; \&\& \;\; (\textit{batch is balanced}) \quad \rightarrow \\ (\textit{remaining images belong to remaining } \textbf{n-m} \textit{ classes}) \qquad (3.4)$$

The logic in equation (3.9) can be interpreted as follows: - When a balanced batch of size n is passed through the network, and the network is confident of the identity of m images, then the identity of the remaining (n-m) images (that the network is not confident about) are

restricted to the (n-m) missing classes. Let's explain this by a simple example. If a dataset has only 4 classes, a balanced batch looks like this 1, 2, 3, 4. If such balanced batch is passed through the network, and the network is only confident about the identity of 2 images 1, - ,3, -, then the identities of the remaining two images are restricted to the missing classes (either class 2 or 4). Such possible predictions are 1,2,3,2, 1,2,3,4, 1,4,3,2 and 1,4,3,4. In our experiment the network is confident of the identity of the images used to construct the balanced batches, and it is less confident about the identity of the images used to dilute the balanced batches.

	Number of misplaced Images				
	1 image	2 images	3 images	4 images	5 images
Actual Ratio of Balanced Batches	10.0%	2.0%	0.6%	0.24%	0.12%
Ratio of Balanced Batches Predicted by the Network	74.2%	56.6%	43.8%	33.7%	25.3%
Ratio of Predicted Batches that Conform to Eq(3.4)	74.2%	73%	72.1%	71.4%	70.6%

Table 3.9: Balanced batches are diluted with misclassified images. The table compares the ratio of balanced batches, and batches that conform with eq(3.4) (produced by the network) with the actual ratio of balanced batches.

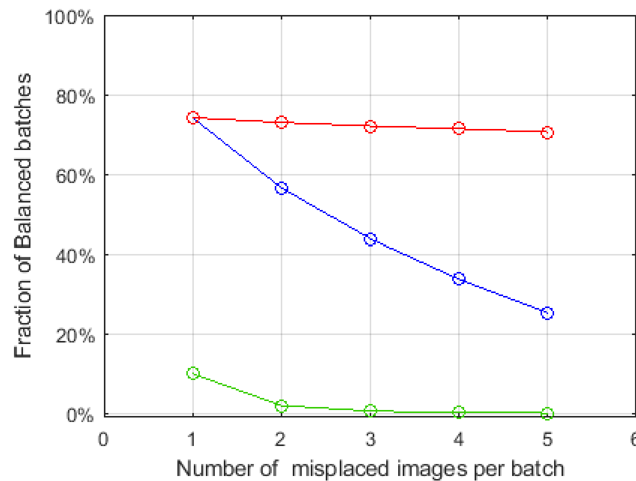

Figure 3.6: This is a visualization of table (3.9). The **blue** curve show the ratio of balanced batches produced by the network. The **green** curve shows the actual ratio of balanced batches, and the **red** curve shows the ratio of batches that conform with equation (3.9).

Lets rephrase this conclusion in more abstract terms. If BN was not implemented in the network then the weight updates are only affected by the contributions of individual images.

Therefore, there is no way for the network to learn how to encode the structure of the training batches. On the other hand, if BN is implemented, then the weight updates are not only affected by the contributions of individual images but are also affected by the normalization statistics of the training batches. These batch statistics reflect the structure of the training batches, and if training is carried out using a single batch structure, then the network will encode that structure, because it was only exposed to the normalization statistics of that structure. In our case the network learns to expect balanced batches. When a batch passes through such network it will deal with it as it will deal with a balanced batch. If the network is confident of the identity of 90% of the images in the batch, then it will interpret the identity of the remaining (hard) 10% in a way that makes the batch as balanced as possible. Therefore, if inference is carried out using the same batch structure used in the training phase, the network learns to rely on that structure to classify confusing images (as for such images the network can not rely on the image itself). In our case, the results show that the network has learned to use the structure of balanced batches as explained in equation (3.9) in more than 70% of the cases to correctly classify confusing images that otherwise will be misclassified.

3.9 Difficulty balancing the test batches

It is tempting to try to translate these big conditional gains into actual gains, however these results are very hard to achieve in practice because structuring test images as balanced batches requires the test image labels. One attempt that is presented in this section is to use the labels generated by standard inference to balance the test batches. In this experiment the 44-layer network used in section (3.4) which was trained on the CIFAR10 dataset is used. The error rate measured using standard inference for this network was 3.89% (table(3.2)). Using the labels generated by this network to balance the test batches will generate batches which are balanced with accuracy of 96.11%. The idea is as follows:- If the performance achieved using random test batches is 96.11%, and the performance achieved using fully balanced test batches is 99.31%, then the hope is that the performance that can be achieved using test batches that are 96.11% balanced will be something in between. Then these more accurate labels achieved using semi-balanced test batches will be used again to rebalance the test batches and generate even better results, where the process of reapplying labels continues until no improvements can be achieved. Figure (3.7) shows this process repeated 20 times for the CIFAR10 dataset, but unfortunately it didn't lead to more accurate results (better labels). The error rate stayed almost constant as a horizontal line.

So why did carrying out inference using test batches that are almost perfectly balanced (with 96.11% accuracy) not improve the results, even though using test batches that are fully balanced improves the results significantly (to 99.31% accuracy)? The reason is because the

hardest 3.69% will be misclassified by standard inference and they will be put in the wrong test batch. Based on the analysis in section (3.7), when such a batch is passed through such network, and the network is confident of the identity of 96.11% of the images in the batch, then the remaining 3.69% of the images will be interpreted by the network to make the batch as balanced as possible. Therefore, the network will most likely confirm the wrong labels given to the 3.69% hardest images by standard inference. Therefore, standard inference is not adequate to solve the problem of balancing the test batches, because it will misclassify the important images, where gains need to be made.

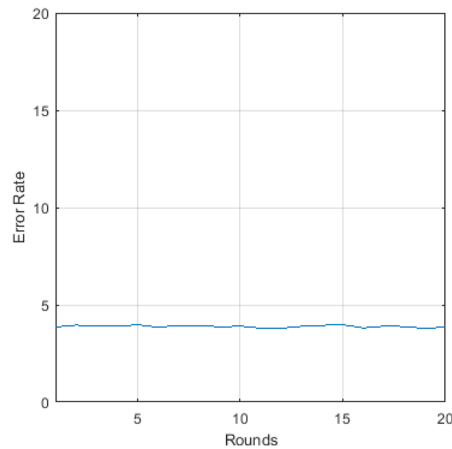

Figure 3.7: the repeated process of balancing test batches starting from labels generated by standard inference fails.

3.10 Summary

- The experiments in this chapter do not claim any improvements to the performance of the standard implementation, but rather point into an interesting behaviour exhibited by the network and caused by batch normalization.
- With batch normalization, the output activations of one image are influenced by all the other images in the current batch through the shared normalization statistics (means and variances). The results (presented in this chapter) show that when a network that implements batch normalization is trained using structured non-random batches, the network will encode the structure of these batches and use it in the decision-making process. If inference is carried out using the same batch structure used in the training phase, and normalization is carried out using the batch's own statistics, then the network will use the batch structure to improve the performance of the network. The batch structure used here is balanced batches, which contain a single instance from each class.
- When a network that implements batch normalization is trained using balanced batches,

and tested using balanced batches, its performance improves significantly. The network achieves these improvements by implementing an additional logic based on the structure of balanced batches as shown in equation (3.4). The network relies on the test image itself when it is confident about the identity of that image, while it relies on the structure of the test batch, when it is not confident about the identity of the test image.

- When test batches are balanced then the main task of the network is to identify the class identity of each image in the balanced batch. The results show that the gains achieved using balanced batches decrease as the size of the batch increases. This makes sense because finding the correct identity of each image becomes harder. Therefore, the performance obtained using balanced test batches will converge to the performance of standard inference as the batch size goes to infinity (or a very large number).

- The t-SNE visualization in section (2.6) shows that the network is using the normalization statistics of the balanced test batch to gradually guide the outputs of confusing images toward producing the correct classification.

- The performance gains shown in this chapter require balancing the test batches, which is not achievable in practice because it requires the labels of test images. Experiments show that it is not possible to use the labels generated through standard inference to balance the test batches, because these labels will misclassify the important (difficult) images where gains need to be made. Nevertheless, further studies can be carried out using less strict batch structures, where the labels of the test images may not be required.

CHAPTER 4

Training deep CNNs using a variable input size

4.1 Introduction

Main stream implementations of convolutional networks [9, 10, 1, 23, 36] are trained using a fixed input window size of $N \times N$ pixels (224×224 for residual networks and 299×299 for inception networks). For such models to be robust against scale variations of the main objects in the input images (become scale invariant), they rely on data augmentation methods. Such data augmentation methods scale the shorter side of the image to a random value between S_{min} and S_{max}, and then crop a random square of size $N \times N$ pixels as input to the network ($N \leq S_{min}$). The amount of scale augmentation that can be introduced by data augmentation is limited, because increasing the scaling range (S_{max}/S_{min}) while keeping the input size ($N \times N$) fixed can cause the cropped square to miss most of the details in the image. The effective amount of scale augmentation can not be increased significantly because the network is trained using a fixed input size. The proposed model presented in this chapter tries to solve this problem by training the convolutional network using a variable input size. Our implementation is similar to the Spatial Pyramid Pooling Network [67], and in the next section will discuss the differences and similarities between the two models.

The proposed model uses the deep residual convolutional network [1] with 34 layers (figure (2.4, right)) as a baseline network. However, instead of training the network using a single fixed input channel size of 224×224, the network is trained using a variable input size. In each new iteration a different size is chosen randomly from a set of predefined sizes, and the network is adjusted and trained using that input size. The predefined sizes start with 160×160 and increment by 32 or 64 to align with the amount of resolution reduction in the network. The number of parameters in the convolutional layers is independent of the size of its input, therefore it is possible to train such layers using a variable input size. In order to train the whole network using a variable input size, the only consideration is to keep the number of inputs (and therefore the number of parameters) to the fully connected layers constant. In our model this is implemented by inserting a variable global average pooling between the convolutional layers and the fully connected layers.

Training the network using a variable input size, allows for a higher effective scaling range without cropping out most of the details in the input images. The experiments on the ImageNet dataset show that this approach produces better results compared to the standard implementation that uses a fixed input size. The experiments on the ImageNet dataset also show that trying to improve the performance of the standard model just by using a more aggressive data scale augmentation does not work. This shows that when using a fixed input size, the effective amount of scale augmentation that can be introduced through data augmentation is limited.

The performance of the network can be measured using a single central crop from the image, or by averaging the prediction of multiple crops. The proposed model shows significantly more improvements to the single-crop results compared to the standard model. The standard index for measuring performance is the multi-crop performance, however the single-crop performance is also important when inference time is critical (when inference needs to performed in real time for computationally extensive tasks such as object detection [93, 13, 97, 96]). Further experiments using both the proposed model and the standard model show that the improvements in the single-crop results are (to some extent) related to the maximum input size used in the training phase. The results for both the standard model (trained using a fixed input size) and the proposed model (trained using a variable input size) show that the multi-crop results improve slowly by increasing the input size, while the single-crop results improve considerably. In conclusion, the results show that training the network using a variable input size that is chosen from a wider range produces the best single-crop performance.

4.2 Related work

In this chapter a convolutional network trained using variable input-channel sizes is presented. The implementation uses the residual model [1] as the baseline network. All weight layers in the residual model are convolutional layers with the exception of the output layer which is a fully connected layer. The baseline network is trained using a fixed input-channel size of 224×224, while our model is trained on 6 different input-channel sizes. In order for this to work the fixed-size average pooling applied after the last convolution layer was changed to have a variable size that is proportional to the input size (the size of the input RGB channels). This will keep the number of inputs to the fully connected layer (output layer) fixed and therefore allow the same network to be trained using multiple scales (multiple input sizes).

This implementation is similar to the SPP network [67], where they used a variable pyramid max-pooling to fix the number of inputs to the fully connected layers, instead of using a variable average pooling. The two models are similar, but there are differences in the implementation and in the final conclusions. First, in their implementation the results of the baseline model can be improved just by substituting the last max pooling layer with (the more complex) pyramid max-pooling, and the improvement can be enhanced further by training the network using multiple input-channel sizes. While in our model a simple global average pooling is used, and therefore the improvement comes solely from training the network using multiple input-channel sizes. However, their implementation of the pyramid pooling, which uses 4 levels, is more complex and can become a bottleneck in certain applications (e.g. object detection) when such pooling is applied thousands of times on the feature maps of the last convolutional layer. Therefore, it is easier to adapt our implementation to more complex tasks such as object-detection. Second,

their model was implemented using the caffe [98] toolbox which restricts the settings of training the network using multiple input sizes. Basically, they used 2 standard networks with two different input sizes (180×180 and 224×224). During training, they switched back and forth between the two network sizes, where switching happens at the end of each epoch. They couldn't switch between the two sizes in each new iteration because that introduces a significant overhead. Our model is implemented directly using CUDA and that allowed the model to be trained using more sizes and larger scales. Also, in our implementation switching between sizes is done with each new batch. Finally, their results only report improvements to the multi-crop performance over the baseline networks, while our results show that the single-crop performance improvements are more significant than the multi-crop performance improvements, while their results does not report such discrepancy. Single-view performance is important for tasks where inference-speed can be an issue such as object-detection.

4.3 Implementation

In this section we look at the implementation details of the proposed model. The model is implemented using the residual network [1] as a baseline network, where training is carried out using a variable input size, instead of using a fixed input size as in the baseline implementation. First, we look at the standard baseline model in relation to the size of the input, and how changing the input size changes the channel sizes at all stages of the network. Then, we see how the baseline model can be changed, so that the convolutional network can be trained using a variable input size.

4.3.1 Baseline network in relation to input size

The 34-layer deep residual network [1] in figure (2.4), shown earlier in chapter two, is used as a baseline network. Using the 34-layer model is a good compromise between size/speed and accuracy. Any standard convolutional network in the literature can be used as a baseline network, and the advantages and reasoning behind choosing the deep residual model [1] are outlined in chapter three. In short improving the performance of a powerful base model like the residual network is more challenging than improving the performance of an older model like AlexNet and VGG models, and it indicates that these findings will probably be applicable to other advanced models that use similar powerful components. The 34-layer baseline network is trained using a fixed input resolution of size $N \times N$ pixels, and this resolution will be reduced after each resolution reduction stage. This input resolution is reduced 4 times after convolutional layers number 1, 7, 15, and 27 by a factor of 4, 2, 2, and 2 respectively. These 4 resolution reduction stages reduce the input resolution by a total factor of 32 resulting in a final resolution

of $\frac{N}{32} \times \frac{N}{32}$ at the last convolution layer. Figure (4.1) shows the resolution reduction process for the 33 convolutional layers. The notation above each resolution shows the number of consecutive convolutional layers with the same resolution. A global average pooling is inserted after the last convolutional layer which reduces its resolution to a single value. The mainstream value for N is 224.

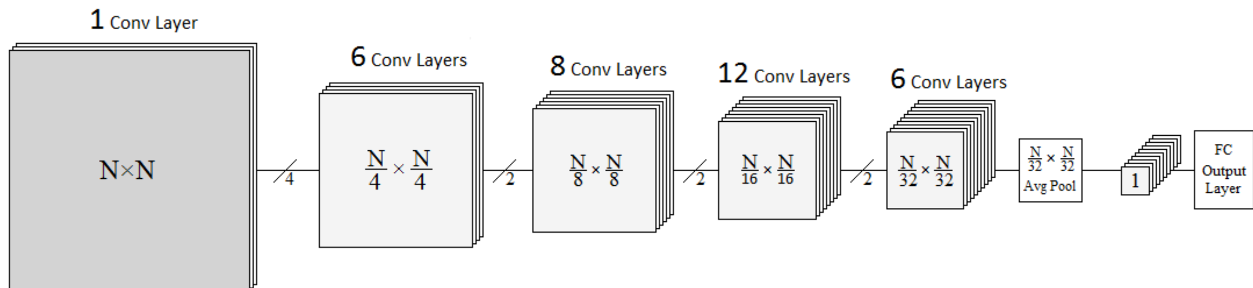

Figure 4.1: Standard network:- shows how the output sizes (resolutions) of all convolutional layers change as the input size $N \times N$ change.

4.3.2 Using a Variable Input Size

Looking to figure (4.1) of the baseline network, changing the size of the input N results in changing the sizes of the inputs and outputs of all the convolutional layers. However, the number of parameters in those convolutional layers does not change by changing the size of the input. Therefore, for the 33 convolutional layers, changing the size of the input N during training does not disturb the learning process. For the output layer which is constructed as a fully connected layer, changing the size of the input will change the number of trainable parameters in that layer. Thus, to train the whole network using a variable input size (using different values for N), the size of the input to the fully connected layers (the output layer) must stay fixed. This can easily be achieved by using a variable average pooling after the last convolutional layer to always produce a fixed size output. Therefore, using an average pooling with a variable size equal to $\frac{N}{32} \times \frac{N}{32}$, will always reduce the channel size of the last convolutional layer to a single value, which in turn fixes the size of the input to the fully connected output layer. Figure (4.2) shows a convolutional network that is trained using a variable input size, where the input size is chosen from a set of predefined sizes. In each training iteration, one of the predefined sizes is chosen randomly, and the network is trained using that input size. The model shown in figure (4.2) is trained using 6 predefined sizes N_1, N_2, N_3, N_4, N_5, and N_6, and for input size $N_i \times N_i$, the average pooling size applied after the last convolutional layer is set to $\frac{N_i}{32} \times \frac{N_i}{32}$. Therefore, each time the input size changes, the size of the average pooling applied after the last convolutional layer also changes to keep the number of parameters in the network fixed.

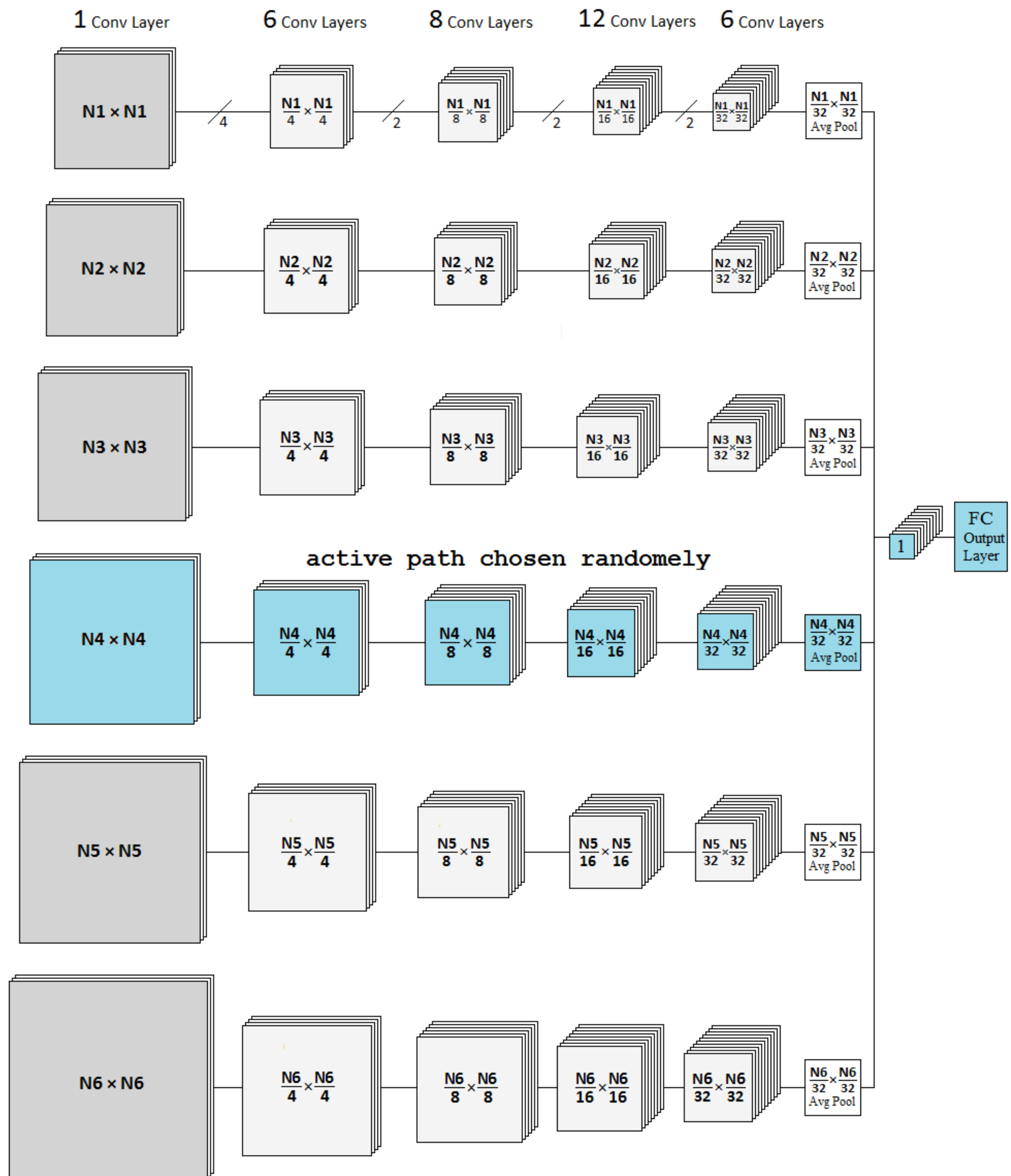

Figure 4.2: variable input window size network with 6 different widow sizes. One size is chosen randomly at each iteration, and the chosen size will decide the resolution sizes at all convolution layers.

Choosing the range of input sizes

There are practical limitations to the range of input sizes that can be used to train the network. The network is trained on real natural images with an average size of 300×400, and therefore the lower limit (smallest size) should not be too small, because that can destroy important details in the input images. In our experiments the smallest input size was set to 160×160 and using smaller sizes did not improve the results further. For choosing the upper limit (largest input size), the constraints are computation time and required GPU memory size. Setting the input size too high will increase the memory footprint of the network significantly, and therefore the network can only be trained using smaller batch sizes. For a large dataset such as ImageNet which is made of 1000 classes, using a small batch size that is much smaller than the number of classes can harm the performance of the network. Using larger input sizes will also increase the training time significantly, when the network is trained using a large dataset such as ImageNet which is made of about 1.28 million (real size) training images. Our hardware resources are 2 PCs, each equipped with an NVIDIA Titan x Pascal GPU that has 3584 CUDA cores and 12GB of memory. These resources allowed us to test input sizes as large as 480×480 which for the 34-layer baseline network shown in figure (2.4) allows for a maximum batch size of 64 images. This small batch size is not optimal for ImageNet, yet the results improved (especially the single-crop results) when using larger upper limits for the input size.

Once the lower and upper limits of the input size are set, the network can be trained by randomly sampling a size in this range, or by choosing from a predefined set of sizes that fall in this range. We have chosen to use a predefined set of sizes, that are multiples of 32 which is equal to the ratio of resolution reduction between the input resolution $N \times N$ and the resolution of the last convolutional layer $\frac{N}{32} \times \frac{N}{32}$. This allows for a cleaner implementation, where increasing the input size one step (by 32) will increase the resolution of the last convolutional value by 1. The first tested implementation uses 6 predefined input sizes equal to 160, 192, 224, 256, 288, and 320, (here 160 means 160×160 pixels). The output size (resolution) of the last convolutional layer for each of these input sizes is 5×5, 6×6, 7×7, 8×8, 9×9, and 10×10 respectively, and therefore the size of the average pooling applied after the last convolutional layer for each of these input sizes is also 5×5, 6×6, 7×7, 8×8, 9×9, and 10×10 respectively. Matching the average pooling size to the output size of the last convolutional layer fixes the number of weights in the network as the input size changes. The second model also used 6 predefined input sizes equal to 160, 224, 288, 352, 416, and 480. For these inputs, the size of the average pooling applied after the last convolutional layer is 5, 7, 9, 11, 13, and 15 respectively. Here increasing the input size one step (by 64) increases the output resolution of the last convolutional layer by 2. We found that this model produces similar results to a finer model that used 11 predefined sizes which starts at 160 and increments by 32.

Increasing the input size will also improve the performance of the baseline network that uses a fixed input size. Therefore, to make fair comparisons, the performance of the proposed model will be compared to the performance of a standard model that requires comparable training times. For this reason, the performance of the first model (160-320), will be compared to the performance of a standard model that is trained using a fixed input size of 224×224, and the performance of the second model (160-480) will be compared to the performance of a standard model that is trained using a fixed input size of 320×320. The proposed models (trained using a variable input size) are named using the upper and lower limits on the input size, therefore the first model is called the (160-320) model, and the second model is called the (160-480) model.

Data Augmentation and Scaling range

The same data augmentation algorithm will be used for both the standard network trained using a fixed input size, and the proposed network trained using a variable input size. The shorter side of the training image is scaled to a random value between S_{min} and S_{max}, and then an $N \times N$ square (or its horizontal reflection) will be randomly cropped from the scaled image and fed to the network. For the standard network that is trained using a fixed input size N, scale augmentation comes only from data augmentation and it is equal to $\frac{S_{max}}{S_{min}}$. Standard models like the VGG network [10], and the residual network [1] use a fixed input size of 224×224, and scale the shorter side of the image between $S_{min} = 256$ and $S_{max} = 512$, resulting in a maximum scaling range of $\frac{S_{max}}{S_{min}} = 2$. Using a higher scaling range (while keeping the input size fixed) makes the size of the scaled image much larger than the size of the input crop $N \times N$, which increases the probability of missing out the important details in the image. Figure (4.3) shows an example of using a scaling range $\frac{S_{max}}{S_{min}}$ equal to 2, 3 and 4, and it shows how the important details (of the main object) are lost (are not included in the input crop) as the scaling range is increased.

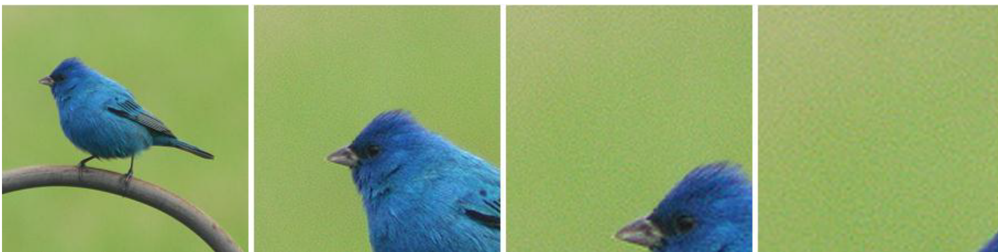

Figure 4.3: original image shown on the left, then the top left corner was cropped with $S_{max}/S_{min} = 2,\ 3,\ and\ 4$ where the randomly chosen scale equal to S_{max}. using the value of 4 cropped out most the details.

The proposed model tries to increase the amount of scale augmentation without cropping out the important details in the input image. This is achieved by training the network using a variable input size. The network is trained using 6 different input sizes (6 different values for

N), and each size has its own S_{min} and S_{max} which are proportional to that size, and where $\frac{S_{max}}{S_{min}}$ is constant across all input sizes. Here scale augmentation comes from data augmentation ($\frac{S_{max}}{S_{min}}$), and from using multiple input sizes ($\frac{N_{max}}{N_{min}}$), where the maximum scaling range is:-

$$ScalingRange = \frac{N_{max}}{N_{min}} \times \frac{S_{max}}{S_{min}} \tag{4.1}$$

For the (160-320) model $\frac{N_{max}}{N_{min}} = \frac{320}{160} = 2$, and for the (160-480) model model $\frac{N_{max}}{N_{min}} = \frac{480}{160}$ $= 3$, which allows them to have double and triple the scaling range of a standard network respectively. Figure (4.4) shows an example of using $\frac{S_{max}}{S_{min}}$ equal to 2, 3 for all 6 input sizes for the (160-480) model, where a maximum scaling range of $\frac{N_{max}}{N_{min}} \times \frac{S_{max}}{S_{min}} = 3 \times 3 = 9$ can be achieved while maintaining more details in the input crop. In comparison, figure (4.3) for the standard network showed how a scaling range of 4 crops out most of the details.

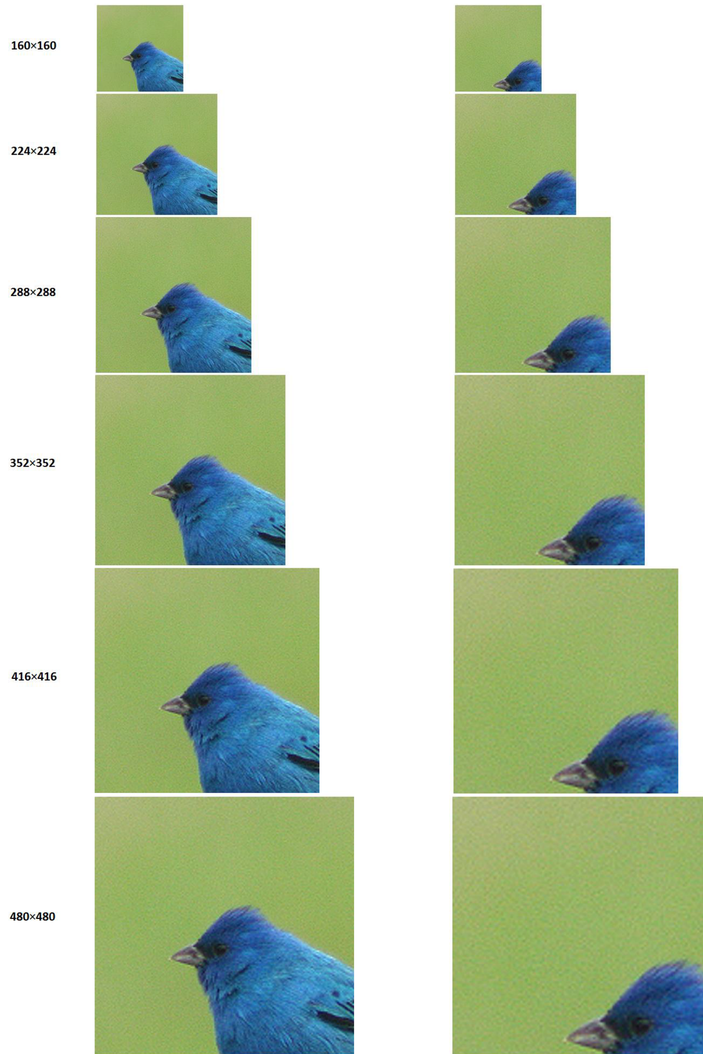

Figure 4.4: Variable input window size network with 6 different window sizes. The left top corner of the input image is cropped using $S_{max}/S_{min} = 2$, and 3 for all 6 input window sizes.

Training and Inference with variable input size

Training and testing a network with a variable window size can be summarized in the following steps. In the training phase, the goal is to strike a balance by allowing enough flexibility, while keeping the implementation simple and efficient. In the inference phase, the goal is to achieve an inference speed that is comparable to the standard implementation.

- In each new iteration a random input size $N_i \times N_i$ is selected from the predefined set of sizes. For each size there is a corresponding scaling range S^i_{min} and S^i_{max}, where the shorter side of the input images will be scaled to a random value between S^i_{min} and S^i_{max}, and then an $N_i \times N_i$ square will be cropped from each image in the current batch to form the input batch that will be fed to the network.

- Using the same input window size $N_i \times N_i$ for all the images in the current batch simplifies the design of the network by utilizing the standard implementation of the convolution operator (utilizing optimized CUDA libraries). Maximum flexibility can be achieved by using a different input size for each image which complicates the implementation of the convolutional layer, and lack of flexibility results from using a single input size for all images (for an entire epoch) which is the case for the SPP network [67] that uses the caffe toolbox [98]. Here we choose something in between, which is to use a different input size for each new batch (iteration). This allows for enough flexibility while keeping the implementation simple and efficient. The only down side is the need for more memory when the biggest input window size is selected because it will be applied to all the images in that batch.

- A variable average pooling size is applied after the last convolutional layer to always shrink the last output channels to a single value. This will keep the number of inputs to the output layer (the only fully connected layer) constant, which in turn will keep the number of weights in the network constant. This will allow the same network to be trained using multiple input sizes.

- The same learning rate and weight decay are used for all input sizes. The experiments showed that normalizing the learning rate according to each input size didn't improve the results, and that using the same learning rate and weight decay for all input sizes works well.

- Inference is carried out using a single input size. From the predefined set of 6 input sizes, the middle input size is selected, and the network is tested using only that size (e.g. the (160-320) model uses 6 sizes 160, 192, 224, 256, 288, 320, and the upper middle size 256 was used). From our experiments the accuracy obtained from using only the middle input

size is very close to the accuracy obtained by averaging the results of all 6 sizes. This is an advantage because although the training process has been made more complex, the inference process is kept the same, which in turn keeps the same inference speed as the standard approach trained using a fixed input size (assuming a single model is used and not an ensemble of models).

4.4 Experiments

The proposed model is implemented by training the 34-layer residual network (figure (2.4)) using a variable input size, and its performance is compared to the performance of the standard implementation that is trained using a fixed input size. To ensure a fair comparison, the proposed model will be compared to a standard model that requires comparable amount of computations. Therefore, the (160-320) model is compared to the standard network trained using a fixed input size of 224×224, and the (160-480) model is compared to the standard network trained using a fixed input size of 320×320. The experiments are carried out using the ImageNet dataset [4] which has 1,281,167 training images divided into 1000 classes. The ImageNet validation set (50,000 images) is used to measure the performance of the network.

4.4.1 Hyper-parameter Tuning

The experiments were carried out using the entire ImageNet dataset, where training takes between one to four weeks (depending on the input size). Thus, carrying out an extensive search to optimize the hyper-parameters is infeasible. For the standard model, we start from the optimal settings used in the pervious chapter to train the network on datasets sampled from ImageNet. Then each experiment is repeated a few times, each time these parameters are tweaked slightly. If there is a noticeable change in the results, then the settings that produced inferior results are discarded, and different settings are tried again. To save time multiple parameters are tweaked at the same run. Once we get 3 runs that produce results that are close to those produced by the best found hyper-parameter setting, the average for those runs is reported as the experiment result. For both standard networks trained using a fixed input sizes of 224×224 and 320×320, we started form the normalized learning rate of 0.001, a weight decay of 0.0005, and a batch size of 128 images. We found that using a smaller weight decay parameter produces slightly better results, and all the results reported here used a weight decay of 0.0001. Weight decay is used to reduce overfitting and networks trained on large datasets such as ImageNet are less susceptible to overfitting. This might explain why a bigger weight decay value (0.0005) works fine for smaller datasets (sampled from ImageNet), while a smaller value (0.0001) produces better results for bigger datasets such as ImageNet. The starting learning

rate was near optimal and decreasing it or increasing it slightly didn't noticeably degrade or improve the results. For the standard network trained using a fixed input size of 320×320, the maximum batch size that can be supported by a 12GB GPU memory is 128 images, and therefore all the experiments for this model were carried out using a batch size of 128 images. For the standard model trained using a fixed input size of 224×224, using a bigger batch size of 256 images improved the results slightly and therefore all the experiments for this model were carried out using a batch size of 256 images.

When training the network using a variable input size, two models were tested in this experiment, the (160-320) model, and the (160-480) model (the two numbers state the lower and upper limits of the input size). The Implementation section outlines how the lower and upper limits of the input size for both models were chosen. Choosing these input sizes was validated using a 100-class dataset sampled randomly from ImageNet. Therefore, these choices were not validated using the entire ImageNet, because that would require a significant amount of training time, where testing a single setting that uses large input sizes requires about 4 weeks. However, the final results on ImageNet show similar trends to the results of 100-classes dataset used to validate these choices, for both the single-crop and multi-crop results.

Existing CUDA library implementations of the convolution operator apply the same convolution across an entire batch of images. Therefore, one of the decisions that was made to simplify the implementation and allow us to use such optimized libraries is to use the same input size for all the images in a single batch. For each new batch one size is randomly sampled from the predefined set of 6 input sizes and all images in the current batch will use that input size. However, when the input size changes, all the input and output sizes of all convolutional layers will change accordingly (as figure (4.2) shows). Therefore, for different batches, the number of accumulative computations required to calculate the derivatives needed to update the weights of the convolutional layers will be proportional to the input size used for that batch. We wanted to see if changing the size of computations in each iteration requires changing the learning rate accordingly to stabilize training. We compared using a single learning rate for all input sizes to using different learning rates (inversely proportional) to different sizes. Our experiments on the 100-classes dataset showed that using a unified learning rate across all sizes is adequate to train the network and achieve good results, and normalizing the learning rate further according to each size didn't improve the results. Therefore, for the final experiments carried out using the entire ImageNet, a single learning rate is used. We started with the normalized learning rate of 0.001 for both the (160-320) and the (160-480) models. This was in the optimal range for the smaller 100-classes dataset used to validate the choices of the input sizes. The experiment was repeated where this learning rate was increased by a factor of $\frac{3}{2}$ to 0.0015, and decreased by a factor of $\frac{2}{3}$ to 0.00066. Increasing the learning rate to 0.0015 didn't change the results, while decreasing it to 0.00066 produced slightly inferior results, therefore the results for the 0.00066

learning rate was discarded, and the experiment was repeated using the 0.001 learning rate.

The same weight decay of 0.0001 that was verified using the standard model is used here. This was done to save time, and we believe that because the network structure is similar, and because both models were trained using the same large dataset, using the same remedy to overfitting is a reasonable choice. For both the (160-320) model, and the (160-480) model the maximum allowed batch size by a 12GB of GPU memory is used. For the (160-320) model the maximum allowed batch size is 128 images, and for the (160-480) model the maximum allowed batch size is 64 images. Using a small batch of 64 images for the (160-480) model is not optimal for a large dataset that is made of 1000 classes. The results for this model will probably be further improved if there was enough resources (GPU memory) that allow for larger batch sizes. Table(4.1) summarizes the hyper-parameters used for both the standard network, and the proposed model trained using a variable input size.

Parameter	Tolerance Range	Used Value	Impact
Learning Rate	(0.000666 - 0.0015)	0.001	significant
L2 reg parameter	(0.0005 - 0.0001)	0.0001	mild
Batch Size (224×224) model	(128, 256)	256	significant
Batch Size (256×256) model	(128)	128	significant
Batch Size (160−320) model	(64, 128)	128	significant
Batch Size (160−480) model	(64)	64	significant
Weight Initialization	lecun [20], glorot [21], kaiming [19]	kaiming [19]	mild
Optimizer	(RMSprop, Adam) + L2 reg	RMSprop + L2 reg	mild
RMSprop Decay Parameter	(0.99-0.9995)	0.999	mild
Learning Rate Decay	(Step, Schedule) Decay	Step Decay by 0.5 5 times	mild
Epochs	100 epochs		
Parameters specific to the proposed model			
Input Size Range	(160-480)	(160-320), (160-480)	significant
Model flexibility	different input size per (image, batch, epoch)	different input size per batch	one tested[1]

Table 4.1: hyper-parameters for the standard model trained using fixed input size, and proposed model trained using a variable input size. RMSprop matches Adam with less memory requirements. Both the maximum input size and maximum batch size were enforced by hardware limitations (GPU memory size of 12GB). Larger values will probably improve the results.

[1]only one implementation was tested where a different input size is used for each new training batch. It is a practical compromise between the other two extreme cases.

4.4.2 Results

Before going into the details of the results, let us clarify the following points. The augmentation method used in these experiments introduces scale augmentation by scaling the shorter side of the image to a random value between S_{min} and S_{max}, then an input square of size $N \times N$ is randomly cropped and fed to the network. The optimal scaling range $\frac{S_{max}}{S_{min}}$ is between 2 and 3 depending on the dataset, and using larger values for S_{max} while keeping the input size N fixed increases the probability of not including important details in the input crop. Therefore, training the network using a variable input size was intended to extend the amount of scale augmentation that can be obtained from data augmentation and was not intended to replace data augmentation. Comparing the two models without data scale augmentation puts the standard model at a disadvantage, because when training the network using a variable input size, implicit data scaling is required to fit the input crop to different input sizes. When data scale augmentation is omitted ($S_{min} = S_{max}$), the gains obtained by our (160-320) model compared to the 224×224 standard model (24.8% to 23.12%) are bigger than the gains obtained when data scale augmentation is used (21.9% to 21.25%). The higher gains obtained when data scale augmentation is omitted are expected because our model trained using a variable input size is implicitly using data scale augmentation, and therefore is not suffering as much.

Second, the performance of the network is often measured by averaging the predictions of multiple crops from the same image taken at different scales between S_{min} and S_{max} and this is called the multi-crop or multi-view result. All results reported in this thesis are multi-crop results, unless stated otherwise. The other performance index is the single-crop result obtained using a single central crop from the test image. Because the two performance indices show different gains, both results are reported.

Table (4.2) compares the performance of the standard model with input size equal to 224×224 to the performance of the proposed model trained using 6 different input sizes (160, 192, 224, 256, 288, and 320) that requires comparable training time. Because the experiments show correlation between the input size and single-crop performance, inference for both models was carried out using the same single input size of 256×256 (for fairness). The table shows both the single crop and the multi-crop results averaged over 3 runs. The proposed model improves both the single-crop and the multi-crop results, however the single-crop results show more gains than the multi-crop results. Both gains are statistically significant as table (4.4) shows.

	224×224 Stadard Model	(160−320) Variable Model
Multi-crop Results	21.9% ±0.29	21.25% ±0.2
Single-crop Results	27.1% ±0.226	25.2% ±0.26

Table 4.2: the single and multi crop results for the 224×224 standard model vs the proposed (160-320) model.

Table (4.3) compares the performance of the standard model with input size equal to 320×320 to the performance of the proposed model trained using 6 different input sizes (160, 224, 288, 352, 416, and 480) that requires comparable training time. The inference for both models was carried out using a single input size of 352×352. The table shows both the single crop and the multi-crop results averaged over 3 runs. The (160-480) model improves both the single-crop and the multi-crop results, however the single-crop results show more gains than the multi-crop results. The gains of the multi-crop are not statistically significant as table(4.4) shows, probably because the sample size is small (only 3 runs), and because of resource limitations, the batch size used for the (160-480) model was small (64 images).

	320×320 Stadard Model	(160−480) Variable Model
Multi-crop Results	21.5% ±0.27	20.9% ±0.236
Single-crop Results	25.6% ±0.25	23.9% ±0.19

Table 4.3: the single-crop and multi-crop results for both the 320×320 standard model, and the proposed (160-480) variable model.

	p-value from t-test, sample-size = 3	
	multi-crop results	single-crop results
Standard 224×224 model vs (160−320) proposed model	0.041	0.00068
Standard 320×320 model vs (160−480) proposed model	0.052	0.0011

Table 4.4: p-values that compares the 224×224 standard model with the (160-320) proposed model, and the 320×320 standard model with the (160-480) proposed model. All gains are statistically significant for a level of significance $\alpha = 0.05$, except for the multi-crop result of the (160-480) model.

4.4.3 Single-crop performance

Based on the results in tables (4.2, 4.3), the proposed model trained using a variable input size improves the single-crop results more than it improves the multi-crop results. Also, based on the results in tables (4.2, 4.3) increasing the input size improves single-crop results more than it improves the multi-crop results for the standard network. There is a connection between these results because a model trained using a variable input size is usually trained using a larger maximum input size than the standard counterpart that requires a comparable training time. For example, the (160-480) model is trained using a maximum input size of 480×480 while the standard counterpart is trained using a fixed input size of 320×320. This section looks at how increasing the input size, and using a variable input size both improve the single-crop performance.

First let's look at the impact of increasing the input size on the single-crop performance of the standard network trained using a fixed input size. To thoroughly measure the impact of the input size on the single-crop performance of the standard network, a third additional model that is trained using a bigger fixed input size of 480 × 480 is tested. This model took the longest training time and was trained three times using the same learning rate and weight decay used by the smaller models and using the maximum allowed batch size of 64 images. Table (4.5) shows the single-crop and multi-crop results for all three standard models. Table(4.6) shows the statistical significance (p-values) of the single-crop and multi-crop gains obtained by increasing the input size from 224 × 224 to 320 × 320 to 480 × 480. The p-values clearly shows that the single-crop gains are statistically significant while the multi-crop gains are not for a level of significance $\alpha = 0.05$.

	224×224 Stadard Model	320×320 Stadard Model	480×480 Stadard Model
Multi-Crop Results	21.9% ±0.29	21.5% ±0.27	21.37% ±0.231
Single-Crop Results	27.1% ±0.226	25.6% ±0.25	24.5% ±0.247

Table 4.5: the results show that the single-crop performance improves as the input window size increases, while the multi-crop performance improves only slowly.

	p-value from t-test, sample-size = 3	
	multi-crop results	single-crop results
224×224 vs 320×320 fixed input size	0.158	0.00166
320×320 vs 480×480 fixed input size	0.554	0.00572

Table 4.6: It is very clear from these p-values that increasing the input size for the standard model from 224 × 224 to 320 × 320 to 480 × 480 significantly improves the single-crop performance of the network, while the improvements to the multi-crop performance is not statistically significant, for a significance level $\alpha = 0.05$.

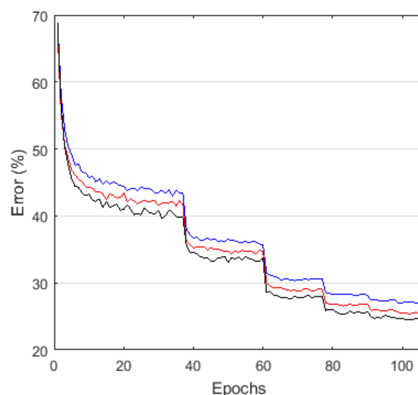

Figure 4.5: test error measured using the central crop of the test images. The **blue curve** is for input size 224 × 224, the **red curve** is for input size 320 × 320, and the **black curve** is for input size 480 × 480.

Now lets look at the impact of using a variable input size on the single-crop performance. To thoroughly measure such impact, a third additional model trained using 3 different input sizes (160, 192, and 224) was implemented and tested (we call this the (160-224) model). The idea is to compare the single-crop performance of two models that are using the same maximum input size. Compare the (160-224) model to the 224×224 standard model, the (160-320) model to the 320×320 standard model, and the (160-480) model to the 480×480 standard model. Because in all three cases, both models are trained using the same maximum input size of 224×224, 320×320, and 480×480 respectively, they are getting the same advantage obtained from using larger input sizes (discussed in the previous paragraph) for the single-crop performance. Therefore, any difference in the results will be attributed to using a variable input size vs using a fixed input size. Table (4.7) compares the single-crop results for these models, and figure (4.6) is a visualization of these results. It is clear that the (160-224), (160-320) and (160-480) variable input models outperform the 224×224, 320×320, and 480×480 standard counterparts respectively (although not all differences are statistically significant, p-value is 0.119, 0.104, and 0.0344 respectively). What the results in table (4.7) or figure (4.6) show is that the single-crop performance gains of the proposed model (trained using a variable input size) can be attributed to 2 factors. The first factor is using a variable input size instead of a fixed input size, and the second factor is using larger input sizes. As a result, the (160-480) model produced the best single-crop performance (23.9%) because it was trained using a variable input size, and because it has used the largest tested size of 480×480. Also the (almost) parallel lines in figure (4.6) show that the gains obtained from using larger input sizes are orthogonal to the gains obtained from using a variable input size.

Maximum Input Size	224×224	320×320	480×480
Fixed Input Size	27.1% ± 0.226	25.6% ± 0.25	24.5% ± 0.247
Variable Input Size	26.75% ± 0.23	25.2% ± 0.26	23.9% ± 0.19

Table 4.7: the single-crop results for both the standard model, and the proposed model, reported at 3 different input sizes (for the proposed model these are the maximum input sizes).

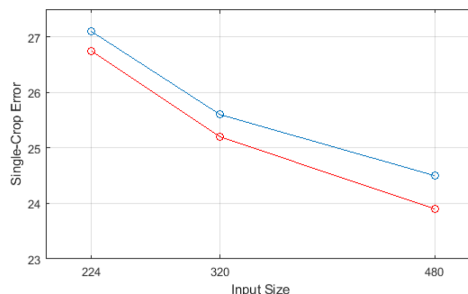

Figure 4.6: Visualization of the results in table(4.7). **Blue Curve** for the standard model, **Red Curve** for the proposed model. The curves show that both increasing input size, and using a variable input size improve the single-crop performance.

4.4.4 Discussion and conclusions

This section discusses the results from a few different angles. First we look at the memory requirements for the proposed models trained using a variable input. We also discuss the model and results in relation to the SPP network [67], which is a similar implementation in the literature. Finally, we discuss the importance of single-crop performance for tasks such as object detection.

In the results section when comparing the multi-crop results, the proposed model trained using a variable input size was compared to a standard model that requires comparable training time. This was done to make the comparison fair. The performance of the (160-320) variable model was compared to the performance of the 224×224 standard model, and the performance of the (160-480) variable model was compared to the performance of the 320×320 standard model. The one disadvantage here for the proposed model is that it requires more memory than the standard implementation. Roughly, the (160-320) model requires $\frac{320^2}{224^2} = \mathbf{2.04}$ times the memory required by the 224×224 standard model, and the (160-480) model requires $\frac{480^2}{320^2} = \mathbf{2.25}$ times the memory required by the 320×320 standard model. This high memory requirement is a result of using the same input size for all images in a single batch, which simplifies the implementation. As a result, the highest batch size we were able to use with the (160-480) model was 64 images, which was small (not optimal) for ImageNet. This disadvantage of using more memory can be solved by using different input sizes for different images in the same batch, and choosing these sizes in a way that keeps the average size below a certain limit. However, this requires a significant change to the implementation of the convolution layer and will prevent us from using optimized CUDA libraries. Using a different input size for each image is the extreme case, and we choose to simplify the design and use a different input size for each new batch. This is still much more flexible than using a single input size for a whole epoch as is the case for the SSP network implementation [67]. The implementation of SSP network [67] was done using the Caffe [98] software package, which only allows for training the network using a fixed input size. In order to overcome this problem, and use this software tool to train the network using a variable input size, they set up two standard networks with two different input sizes, and at the end of each epoch they switched from one network to the other. They were able to train the same weights using both input sizes because they used variable pyramid pooling to keep the number of parameters fixed even if the input size changes. However, they found it impractical to switch between networks at each new iteration (each new batch), because the overhead would severely slow down the training process. In our case, the code was written from scratch (using CUDA libraries when necessary), and changing the standard implementation so that each new batch can use a different input size was straight forward.

Both the SSP network [67] and our implementation show improvements to the (multi-crop)

results when training the network using a variable input size. However, the extra result the we included that was not reported in [67] is that training the network using a variable input size improves the single-crop results much more than it improves the multi-crop results. I believe the reason (why they did not report it) is because their implementation did not use an input size that is greater than the one used by the standard network which was 224×224. They only used two different input sizes equal to 180×180 and 224×224. On the other hand, our results showed that the main reason for the improvement in the single-crop performance is using bigger input sizes. Our proposed models used bigger input sizes than the standard implementation (that requires comparable training time). The (160-320) model used a maximum input size equal to 320×320, while the standard counterpart used a fixed input size equal to 224×224, and the (160-480) model used a maximum input size equal to 480×480, while the standard counterpart used a fixed input size equal to 320×320.

Multi-crop results are the main performance index when measuring the performance of CNNs used for image classification (because multi-crop inference is cheap). Improving the multi-crop performance by adding more layers is cheaper than using larger input sizes (linear vs quadratic increase in size and computations). For this reason many main stream CNN implementations [9, 10, 1, 12, 94] for classification choose to use many layers and relatively a small input size (e.g. 224×224). For more complex tasks such as object detection, inference per object is intrinsically single-crop inference. Main stream CNN for object detection are trained using much larger input sizes (e.g. 800×800). The justification (from the literature [92, 93, 13, 97, 14, 96]) of using larger input sizes is because images may contain many objects and training the network using larger input sizes allows the network to capture all the details in the image. However, based on the results presented in this chapter, the other plausible reason is because the performance of object detection networks is measured using a single-crop per object, and our results show that single-crop performance improves significantly when using larger input sizes. Tasks such as object detection may benefit more from our implementation because it improves the single-crop results more than it improves the multi-crop results. Training object detection networks using a variable input size, allows the network to be trained using much larger input sizes (even larger than 800×800) without increasing the training time, which (depending on the training set) can improve the single-crop performance of the network. Networks for object detection are more complex to implement and optimize and require much more resources (GPU memory and computation time) than what is available to us because they are trained using much larger input sizes. Therefore, we chose to focus on image classification in our research.

CHAPTER 5

Discovering Hierarchical Structures in the Training Data

5.1 Introduction

The experiments in this chapter investigate the ability of standard convolutional networks to discover hierarchical structures in the training data. The results presented show that a plain convolutional network was able to discover such structures without incorporating hierarchical classification into the design of the network. Our experiments and results do not rely only on analysing the confusion matrix (as is the case for generic datasets) but on measuring the performance of the network using a hierarchical dataset that was specifically constructed for this purpose. These results (in addition to transfer learning) might explain why convolutional networks are very effective in solving large recognition problems.

One of the strengths of deep CNN is their ability to learn how to classify images from large datasets with up to thousands of classes like ImageNet. Since 2012, and starting with AlexNet [9], all the winners [48, 10, 23, 1, 37] of the classification competition part of the ImageNet challenge were deep convolution networks. The first set of experiments presented in this chapter measures the resilience of CNNs in solving large recognition problems. Using multiple datasets (sampled randomly and uniformly from ImageNet) with different sizes, the results show that the performance of the network drops slowly as the number of classes in the training set increases significantly. The results show that the error rate roughly doubles when the number of classes increases 10 times.

The main experiments in this chapter may shed the light on some of the capabilities of deep convolution networks that makes them effective in solving large recognition problems with a large number of classes. In this experiment a hierarchical dataset made up of multiple coarse categories (sampled from ImageNet) was used to train a deep convolution network, and the results of this network were compared to the results obtained by learning each coarse category on a separate network. The performance of the shared network trained on all categories slightly surpassed the overall performance of the all the networks trained on single categories. This means that the shared network was able to discover the hierarchical structure of the training data and was able to break down the main task (based on that hierarchy) into smaller tasks and learn them simultaneously and efficiently with no drop in performance.

Finally, multiple attempts were tried to incorporate the hierarchical structure of the dataset into the network implementation. This was done by incorporating the labels of the coarse categories, in addition to the class labels, in the network implementation. The best incorporation was based on a t-SNE visualization which reveals that the network learns to differentiate between coarser categories in earlier stages, while it learns to differentiate between finer categories (individual classes) in later stages in the network. Based on this analysis the coarse category labels were incorporated at an earlier stage compared to the standard class labels.

5.2 Related Work

This section compares the results, findings, and conclusions presented in this chapter with related work in the literature. In the main experiment, a plain convolutional network was able to discover that a dataset is made up of multiple coarse categories by achieving a recognition rate for each of these categories similar to that achieved by learning each coarse category on a separate network. This shows the ability of convolutional networks to discover hierarchical structures in the training data without any modification to the standard design of these networks. What is interesting about these results is that they were achieved without incorporating hierarchical learning and classification into the implementation of the network. Incorporating the hierarchical structure of the data into the design of the network can improve its performance as the results presented at the end of the chapter show.

The main stream approach of discovering hierarchical structures in generic datasets is to use a standard classifier to generate a confusion matrix and then apply a clustering algorithm (spectral clustering is often used) to that matrix to form a tree-like structure. Then that hierarchical structure can be incorporated into the design to improve the performance of the standard classifier, resulting in a hierarchical classifier. For linear classifiers such as SVMs, hierarchical classification is used mainly to speed up computations. With one-vs-all SVMs the number of required binary classifiers increases linearly as the number of classes increases, and using hierarchical classification (which only compares similar classes) cuts the number of required binary classifiers significantly. This problem doesn't exist with deep CNNs when only the size of the output layer changes as the number of classes increases. Bilal et al [68] examined the implementation of many CNN models, and found that main stream CNNs did not make any use of hierarchy information in their implementation. Here we look at two attempts [69, 68] to incorporate hierarchical classification in the implementation of deep CNNs, and compare their structure and findings to ours. We also look at a study by Deng et al [70] which investigates applying shallow classifiers to large datasets, and show how similar their behaviour to that of CNNs.

First we look at the HD-CNN [69] model, which is short for Hierarchical Deep CNN, which is an attempt to incorporate hierarchical classification with convolutional networks. HD-CNN [69] used the confusion matrix generated by a standard CNN on the ImageNet dataset to implement a two-level hierarchical convolutional network where the first level is made of a single coarse-category classifier and the second level is made of multiple fine-category classifiers. To reduce the cost of the implementation and make it practical in terms of size and computation time, all classifiers share most of the layers and only few layers are specific. As a baseline network they used the NIN [95] model, and the VGG [10] model. This implementation mimics the standard hierarchical implementation that is usually used with shallow classifiers, where a

dedicated classifier is used at each internal node in the hierarchy. Our attempt to incorporate hierarchical classification to CNNs (presented at the end of this chapter) is different from HD-CNN [69], and it has some resemblance to the implementation by Bilal et al [68]. Bilal et al [68] incorporated the hierarchical structure of the training data into the implementation of the convolutional network differently, and in a way that only inceases the computation cost marginally. Linear classifiers were trained on the features generated by convolutional layers at different depths, and then spectral clustering was applied to the confusion matrices generated by those classifiers. Then an auxiliary output layer was attached to each of those investigated layers with the number of outputs reflecting the number of coarse categories generated by the corresponding confusion matrix. They used AlexNet [9] as a baseline network. In our implementation the hierarchical structure of the data is known, where the data is divided hierarchically into 10 coarse categories, and as a result we only needed to incorporate a single auxiliary output layer that predicts the coarse category label. The position (depth) of this extra output layer were decided based on the hints obtained from t-SNE projections.

Although, we achieved performance gains by incorporating hierarchical classification, the main results in this chapter were achieved using a standard convolutional network. The main difference between our study and those [69, 68] presented in the previous paragraph, is that our main results were achieved using a standard CNN applied on a hierarchical dataset, while their results were achieved using a hierarchical CNN applied on a generic dataset (ImageNet). Despite these differences, two of the findings by Bilal et al [68] are similar in some way to the results presented in this chapter, although they have been reached using different means. First, they found that CNNs are capable of discovering the hierarchical structure of the data or part of it. They reached this conclusion by reordering the confusion matrix according to the WordNet hierarchy [89] which was used to construct the complete version of ImageNet, and the resulting matrix formed non-overlapping squares with different sizes along the diagonal. This implied hierarchical confusion between the classes. We reached a similar conclusion by a more controlled experiment and by comparing the performance of the network trained on the hierarchical data with the performances of networks trained on individual coarse categories. Second, they found out that earlier stages in the network develop feature detectors that are capable of differentiating between large categories with big noticeable differences between them, while the latest stages develop feature detectors that are capable of differentiating between individual classes that might have small subtle differences between them. They reached this conclusion by training linear classifiers on the features generated by convolutional layers at different depths, and then investigating the confusion matrices of these classifiers. They found that earlier layers produce coarser confusion patterns, while later layers produced finer hierarchical confusion patterns. In our controlled experiment both the fine class label and the coarse category label of each image are known, and therefore we were able to reach a similar conclusion by directly projecting the

features of individual images at different layer depths. The projections of earlier layers show more generic patterns that differentiate between coarse categories, while the latest stages show finer patterns that separate between individual classes. Reaching the same conclusions using completely different approaches, further justifies these conclusions.

Deng et al [70] investigated applying shallow classifiers (SVMs, k-NN) to large datasets with many classes. They experimented with 1k ImageNet (standard benchmark), 7k ImageNet (7404 leaf classes), and 10k ImageNet (10804 classes, the entire ImageNet release of 2009). They also experimented with smaller datasets (with sizes between 134 and 262 classes) sampled from Image Net to form diverse and dense datasets. Although they didn't use CNNs in their experiments, two of their findings match similar conclusions reached in this chapter. In their experiments with the small datasets they found that the performances of all the used classifiers is significantly influenced by how dense or diverse the dataset is. Dense datasets (with many similar classes) are more challenging with much higher error rates compared to more diverse datasets. In our experiments with CNNs, dense datasets made of 10 similar classes (single coarse category) had an average error rate of 17.03%, while diverse datasets with 10 classes had an average error rate of only 4.8%. Both findings show that the density of the dataset (ratio of similar classes and how similar they are) is a better indicator of how difficult it is to classify that dataset than the number of classes in the dataset. Their [70] second finding states that the performance of the classifier drops at a slower rate compared to the increase in the number of classes in the dataset. Surprisingly they noticed a very similar behaviour for the SVM and k-NN classifiers to our findings about CNNs, where the error rate only increases by a factor of 2 as the number of classes increases by a factor of 10. This indicates that other classifiers can also discover hierarchical structures in the data, where confusion happens mainly between similar classes. However, CNNs are still able to do that at a much higher performance point with much lower error rates.

5.3 Network performance vs problem size

The first experiment in this chapter tries to measure how effective convolutional networks are in solving large problems. This is done by measuring the error rate for multiple datasets with different sizes, and examining how the error rate increases as the number of classes increase. Datasets with 5 different sizes were randomly and uniformly sampled from the 1000-classes ImageNet. The number of classes of these datasets were 10, 50, 100, 500, and 1000 classes. To reduce variance in the results, multiple datasets were sampled at each size (except the last one) and the average performance was reported. The number of datasets sampled at each size were 10, 10, 5, 2, 1 respectively. The ImageNet dataset used here is the one used in the ILSVRC 2015 competition, and 891 out of the 1000 classes had the maximum number of training images which

is equal to 1300 images. All the classes sampled here are from those 891 classes, and therefore all the classes used in this experiment had 1300 training images (expect for the 1000-classes dataset which uses all the classes in ImageNet). All test sets are sampled from the ImageNet validation set, and therefore each class has 50 test images.

The experiments are carried out using the 34-layer residual network shown in figure (3.1). To measure how the performance of the network is affected by increasing the size of the dataset, the same network structure need to be used for all datasets (the only difference being the number of neurons in the output layer which is equal to the number of classes in the corresponding dataset). This is not the main experiment in this chapter, and therefore we will briefly discuss the process of setting up the hyper-parameters. In short, for the smaller datasets (with 10, 50 and 100 classes) a weight decay equal to 0.0005 is used, which is higher than the weight decay used for the bigger datasets (with 500 and 1000 classes) which is equal to 0.0001. All data sets used a normalized learning rate equal to 0.001. For smaller datasets the batch size is equal to the number of classes (batch size equal to 10, 50, and 100 images for datasets with 10, 50, and 100 classes respectively), for the 500-class dataset the batch size was set to 128 images, and for the full 1000-class ImageNet the batch size was set to 256 images. The same weight initialization method, color augmentation method, and optimization method used in chapter three for datasets sampled randomly from ImageNet are used in this experiment. The detailed justification for these selections was presented in similar experiments in the previous chapters. Finally, to reduce overfitting especially for the smaller datasets, the same aggressive data augmentation method used in chapter three is used here. The size of the cropped square is chosen randomly to be between 8% and 100% of the size of the input image, and the aspect ratio is changed randomly to be between 3/4 and 4/3.

Table (5.1) shows the multi-crop results for all 5 dataset sizes. The results show the error rate increasing by a factor close to 2 from 4.81% to 10.1% to 21.8% as the number of classes increases by a factor of 10 from 10 to 100 to 1000 classes. This shows that the error rate increases at a much slower rate compared to the increase in the number of classes, which shows how effective these networks are in solving very large problems. The exact values of these results may vary based on the makeup of the datasets which are randomly sampled from ImageNet, and to reduce this variance, multiple datasets are sampled at each size. However, despite the small variance the error rate always grows at a much lower rate compared to the increase in the number of classes.

Data Size	10 Classes	50 Classes	100 Classes	500 Classes	1000 Classes
Test Error	4.81%	7.7%	10.1%	16%	21.8%

Table 5.1: Results for datasets with different sizes sampled from ImageNet.

Figure (5.1) shows the relative increase in the error rate compared to the relative increase in the number of classes (here the relative increase in the number of classes reflects the relative increase in the amount of training data because most classes have the same number of training images). The relative increase in the error rate and the relative increase in the number of classes are obtained by dividing all entries in table (5.1) by the entries in the first column. As the number of classes is increased up to 100 times, the error rate only increases 4.5 times. This shows the resilience of the performance of deep convolution networks toward increasing the size of the recognition problem. One of the known factors that makes deep CNNs effective in solving large recognition problems is transfer learning, where such networks develop better feature detectors if they are trained on larger and more diverse datasets. Transfer learning happens between classes that share features.The results presented in the next section, show the other side of the coin, where the network learns multiple coarse categories that often do not share low or medium level features.

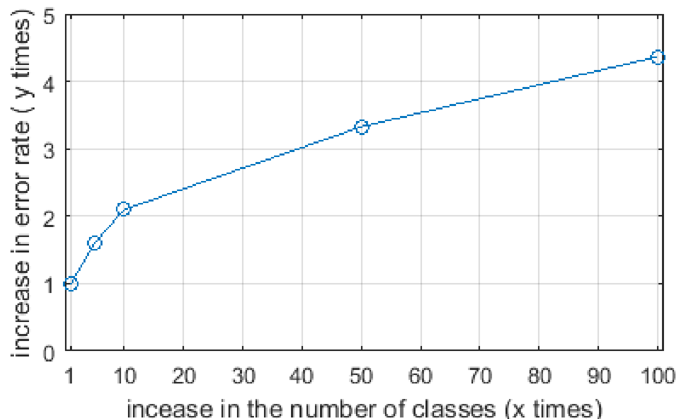

Figure 5.1: the relative increase in error rate compared to the relative increase in the number of classes.

5.4 Learning Hierarchical Data

As the previous experiment showed, one of the strengths of deep convolution networks is their ability to effectively solve large recognition problems such as the 1000-classes ImageNet. It also showed the performance dropping slowly as the size of the task increased significantly. Large datasets such as ImageNet usually exhibit hierarchical structures where there are many similar classes that can be be grouped into coarser categories (e.g. multiple species of dogs, cats, birds, multiple types of cars etc.). The main experiment presented in this section shows the ability of standard convolutional networks to discover such hierarchical structures. These results (in addition to transfer learning) might explain the success of these networks with large datasets.

The methodology used in this experiment is different from the standard methods that usually rely on the confusion matrix of the classifier generated using generic data. In this experiment a hierarchical dataset was constructed from ImageNet where classes are naturally divided into coarse categories.

5.4.1 Constructing a Hierarchical Dataset

A dataset made up of 100 classes was constructed from ImageNet ILSVRC 2015, where the chosen classes belong to 10 different coarse categories. These categories are, birds, bugs, cars, cats, china and cookware, fruits and vegetables, furniture, lizards, monkeys, and snakes. Each of these naturally divided categories consists of 10 classes, for example the cars category is divided into ambulances, jeep (four wheel) cars, family cars, convertible cars, police cars, taxis, sports cars, small buses, large family cars, pickup cars. Figure (5.2) shows ten images from each category, one for each class in that category (total of 100 classes). The training dataset is sampled from the ImageNet training set, with 1300 images per class, 13000 images per category, and 130000 images in total for all 10 categories. The test set is sampled from the ImageNet validation set, with 50 images per class, 500 images per category, and 5000 images in total for all 10 categories. For reproducing the results, table (B.6) shows the folder names of the classes that make up all categories, where each row represents a single category. Throughout this section the word category (or coarse category) means a group of similar classes (in this case 10 classes).

5.4.2 Experiment

Using the hierarchical data, the main experiment can be divided into two parts. In the first part, all 10 coarse categories (all 100 classes) are treated as a single standard dataset and used to train a single deep convolution network. The plain implementation of the network was not changed to incorporate the hierarchical structure of the data (only standard one hot class labels are used). In the second part of the experiment, each coarse category (made up of 10 classes) was considered as a separate dataset, and was used to train a separate convolutional network. Therefore, 10 separate networks were trained using the 10 different categories, where the size and structure of these networks is the same as the size and structure of the shared network used to learn all categories (the only difference is the size of the output layer which reflects the number of classes). For each category, the recognition rate achieved using the shared network (trained using all categories) is compared to the recognition rate achieved using a separate network trained only using that category. This comparison will measure if there is a drop in performance per category for the shared network. Such a drop should reflect the added confusion caused by learning all the categories on the same network. If there is no drop (or the drop is very

small), then that indicates that the network was able to distinguish between the different coarse categories (able to discover the hierarchical structure of the data), where confusion is restricted by the network to only happen between similar classes that belong to the same category.

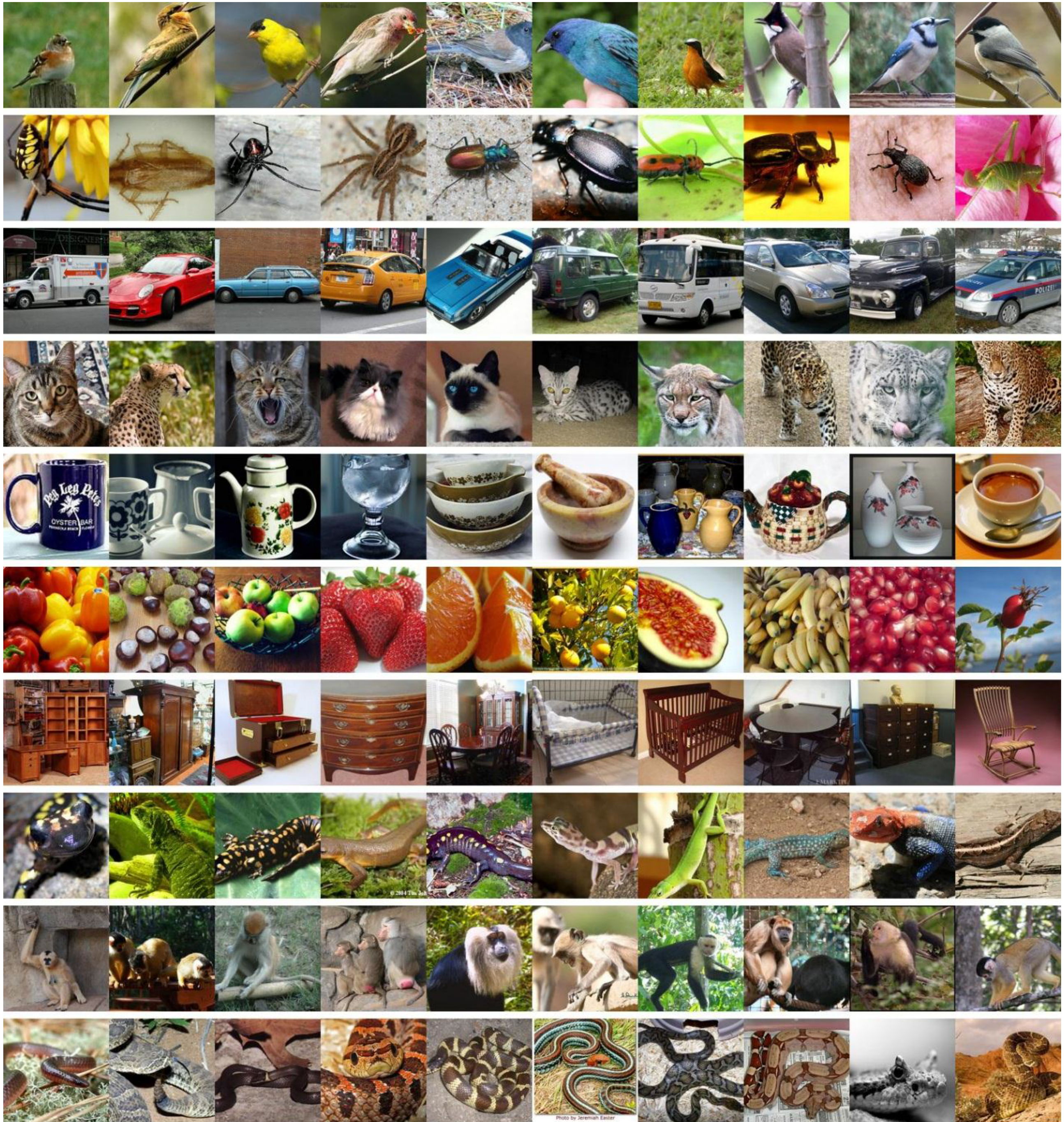

Figure 5.2: each column is composed of ten images that belong to one of the 10 categories. Each of the ten images belong to a different class in that category. Images are scaled down and the center square crop is used for clearer display.

Setting the hyper-parameters

The 34-layer residual network shown in figure (3.1) is used for this experiment. Changing this standard implementation, by changing the depth or width of the network, or by changing the positions of the max-pooling stages, does not change the conclusions of this experiment. As it has been explained in previous chapters, this residual implementation contains most of important components (BN, residual connections etc.) that are still used today to achieve competitive results. Therefore, we expect our conclusions to hold for state of the art residual models [12, 94]. In comparison the HD-CNN [69], and Bilal et al [68] implementations use older baseline network models such as the NIN [95], AlexNet [9], and the VGG [10] model.

Out of the 1300 training images per class, 100 images were set aside to validate the selection of the hyper-parameters. Once the hyper-parameters are selected, all the training images are used to train the network(s). These datasets are small, and therefore we were able to do an exhaustive search for the hyper-parameters which include the learning rate, weight decay, batch size, and the RMSprop (optimizer) decay parameter. Using the validation set, we found that a weight decay of 0.0005, an RMSprop decay of 0.999, and a normalized learning rate of 0.001 works well for both the single network trained on all 10 categories (all 100 classes), and for the 10 networks each trained on a single category (10 classes). Also a batch size equal to 10 images works well for the 10 networks each trained on a single category. These are similar to the hyper-parameters validated in the experiments carried in chapter three for small datasets (with 10, 50 and 100 classes) sampled randomly from ImageNet. However, there was one hyper-parameter that did not match those used in chapter three which is the batch size used for the network trained on all 10 categories. For small and medium datasets that are randomly sampled from ImageNet (as the experiments in chapter three) we found that using a batch size equal to the number of classes always produces good results. For the 100-classes hierarchical dataset used in this experiment which is naturally divided into 10 coarse categories, we found that using a smaller batch size (20 or 25 images) produced better results than using a batch size equal to the number of classes (equal to 100 images). The updated result obtained using a batch size equal to 25 images implies that the optimal batch size for a hierarchical dataset like ours is different from the optimal batch size for randomly and uniformly constructed datasets.

Using The same network structure for both the single network trained on all 10 categories and for the 10 networks each trained on a single category makes the comparison fair. The only difference is the size of the output layer which reflects the number of classes in the dataset (10 vs 100 neurons). The aggressive data augmentation method used in the previous experiment is used here to reduce overfitting especially for the networks trained using a single category. Table(5.2) summarizes the main hyper-parameters.

Parameter	Tolerance Range	Used Value	Impact
Learning Rate	(0.001 - 0.0016)	0.001	significant
L2 reg parameter	(0.00005 - 0.005)	0.0005	mild
Batch Size Shared Network	(20-25)	25	significant
Batch Size 10 Separate Networks	(10-20)	10	significant
Weight Initialization	lecun [20], glorot [21], kaiming [19]	kaiming [19]	mild
Optimizer	(RMSprop, Adam) + L2 reg	RMSprop + L2 reg	mild
RMSprop Decay Parameter	(0.99-0.9995)	0.999	mild
Learning Rate Decay	(Step, Schedule) Decay	Step Decay by 0.5 5 times	mild
Epochs	100 epochs		

Table 5.2: hyper-parameters for both the shared network trained on all 10 coarse categories and for the 10 networks each trained on a single coarse category. The same parameters are used for models except the batch size. Using a small batch size for the shared network is important to achieve good results.

Results

Table (5.3) shows the results per category for both parts of the experiment. These results are achieved by averaging the results of five runs, each carried out using the optimal hyper-parameter setting from the previous section. The top row shows the results per category obtained using 10 separate networks, while the bottom row shows the results per category obtained using a single shared network. The last column shows the average results for all 10 categories. For most categories the results were very close, and interestingly the overall performance of the shared network slightly exceeded the average performance of the 10 networks (16.78% compared to 17.03%, although not statistically significant). It is not accurate to use the standard two-samples t-test to measure the statistical significance of the difference between the overall performance of the main network trained on all coarse categories and the overall performance of the 10 networks each trained on a single coarse category. The reason is because the overall performance of the 10 networks is the average of 10 independent runs and matching different runs (out of total 50 runs, 5 per network) produces different averages and therefore different p-values. The proper way is to use the paired-t-test where there is a pair of values for each coarse category, one obtained using the main shared network and one obtained using one of the 10 networks trained only using that specific category. Here the average result of 5 runs per coarse category is used to construct both samples (the entries in table(5.3)), and the size of each sample is equal to the number of coarse categories (10). Using a two-tailed paired-t-test, the p-value is 0.3325, and

for the standard significance level $\alpha = 0.05$ it is clear that the difference between the overall performance of the 10 networks each trained on a single coarse and the overall performance of the single main network trained on all categories is not statistically significant.

From the results in table (5.3) and from the p-value of the paired-t-test, the shared network recognized that the 100 classes belong to 10 different categories, and was able to learn all of them with accuracy comparable to learning each one on a separate network. Similar conclusions can be reached by examining the confusion matrix of the shared network which shows that confusion mainly happens between the classes that belong to the same coarse category (the next section examines the confusion matrix of the shared network). However, even if the confusion matrix shows clear separation between the different coarse categories, it can not measure if there is a drop in the recognition rate for each of those categories. Our direct approach which measures the recognition rate twice for each category (once using the shared network, and once by training that category on a separate network) shows if there is a drop in the performance of the shared network for each of the categories.

	Cars	Bugs	Cats	China	Birds	Fruits	Furniture	Lizards	Monkeys	Snakes	**Avg.**
Network Per Cat.	16.6%	12.6%	18.1%	22.9%	2.43%	10.4%	16.4%	22.7%	23.4%	24.5%	**17.03%**
Shared Network	16.2%	11.8%	18.2%	24.0%	2.26%	9.19%	17.0%	22.3%	22.8%	24.1%	**16.78%**± 0.285

Table 5.3: Results per category for the shared network vs the results obtained using separate network per category.

Table (5.3) shows that the performance of the network trained on all categories is equal to the average performance of the 10 networks each trained using a single category. This shows that the performance of a deep CNN used to learn a dataset that can be hierarchically broken into smaller datasets is equal to the average performance of learning each of the smaller datasets individually using a separate deep CNN. In reality however, very big datasets such as ImageNet have a mixed bag of classes, where some can be separated into categories (cars, cats, dogs etc.), and others that do not belong to a specific group or category. For such datasets, the performance of deep convolution networks will be affected by both, the difficulty of distinguishing between similar classes (the members of the same category), as well as the size of the dataset (the total number of classes).

What is noticeable about these results, is that increasing the number of classes 10 fold (from 10 to 100 classes) did not increase the error rate, but rather it was reduced slightly from 17.03% to 16.78% (although within the margin of error). By comparison, the previous experiment in this chapter, for randomly constructed datasets, showed that increasing the number of classes 10 fold roughly doubles the error rate (increases the error rate by 100%). This shows how effective standard convolutional networks are in classifying hierarchical data, and it is worth repeating that these results were achieved without incorporating the hierarchical structure of the data into the implementation of the network.

To further put the scale of these results into prospective, figure (5.3) compares the main results in table (5.3) obtained using our hierarchical dataset with results obtained using a 100-classes dataset, that is randomly constructed and randomly divided into 10 groups. The yellow bars show the error rates per category or group when each category or group is learned separately, and the blue bars show the error rates per category or group for the shared network. By visually comparing the two figures, the shared network preformed much better with the hierarchical data (left), than it did with the random data (right). For the hierarchical data, the shared network trained on all 10 coarse categories succeeded in matching the performance of the 10 networks each trained on a single category. For the randomly divided data, the shared network trained on all 10 groups failed to match the performance of the 10 networks each trained on a single group.

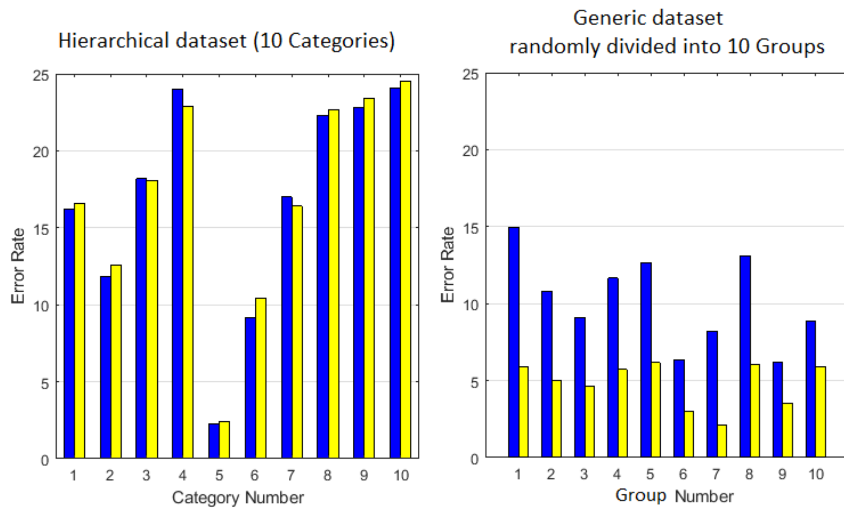

Figure 5.3: blue bars show the error rate per category or group obtained using a single network, yellow bars show the error rate obtained using a separate network per category or group. **left:-** for the naturally divided dataset, **right:-** for the randomly divided dataset.

5.4.3 Analysing the Confusion Matrix

Confusion matrices are usually analysed by applying a clustering method (usually spectral clustering) to see if the confusion patterns of a classifier form a hierarchical structure. Before investigating the confusion matrix of our main network trained on all coarse-categories, let's address the limitations of applying spectral clustering to construct the hierarchical structure of the data from the confusion matrix. Spectral clustering relies on constructing a similarity matrix between all the input data points in the dataset, and the number of rows in such symmetric square matrix is equal to the number of entries in the dataset. Clustering is carried out so that the total similarity between the members of a single cluster is maximized and the total inter-cluster similarities are minimized. This allows the formed clusters to follow the native shape of the data rather than enforcing artificial shapes (e.g. circles in the case of k-means clustering). If we use this definition of spectral clustering, then each image should be labelled using a standard trained network, and a similarity matrix should be constructed based on the similarities between the raw labels regardless of the assigned class. The problem with this approach is that the resulting clusters will probably not follow the class-lines where images from a single class are divided across multiple clusters. Such clusters will be useless in grouping similar classes into coarser categories. To deal with this problem spectral clustering is applied on the class level and not on individual images. Rather than constructing a large similarity matrix between individual images, we use the confusion matrix as a similarity matrix between classes. This practical approximation has its own disadvantage, where subtle information in the raw labels are lost. For example if the confusion matrix is used, a raw image label like (0.45, 0.55, 0) is quantized to (0,1,0), and if such label is correct, then the confusion matrix will extract nothing from the quantized label about the similarities between these three classes, while the raw label clearly shows more similarity between the first and second classes. Unfortunately, it is difficult to utilize such information that exist in the raw labels without breaking the class boundaries.

In the case of the HD-CNN [69], the 1000 classes of the ImageNet dataset are divided into a number of coarse-categories, where the number of coarse-categories is a tuned hyper-parameter. Therefore, the hierarchy is a two level hierarchy, and spectral clustering is applied only once on the 1000×1000 confusion matrix. If the goal is to produce a hierarchy with more than two levels then spectral clustering needs to be applied multiple times (at each internal node of the hierarchy tree). In the implementation by Griffin et al [71], a multi-level hierarchical structure that have the shape of a binary tree was constructed for the Caltech-256 dataset. At each internal node, spectral clustering is applied to produce two clusters, and the process is continued until each cluster represents a single class. Their implementation use SVMs, and therefore their goal was to speed up computations by constructing a hierarchical SVM classifier

which will be faster than the 256 one-vs-all SVMs.

In our case, the hierarchical structure of the data is known where the dataset is divided into 10 coarse categories. Therefore, we are not relying on the confusion matrix to discover the hierarchical structure of the data. Our analysis of the confusion matrix is to examine if the confusion between the classes follows the hierarchical division between the coarse categories. To do that, we need to measure the percentage of confusion that occurs between classes that do not belong to the same category. To measure that, all the classes that belong to the same category are merged together to form a super-class. The merger process goes like this: if an image is misclassified as a class from the same category then this is considered a correct classification, while if an image is misclassified as a class from a different category then this is considered a wrong classification. The merger reduces the 100×100 confusion between the 100 classes into a 10×10 confusion matrix between the 10 coarse categories, where only Inter-Category confusion is considered (figure(5.4) draws the original 100×100 confusion matrix, and table(5.5) shows the resulting 10×10 confusion matrix). Table ((5.4) shows the error rate per category calculated using the diagonal of the resulting 10×10 confusion matrix, where the overall confusion (error rate) between categories is only 1.49%. Therefore, out of the 16.78% total misclassified images between the 100 classes in table (5.3) only 1.49% happen between categories, while the rest (16.78 - 1.49% = 15.29%) happens internally inside each category.

Super Class	Cars	Bugs	Cats	China	Birds	Fruits	Furniture	Lizards	Monkeys	Snakes	**Avg.**
Erroe Rate	0.26%	1.21%	0.94%	3.72%	0.23%	1.14%	1.84%	2.35%	0.43%	2.78%	**1.49%**

Table 5.4: results obtained by merging each category into a super-class. results show leakage between categories.

There is a discrepancy between the results in table (5.4) obtained using the confusion matrix, and the main results in table (5.3) obtained by comparing the performance of the main network trained on all categories to the performance of the 10 networks each trained using a single category. Table (5.4) shows that learning all categories on the same network will cause an Inter-category confusion equal to 1.49%, which might imply that learning all categories on the same network will cause an overall drop in performance equal to 1.49% compared to learning each category on a separate network. On the other hand, the factual measurements in table (5.3) shows that learning all categories on the same network improves the performance slightly with a marginal net gain equal to = 17.03% - 16.78% = 0.25%. If the confusion matrix shows that 1.49% of the images will suffer by learning all categories on the same network, then 1.49% + 0.25% = 1.74% of the images must have benefited from learning all categories on the same

99.74	0	0.1	0	0	0	0.16	0	0	0
0.2	98.79	0	0.44	0	0.12	0.17	0.28	0	0
0	0	99.06	0	0	0.04	0	0	0.9	0
0.15	0.21	0.14	96.28	0	0.5	2.3	0.3	0	0.12
0	0.1	0.05	0	99.77	0	0	0	0.08	0
0	0.74	0	0.4	0	98.86	0	0	0	0
0.12	0	0.32	1.3	0	0	98.16	0	0.1	0
0	0.58	0	0.14	0.21	0.15	0.17	97.65	0.2	0.9
0	0.12	0	0.09	0	0	0.14	0	99.57	0.08
0	0.6	0.34	0.11	0	0	0	1.5	0.23	97.22

Table 5.5: Confusion matrix between coarse-categories resulted from merging the main confusion matrix between classes. The matrix entries are normalized so that each rows sums up to 100.

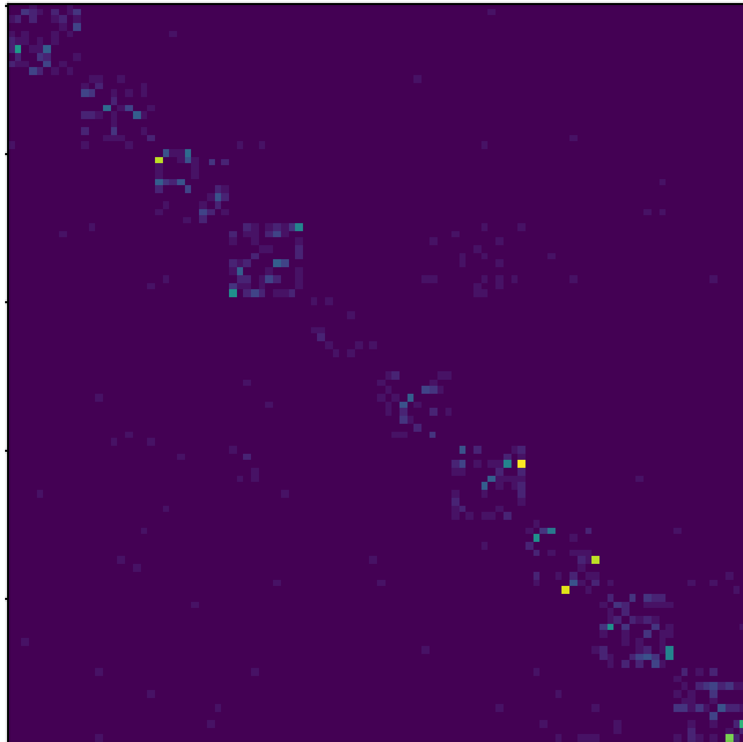

Figure 5.4: Confusion matrix of the main network trained on all coarse-categories. Most of the confusion is internal between the classes of the same category. The diagonal values or correct classifications are zeroed out to highlight the confusion between the classes.

network in order for the net gains to be 0.25%. This can be further verified by measuring the difference in performance for each of the 100 classes, and then plotting a histogram of these 100 differences as figure (5.5) shows. Figure (5.5) shows that roughly half of the individual classes slightly benefit from learning all categories on the same network, while the other half loses slightly.

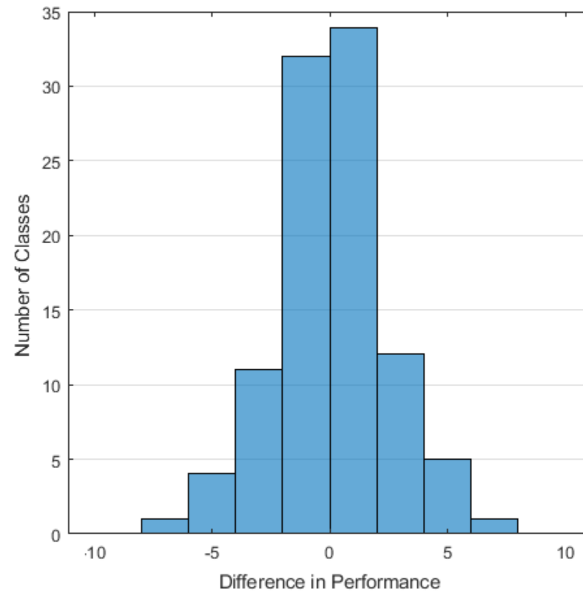

Figure 5.5: Histogram of the difference in performance per class, between using the shared network, and using a separate network per category. 100 values for 100 classes.

Comparing the results in table (5.4) obtained using the confusion matrix with the main results in table (5.3) helps to clarify the following points. First, although the performance of learning all categories on the same network matches the performance of learning each category on a separate network, some images (1.49%) will suffer, and some images (1.74%) will benefit from learning all categories on the same network. Second, relying only on the analysis of the confusion matrix is not sufficient to measure how well the shared network, trained on all categories, will do compared to learning each category separately on a dedicated network. This is because the confusion matrix can only measure the added confusion (1.49% Inter-category confusion) caused by increasing the number of classes, but it cannot measure how much the network will benefit from learning more classes. Only a direct comparison, like the main results in table (5.3), can measure how well the main network, trained on all categories, will perform compared to learning each category on a separate network.

5.5 Visualization

The results in the previous section showed the ability of a standard deep convolutional network to efficiently learn a hierarchical dataset that is made up of multiple coarse categories, where the overall performance of the network was equal to the average performance of learning each category on a separate network. The results showed that most of the confusion happens between classes that belong to the same coarse category, and only 1.49% out of the 16.78% total confusion happens between classes that belong to different categories. In this section we will try to investigate how the standard network was able to provide efficient separation between the different categories, and was able to learn all of them on the same network with no drop in performance.

To look inside the mechanics of the network and see what happens as the signals generated from the input images are propagated forward, a t-SNE visualization was carried out for the outputs of convolutional layers at different depths in the network. The visualization was carried out using the 34-layer residual convolutional network, and the 5000 test images of the hierarchical dataset used in the main experiment. For each test image, the high-dimensional output of a convolutional layer was smoothed out to a lower dimension (50-d) using PCA, and then projected into a 2D plane using t-SNE. To see how the network treats images from different coarse categories, each test image was coloured using the corresponding coarse category label. Figure (5.6) shows the t-SNE visualization for the outputs of convolutional layers 7, 15, 27, and 33, where the 10 used colours represent the 10 different coarse categories. Each dot represents the output of a single test image.

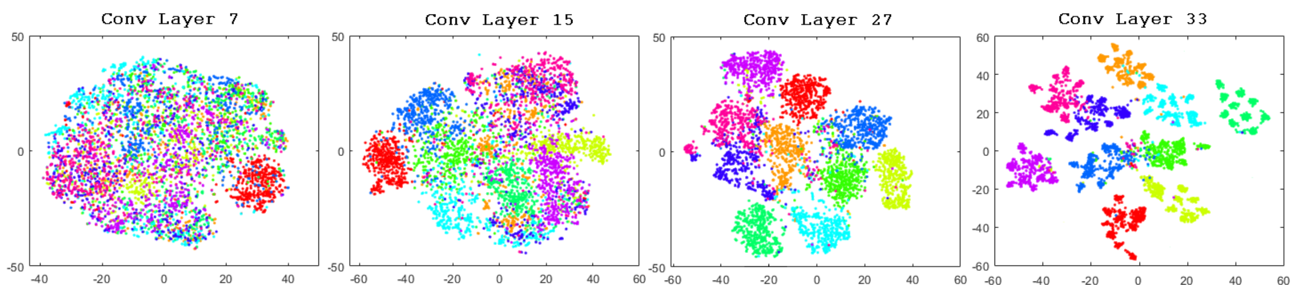

Figure 5.6: t-SNE visualization of the outputs of convolution layers 7, 15, 27, and 33. The 10 colors represent the 10 different categories. The projection shows that the network divides the classes into categories fist, and only in later stages it divides each category into multiple classes.

The main observation form the t-SNE projection in figure (5.6) is that the network starts to separate test images based on their coarse category identities first, and only in the latest stages it starts to separate images based on their fine-class identities. The projection of outputs of the

7th convolutional layer shows a clear formation of one coarse category (in red), and later on in layer 15 other clusters that represent other categories start to form. When the forward signal reaches layer 27 the outputs are clearly clustered based on their category identities. Up until layer 27, the projection shows that the feature detectors developed by the network discriminate between images based on their coarse category identities, and it does not show clear signs of discrimination based on the individual class identities of the images, even though training is done using only class labels. Only the projection of 33th convolutional layer shows the formation of small clusters that represent individual classes. The projection of the 33th convolutional layer output shows a two level clustering when images are first clustered into coarse categories, and then each coarse category is further divided into individual classes. The projection shows that for hierarchical data, the early and middle stages of the network develop feature detectors that respond to big variations, which mainly discriminate between coarse categories, while only in the latest stages the network develops feature detectors that respond to finer and more subtle variations that further divide these coarse categories into individual classes.

The t-SNE projection in figure (5.6) may provide some insight of why confusion mainly happens between the classes of the same coarse category, and it rarely happens between classes that belong to different coarse categories. Based on figure (5.6), the reason for this behaviour is because the network learns to separate between coarser categories (with big differences between them) at a much earlier stage in the network, and such discrimination becomes stronger as the signal travels forward toward the output. Many convolutional layers develop feature detectors that respond to big variations in the images which allows them to discriminate between coarse categories, and this discrimination becomes stronger in the later stages. This discrimination that responds to big variations which is well established throughout the network reduces the probability of confusing between the members of different coarse categories. Bilal et al [68] also concluded that the network discriminates between coarser categories at an earlier stages when experimenting with ImageNet. However, because the hierarchical structure of our dataset is well defined, our conclusions were reached using a different and more direct visualization.

The next section looks at ways to incorporate the hierarchical structure of our dataset to improve the performance of the standard convolutional network. Specifically, how to incorporate the coarse category labels into the implementation and training process of the network. Different implementations are tested, and the best performing one was inspired by the t-SNE projection in figure (5.6) which shows that discrimination based on the coarse category identity of the image happens at an earlier stage compared to that based on the class identity.

5.6 Incorporating the Coarse Category Labels

The dataset used in this experiment is the same dataset used the main experiment which is made up 100 classes that are divided into 10 coarse categories. The standard way of implementing hierarchical classification in shallow linear classifiers such as SVMs, is to train a classifier at each internal node of the hierarchical tree. In our case, the dataset has a two level hierarchy, where the root node divides the dataset into 10 coarse categories, and the second level nodes divide each of the coarse categories into 10 individual classes. If standard hierarchical classification is used, then a total of 11 CNNs are needed, where a root classifier determines the coarse category number of the image, and that number is used to select one of 10 second level classifiers to determine the class identity of the image. This will be wasteful (and impractical) for an expensive classifier such as a deep convolutional network. Even sharing most of the convolutional layers between all 11 classifiers (similar to the HD-CNN implementation [69]) is still expensive for a very deep convolutional network such as the 34-layer residual convolutional network used in this experiment. It will be more effective to use a standard network with many more layers. The straight forward method of incorporating the hierarchical structure of our dataset into the implementation of the network is by utilizing both the class label and the coarse category label of the training images. This can be done by adding an extra output layer, or a stack of a few layers, that predicts the coarse category label of the image, which is very cheap and more practical for an expensive classifier. The question is how to use the coarse category label? and in this section we test few different implementations.

5.6.1 Extend the output layer

The easiest way to incorporate the coarse category label is to extend the output layer to predict both the class label, and the category label of the input image. The class label is encoded as a vector of 100 numbers to represent 100 classes, and the coarse category label is encoded as a vector of 10 numbers to represent the 10 coarse categories. The output layer, which is implemented as a fully connected layer, will have 110 output neurons, where 100 neurons predict the class label, and 10 neurons predict the category label. Out of the 100 neurons that predict the class label one neuron should be ON to predict the class identity of the image, while all other 99 neurons should be OFF (equal to 0). Also, out of the 10 neurons that predict the coarse category label, one neuron should be ON to predict the category identity of the image, while all other 9 neurons should be OFF (equal to 0). Therefore, out of the total output 110 neurons, 2 neurons should be ON to predict both the class number and category number of the input image. In this section we will try different choices for the activation function of the combined output layer. In the first implementation a single softmax is applied to the entire 110 output

neurons, and the 2 ON neurons in the combined label are set to 0.5 so that the entire label add up to 1. In the second implementation, 2 softmax functions are used, where one is applied to the 100 output neurons that represent the class label, and another one is applied to the 10 output neurons that represent the category label. In the third implementation a log sigmoid activation function is applied after each neuron, which does not restrict the total summation of the output neurons to 1. For the second and third implementations, the 2 ON neurons in the combined label are set to 1. The top figure in figure (5.7) shows the label encoding for the first implementation that uses a single softmax, and the bottom figure shows the label encoding for the second and third implementations that use two softmax functions and log sigmoid functions respectively.

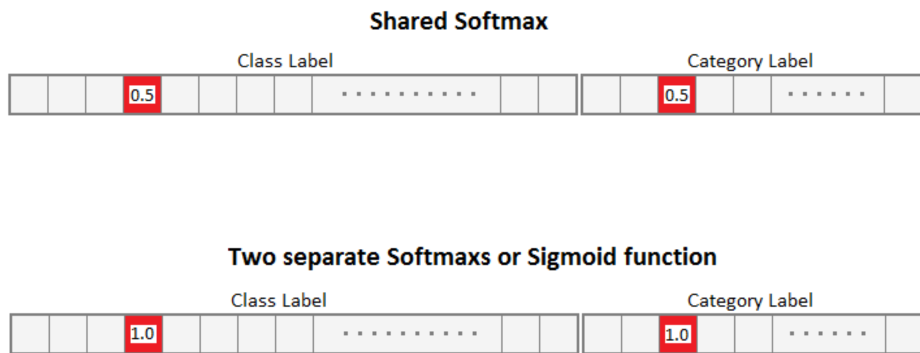

Figure 5.7: A combined class/category label coded using a shared softmax (top), and using two separate softmaxs or log sigmoid (bottom).

Table (5.6) shows the results for the three different implementations of the combined output layer in comparison to the standard network implementation that only predicts the class label. For all three implementations that incorporate the category label, the results show marginal improvements compared to the standard implementation. The results were obtained by averaging the results of 5 runs. Compared to the performance of the standard implementation, only the improvements of the third implementation that used the log sigmoid activation function are statistically significant (table(5.8)).

	Standard Class Label	Class plus Category Labels		
		Shared Softmax	Two Softmaxs	Logsig
Error Rate	16.78% ± 0.285	16.37% ± 0.32	16.42% ± 0.3	16.29% ± 0.37

Table 5.6: Three different coding schemes for the combined class/category label were tried. They all produced marginal gains compared to using only class labels.

5.6.2 Using the Category label based on the t-SNE visualization

Using the coarse category label of the image at the same output layer as the class label didn't improve the results substantially as table (5.6) showed. In this section we will try to use the hints from the t-SNE visualization in figure(5.6) in order to incorporate the coarse category label differently. Figure (5.6) showed that the network was able to figure out the coarse category identity of the image at an earlier stage compared to its class identity. Figure (5.6) showed that the network was able to clearly, though not perfectly, separate the 10 coarse categories into clusters at convolutional layer 27, while it was only able to separate individual classes into clusters at the last convolutional layer (layer 33). Therefore, it make sense to try to incorporate the category label at an earlier stage in the network compared to the standard class label (which is usually used at the last stage in the network). In this implementation, an extra auxiliary output layer that predicts the coarse category label is attached to the network at an earlier stage compared to the main output layer that predicts the class label. Multiple locations were tried to attach the extra output layer (after convolutional layer 7, 15, 27 etc.). The best result were obtained by attaching the extra output layer that predicts the coarse category label after convolutional layer 27. Figure (5.8) shows the implementation, where the extra output layer, which is preceded by a global average pooling, is attached after convolutional layer 27.

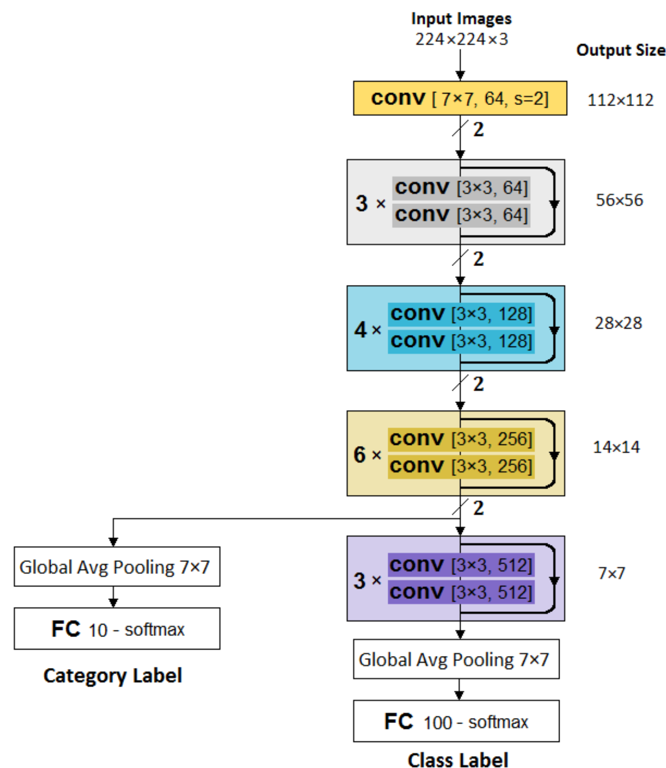

Figure 5.8: Category label is used after convolution layer 27. A global average pooling is applied to reduce the output channels to single values, and then the resulting 256 values are used as inputs for a fully connected layer that predicts category labels.

The reason for preceding the extra output layer by a global average pooling is to reduce the large output size of convolutional layer 27 from 49×256 to 256 to prevent overfitting in the extra fully connected output layer. Smaller pooling sizes were tried, but global average pooling produced slightly better results. In this implementation, the weights of convolutional layers 1 to 27 will be updated using the combined error signal that is propagated back from both the category label and class label. On the other hand, the weights of convolutional layers 28 to 33 will be updated using only the error signal propagated back from the class label. Table (5.7) shows the results of this implementation compared to the result of the standard implementation that only uses class labels. The results are the average of 5 runs and it shows that the error rate was reduced from 16.78% to 15.74%, and the difference is statistically significant (table(5.8)). It is clear from tables (5.6, 5.7) that using the coarse category label at an earlier stage produces better results than using it at the same layer as the class label.

	Standard Class Label	Class Label at Layer 34 Category Label at Layer 27
Error Rate	16.78% ± 0.285	15.74% ± 0.27

Table 5.7: In this implementation, the category label is separated from the class label and is used at an earlier stage. This utilization of the category label produced the best results.

	p−value in comparison to standard model sample size is 5
Shared Output Layer Shared Softmax	0.064
Shared Output Layer Two Softmaxs	0.076
Shared Output Layer LogSigmoid	0.031
Class Label at Layer 34 Category Label at Layer 27	0.0033

Table 5.8: p-values that shows the statistical significance of the difference in performance between the standard model trained on class labels only, and 4 different models trained using both the class and coarse-category labels. Only two models achieved statistically significant improvement for a level of significance $\alpha = 0.05$. It is clear that the improvements achieved by incorporating the coarse-category label earlier in the network achieved the most statistically significant improvements with the smallest p-value.

This implementation was inspired by the t-SNE visualization in figure (5.6) which showed that the network was able to clearly cluster the images into coarse categories well before the forward signals reached the output layer. The results in this section found that a good place to

insert the category labels is after convolutional layer number 27 (or near it), which concur with the t-SNE visualization in figure (5.6) which also shows a clear separation between the 10 coarse categories at the output of layer 27. Interestingly, if the t-SNE visualization for layer 27 is carried out after attaching the extra output layer to layer 27, the new visualization shows even cleaner and greater separation between the 10 coarse categories as figure (5.9) shows. The left figure shows the old t-SNE visualization for layer 27 obtained using the standard network, while the right figure shows the new visualization for layer 27 obtained after attaching the category labels to layer 27. Figure (5.9) and the results in table (5.7) show that using the category label at the right place allows the error signal propagated back from the category label to enhance (and not interfere with) the main error signal propagated back from the class labels. Here, both signals are telling the network to separate between the 10 coarse categories at convolutional layer 27, which resulted in a much cleaner separation.

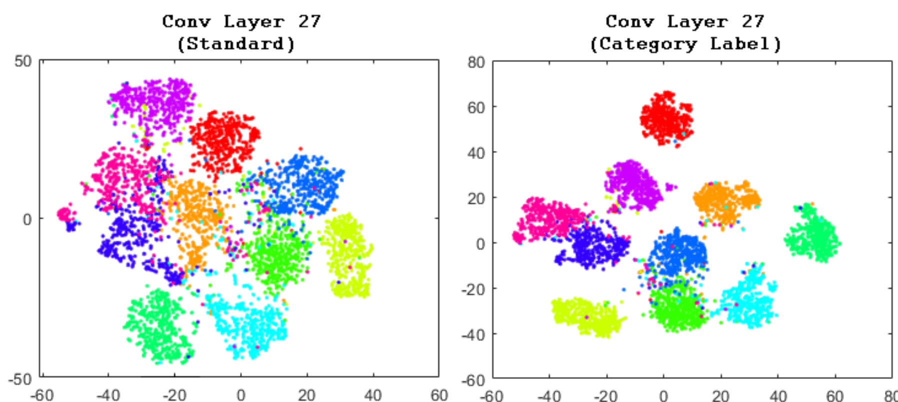

Figure 5.9: t-SNE visualization for the outputs of convolution layer 27. **left** for standard network, **right** for the network that uses the category label at convolution layer 27.

At the inference stage, the extra output layer that predicts the coarse category label is thrown away, and the standard network is used to predict the class label of the test images. Therefore, the inference time of the new model is the exact inference time required by the standard network.

5.7 Summary

- Deep convolution networks are very effective in solving big recognition problems with a large number of classes, such as the 1000-classes ImageNet. Our experiments show the error rate increasing at a much slower rate compared to the increase in the number of classes learned by the network. Very large datasets that have thousands of classes are by default dense datasets with well defined hierarchical structure. The main experiments

in this chapter show that CNNs do well with hierarchical datasets, where the network was efficiently able to restrict confusion between similar classes, and where increasing the number of classes 10 fold did not affect the performance of the network, as long as the added classes belong to a different coarse category. The efficiency in dealing with hierarchical data might explain, in addition to other reasons, why CNNs are very effective in solving large recognition problems. Another established reason is transfer learning.

- If a deep convolution network is trained on a big dataset that can be hierarchically broken down into coarse categories, then the difficulty in distinguishing between the classes of each of those categories will decide the performance of the network and not the size of the dataset. The performance of the network trained on the big dataset will be roughly equal to the average performance of learning each of the coarse categories separately. A similar conclusion was reached in [70] where they found that the density of the dataset (ratio of similar classes) is a more decisive factor in deciding the performance of the classifier than the size of the dataset.

- A t-SNE visualizations shows that early and middle layers in the convolutional network develop feature detectors that respond to big variations in the input images, which allows them to discover coarser patterns in the training data. On the other hand, the visualization shows that the latest stages in the network develop feature detectors that respond to small and subtle variations in the input images which allows them to discover finer patterns that represent individual classes. A similar conclusion was reached in [68] using a different method.

- If a hierarchical dataset can be broken down into multiple coarse categories where each input image can be labelled using both the class label and the coarse category label, then incorporating the coarse category label into the network implementation can improve the performance of the network. Our experiments show that a good way to use coarse category labels is to incorporate them at an earlier stage in the network than the standard class labels, and this implementation was inspired by the t-SNE visualization in figure (5.6).

CHAPTER 6

Multitasking Convolutional networks

6.1 Introduction

Multitasking is the ability to learn multiple tasks on a single machine learning model. Tasks can be learned sequentially, one after the other, or concurrently at the same time. If a deep neural network is used to learn multiple tasks sequentially, then the network suffers from catastrophic forgetting [28], where it forgets old tasks as it learns new ones. For this reason most multitasking neural networks learn tasks concurrently, where the model learns all tasks in a single learning session. Many multitasking neural network models follow the Caruana model [30] which was introduced in 1998. Caruana found that learning multiple auxiliary tasks (related to the main task) has helped the network to improve its performance on the main task. In his multitasking implementation each input data point (image, speech, words, etc.) has multiple labels, one for each task, and the combined error signal propagated back from all labels is used to update the network weights. Various implementations of Caruana's model have been adopted by others [31, 32, 33] where the differences reflect the specific application of the network, and how the structure is tuned and divided between shared layers and task specific layers.

The multitasking deep convolutional network presented in this chapter differs from Caruana's model, where each task has its own complete dataset. There are advantages and disadvantages for either approach which are discussed in the "related work" section in this chapter (the "related work" section also discusses other models in the literature [76, 77] that use a separate dataset per task). Our experiments show that with batch normalization, a deep convolution network was able to learn multiple homogeneous tasks with equal sizes by circulating through batches from those tasks in a round robin fashion. Multiple tasks were created by randomly sampling classes from ImageNet so that all tasks have the same number of classes. The multitask network was able to learn all tasks concurrently and was able to outperform the standard single-task networks by considerable margins.

The results show that the gains obtained from the multiutask network depend on the similarity between the tasks and the number of tasks learned by the network. The gains obtained from the multitask network increase if the tasks are similar, and decrease (become minimal) if the tasks are dissimilar. Also if the number of similar tasks learned by the network increases, then the gains obtained from the multitask network increase. These observations imply that the gains obtained from the multitask network are caused by the transfer of knowledge between similar tasks. At the inference stage the network uses a unique set of batch normalization statistics (means and variances) for each task. An examination of these normalization statistics show that they are similar in the early stages and more discriminative in the latest stages. If batch normalization was omitted, then the multitask network treats all tasks as a single big task, and struggles to separate between the individual tasks.

6.2 Related Work

The multitask convolutional network model presented in this chapter shares all the hidden layers between the tasks, and only the output layer is task specific. This is similar to the standard implementation of multitask neural networks used by Caruana [30], and others [31, 32, 33]. The important aspect in our implementation is in how data is presented to the network. The network circulates through batches of equal size from the tasks in a round robin fashion. The results show that without BN the network struggles to separate between the tasks and treats all of them as a single big task, and with BN the network succeeds in learning all tasks with a performance that surpasses that of single task networks.

The main difference between our implementation, and that of Caruana [30], is that in our implementation each task has its own dataset (multiple datasets are used to train the network), while in Caruana's model, there is only a single dataset where each input data point is annotated with multiple labels for multiple tasks. One advantage of using multiple datasets for multiple tasks is increasing the transfer of knowledge between the tasks. The results in this chapter show that the gains obtained from using the multitask network increase as the number of tasks learned by the network increases which points to an increased transfer learning between the tasks. With Caruana's model the transfer of knowledge between the tasks comes only from the labels as all tasks share the same input. On the other hand, the transfer of knowledge in our model comes from both the inputs and the outputs. However, Caruana's model is more suited when the dataset has extra information that is only available at the training stage, but is not available at the inference stage (when the model is utilized in practice). Such valuable extra input features can be used as auxiliary outputs (auxiliary tasks) to improve the performance of the network on the main task (e.g. the Pneumonia Prediction experiment presented in chapter 2 is one example).

The multitask cross-stitch network [75] tries to answer a different question: which layers should be shared, and which layers should be task specific, and how much sharing should exist in those layers? because each layer can be task specific, a whole baseline network is needed for each task. For each task, the outputs of layer l are a linear combination of the outputs of all the baseline networks at layer l. A set of adaptive parameters α_l control the amount of sharing at each layer l, and will be learned from the tasks. The separation between the tasks is established through the initialization of the α parameters, where for each task the α parameter of the corresponding baseline network is initialized to a higher value (e.g. 0.9) and the α parameters that correspond to the other baseline networks are initialized to a smaller value (e.g. 0.1). In our implementation the separation between the tasks is provided by the normalization statistics (means and variances) of BN, especially in the latest stages in the network as the results presented in this chapter show.

The relationship multitask network [76] tries to explicitly model the relationship between the tasks to avoid negative transfer of knowledge between them [99]. They impose tensor normal priors on the parameters of each task specific layer. Their results show improvements compared to other models on domain-adaptation problems (similar datasets with the same classes, where the input images belong to different domains). Task relatedness is not explicitly modeled in our implementation; however, the results show that the gains obtained from the multitask network reflect the kind of relationship that exist between the tasks. When tasks are uniformly sampled from ImageNet, the gains obtained from the multitask network are high. On the other hand, when each task represents a different coarse category (classes from different tasks do not share many common features) the gains are minimal. This implies that randomly and uniformly sampled tasks have a positive relationship between them where significant transfer learning occurs between the tasks, while distinct and dissimilar tasks that represent different coarse categories have a neutral (or even negative) relationship between them and little to no transfer learning happens between the tasks. This shows that the performance gains of the multitask network can act as an indicator of the kind of relationship that exists between the tasks. In fact, the relationship network [76] models three kinds of relationships using three separate covariance matrices. It models feature relationships and class relationships in addition to task relationships. Therefore, the gains obtained from their model are attributed to modelling all three relationships and not only task relationships. Like our model, relationship networks are trained on tasks, where each task has its own dataset.

All tasks in our multitasking model are image-classification tasks for natural images drawn from ImageNet, and all tasks have the same number of classes. Therefore, these tasks are homogeneous and belong to a single domain. Convolutional networks used for object detection [93, 13, 96] learn two heterogeneous tasks, (the object class label, and the coordinates of the bounding box), and both tasks are from the same domain (image recognition). Extending our model to learn multiple heterogeneous tasks from the same domain of image recognition is possible if those tasks require similar network structures. However, extending it to tasks that belong to different domains (images, speech, text) is likely to be difficult because the building blocks that work for one domain might not work for others. The MultiModel Mutitasking implementation [77] titled "One Model To Learn Them All" (discussed in chapter 2) shows the obstacles that need to be dealt with in order to implement a single model to learn multiple tasks from different domains. The inputs from the different domains need to be translated into a unified representation, and the shared multitasking model that will be trained using the shared representation should include building blocks from different domains (required to cover all the tasks). They included convolutional layers which are needed for image related tasks and an attention mechanism, and sparsely-gated layers which are needed for language related tasks. Their results show transfer of knowledge between tasks that belong to different domains.

6.3 Multitasking Implementation

This section explains how deep convolution networks that implement batch normalization can be used to learn multiple tasks in a round robin fashion on the same network. These results are obtained from training a single deep convolution network on multiple image datasets with the same size (the same number of classes), that are sampled randomly from the 1000-classes ImageNet. Therefore, these tasks are homogeneous with similar complexities, and belong to the same domain (image recognition). This does not imply that this setup is not applicable to other domains (e.g. speech, text). We expect this setup to work in other domains if the base model utilizes BN, and if it allows for the transfer of knowledge between similar datasets. Our model is similar to Caruana's model in that all hidden layers are shared among all tasks. However, data is presented differently in our model, where the network circulates through multiple tasks because each task has its own dataset, while in Caruana's model there is a single dataset with multiple outputs. Here, the words 'dataset', and 'task' will be used interchangeably. The details of the training phase and the inference phase of our multitask convolutional network are implemented as follows: -

- **Training: -** for each task or dataset, batches are created to only include images from that dataset. Then the network will circulate through all datasets in a round-robin fashion, by being trained on a batch from the first dataset, then on a batch from the second dataset, and so forth until it reaches the last dataset, and then circulates back through all of them again and again until training terminates. This round-robin circulation gives all datasets the same amount of exposure to the network, and does not allow the network to either focus on or forget any of the tasks.

- **Inference: -** for batch normalization, a fixed set of means and variances are calculated for each dataset using the training images of that dataset. When a test image from a certain dataset is passed through the network, the corresponding set of means and variances are used to classify that image. Therefore, if the network is trained to learn N different datasets simultaneously, there will be N sets of fixed means and variances that need to be calculated from the N training datasets and be used to carry out inference on the corresponding N test sets. Thus, these N sets of fixed means and variances act like a switch in the network implementation, where using the jth set of means and variances will switch the network to be able to classify images from the jth dataset.

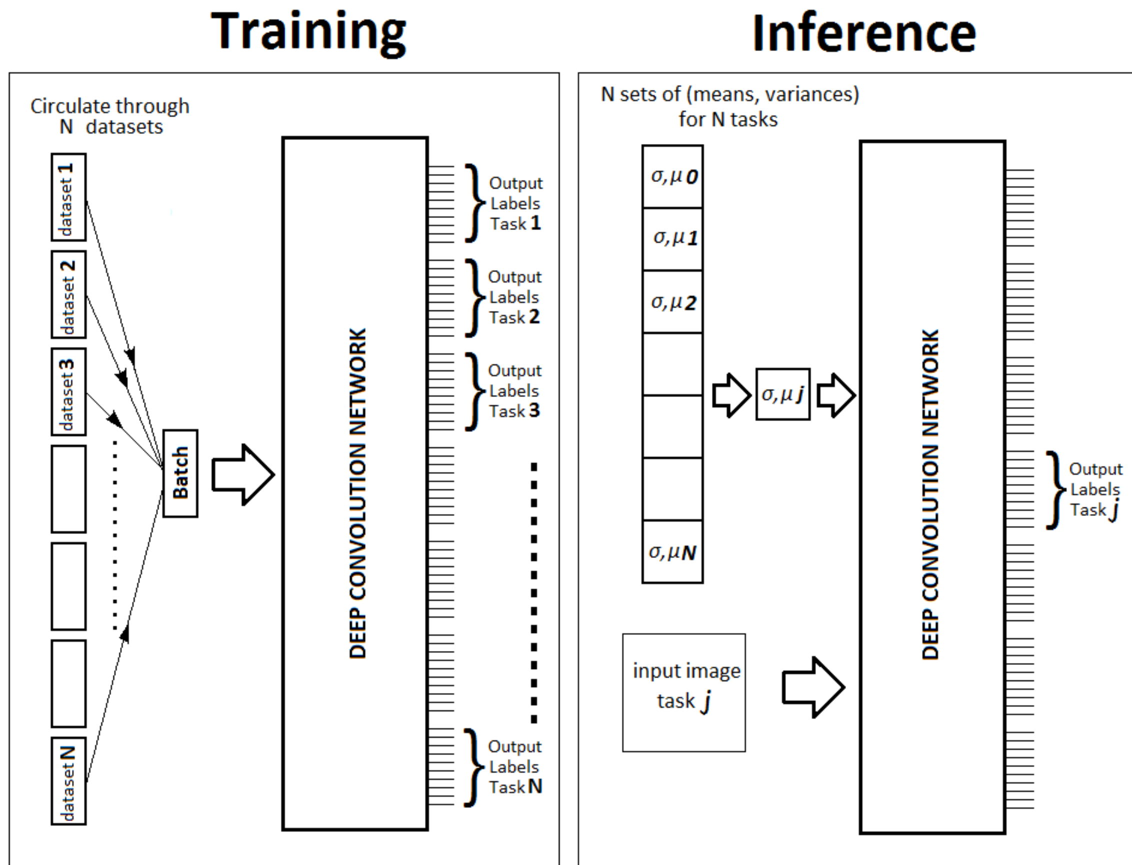

Figure 6.1: **left:-** training procedure network circulates through batches from N tasks in a round robin fashion, **right:-** inference procedure, j^{th} set of means and variances are used with test images from the corresponding j^{th} task.

Figure (6.1, left) shows how the training process is implemented where the network circulates through batches from N independent tasks, and figure (6.1, right) shows how the inference process is implemented where the jth set of means and variances are used to classify an image from the jth task. The labelling scheme is straight forward; if the total number of classes in all tasks is M, then the label predicted by the network is represented by M digits where one is ON (equal to 1), and all other are OFF (equal to 0) to reflect one of the M classes. If the number of classes per task is m, then the first m digits will represent classes from the first task, the second m digits will represent classes from the second task, and so on. Therefore, all hidden layers are shared between all tasks, and only the output layer is task specific. Also, there are no barriers that keep the network from misclassifying an image as a class from a different task, and therefore inter-task leakage is possible. What makes the shared weights of the network able to classify images from all tasks is coupling them with the correct set of fixed means and variances at the inference stage. The size of these N sets of means and variances is very small compared to the number of weights in the network, and therefore the total number of parameters needed in the multitask network is (almost) equal to the number of parameters in the standard single task network.

6.4 The Baseline Network

The residual convolutional network model [1] is chosen as baseline model to carry out experiments in this chapter (the justification of this choice is presented in chapter 3). In an attempt to save time, both the 18-layer and the 34-layer models from He et al [1] were tried initially, and in all those early trials the 34-layer model outperformed the 18-layer model, either significantly or slightly. Therefore all experiments were carried out using the 34-layer model. However, we found that when the total number of classes (of all tasks) used to train the multitask network is high (500 to 1000 classes), then using a wider version of the 34-layer residual network in figure (3.1) produced better results. In each convolutional layer, the number of the output channels in this wider version is 1.5 times the number of output channels used in the standard implementation shown in figure (3.1). Table (6.1) shows the details for the wider network. For the standard single-task networks (where the number of classes did not exceed 100) using the wider network produced negligible improvements. Also for the multitask network where the total number of classes in all tasks is small (100 classes), the wider version was not necessary. Using the wider network increased the training time by a factor of 1.6, which for the standard (small) input size of 224×224 pixels is still attainable. This is the only noticeable disadvantage for the multitask network, which requires more training and inference time than the single task networks. However, that is alleviated by replacing many single task networks with a single multitask network that significantly outperforms the single task networks. All the experiments in this chapter will be carried out using the wider network in table (6.1), which will focus the discussion on the proposed multitask model rather than on the choices of the baseline network, and will marginally improve the performance of the single task networks and make the comparison as fair as possible.

6.5 Experiments: Multitasking with BN

In this section the multitask model shown in figure (6.1) which uses the baseline residual convolutional network in table (6.1) will be trained on 10 tasks. In order to assess the success of the multitask network, its performance will be compared to the performance of the single task networks. Each task is represented by a dataset that is randomly sampled from ImageNet (using a uniform distribution), where all tasks have the same number of classes. The experiment is repeated three times, with the number of classes per dataset (task) equal to 10, 50, and 100. The test set for each of these datasets is sampled from the ImageNet validation set. These experiments are also carried out without BN to show that the multitasking network will fail without it. For reproducing the results, table (B.5) shows the class names used in the first experiment, tables (B.7) and (B.8) show the class names for the second experiment, and tables

(B.9), (B.10), (B.11), (B.12), and (B.13) show the class names for the third experiment, where each part in a table shows the class names for a single dataset.

output size	34 Layers
112×112	*conv*, 7×7, stride 2 96
56×56	*max pool* 3×3, stride 2
	$\begin{bmatrix} conv, 3\times3, 96 \\ conv, 3\times3, 96 \end{bmatrix} \times 3$
28×28	*max pool* 3×3, stride 2
	$\begin{bmatrix} conv, 3\times3, 192 \\ conv, 3\times3, 192 \end{bmatrix} \times 4$
14×14	*max pool* 3×3, stride 2
	$\begin{bmatrix} conv, 3\times3, 384 \\ conv, 3\times3, 384 \end{bmatrix} \times 6$
7×7	*max pool* 3×3, stride 2
	$\begin{bmatrix} conv, 3\times3, 768 \\ conv, 3\times3, 768 \end{bmatrix} \times 3$
1×1	*global avg pool* 7×7 $X-$d *fc*, *softmax*

Table 6.1: Network Structure

6.5.1 Setting the Hyper-parameters

The standard single task networks with BN that are trained on 10, 50, and 100 classes, have the same structure as the networks used in chapter three which are also trained on datasets that are randomly sampled from ImageNet with 10, 50, and 100 classes. Therefore, the same network setup and hyper-parameters will be used here. The normalized learning rate is set to 0.001, the weight decay parameter is set to 0.0005, the decay parameter for the RMSprop optimization method is set to 0.999, and the batch size is set to 10, 50, and 100 images for the tasks with 10, 50, and 100 classes respectively. For the single task networks without BN we found that the normalized learning rate need to be reduced to 0.0004, while the other hyper-parameters worked fine. This is expected as including BN allows for higher learning rates.

For the multitask network, the value of the weight decay depends on the total number of classes (for all tasks) learned by the network. If the total number of classes is small (100 classes), then the weight decay is set to 0.0005 (the same as the standard single task networks), and if the total number of classes is high (500 or 1000 classes), then the weight decay is set to 0.0001. The batch size is equal to the number of classes per task. Trying other values did not affect the results. For the multitask network that is implemented without BN, the same normalized learning rate of 0.0004 that is used for single task networks worked well and tweaking it did not improve the performance of the network. For the multitask network that implements BN, we found that the choices of the learning rate have a significant impact on the performance of the network. In the first two experiments where the number of classes per task is equal to 10 or 50 classes, the normalized learning rate was reduced from 0.001 to 0.0005, and in the third

experiment where the number of classes per task is equal to 100 classes, the normalized learning rate was reduced further to 0.00025. Using higher learning rates caused the multitask network to get stuck in a local minima with much worse final results. For the multitask networks, the choices for the learning rates and batch sizes were validated by setting aside a small partition of the training datasets (100 images per class), and the choices for the weight decay followed the observations from previous experiments which showed that the weight decay should be decreased as the size of dataset(s) learned by the network increases.

For the networks that implement BN, the choice of the weight initialization method is not very important and either one of the well known methods [20, 21, 19] produces good results. For the networks that do not implement batch normalization, the performance is more sensitive to the choice of weight initialization. For these networks the weight initialization by He et al [19], which was designed to work with rectified units $max(x, 0)$, outperforms older methods. Finally, the aggressive data augmentation method (usually used with the inception model [23] and used in the previous chapters) and the colour augmentation method by Krizhevsky et al [9] are used here with both the single task and the multitask networks (with and without BN). Table(6.2) summarizes the hyper-parameters used for all three experiments.

Parameter	Tolerance Range	Used Value	Impact
Learning Rate without BN	(0.0002 - 0.0005)	0.0004	significant
Learning Rate with BN single task networks	(0.001 - 0.0016)	0.001	significant
LR with BN multi−task networks 10 or 50 classes per task	(0.0003-0.0007)	0.0005	very significant
LR with BN multi−task network 100 classes per task	(0.0001-0.0003)	0.00025	very significant
L2 reg param single task networks	(0.00005 - 0.005)	0.0005	mild
L2 reg param multi−task networks	(0.0001 - 0.0005)	0.0001	mild
Batch Size	(10-100)	equal to number of classes per task	significant
Weight Initialization	lecun [20], glorot [21], kaiming [19]	kaiming [19]	significant[1]
Optimizer	(RMSprop, Adam) + L2 reg	RMSprop + L2 reg	mild
RMSprop Decay Parameter	(0.99-0.9995)	0.999	mild
Learning Rate Decay	(Step, Schedule) Decay	Step Decay by 0.5 5 times	mild
Epochs	100 epochs		

Table 6.2: hyper-parameters for single task and multi-task networks trained with and without BN. All hyper-parameters are similar except the L2 weight decay parameter and learning learning LR. Multi-task networks need less L2 regularization because they are trained on more data. The choice of the learning rate is critical for the multi-task network with BN. Using kaiming initialization is important when training without BN.

[1]Using kaiming initialization is more important for the model without BN.

6.5.2 Results

Table (6.3) shows the results for the multitask networks and the average performance of the corresponding 10 single task networks. Table (6.3) shows the results for all three experiments with 10, 50, and 100 classes per task, where the experiments were repeated with and without BN. Tables (B.1, B.2, and B.3) in Appendix B show the detailed results per task, where the multitask network with BN outperforms the 10 single task networks in all 10 tasks. Table (6.3) clearly shows that the multitask network with BN significantly outperformed the single task networks, while it fails without BN to match the performance of the single task networks. Compared to the average performance of the 10 single task networks, the multitask network with BN was able to reduce the error rate by 32%, 28%, and 26% for the three experiments (with 10, 50, and 100 classes per task) respectively. The multitask network with BN didn't just match the performance of the corresponding 10 single task networks, but rather was able to outperform them significantly. On the other hand, the multitask network without BN failed to separate between the 10 tasks and it treated them as a single big task (as the results show).

		10 Classes/Task	50 Classes/Task	100 Classes/Task
without BN	Single Task	5.7%	8.92%	11.07%
	MultiTask	10.6% ±0.218	16.7% ±0.185	22.26% ±0.173
with BN	Single Task	4.81%	7.71%	10.24%
	MultiTask	**3.23%** ±0.19	**5.53%** ±0.223	**7.61%** ±0.164

Table 6.3: The error rates for all three experiments comparing the single-task and multitask performances with and without BN.

Figure (6.2) shows the detailed results per task, comparing the results of the multitask network (with BN) with the results of the 10 single task networks for all three experiments. The blue bars show the error rate per task for the multitask network, and the yellow bars show the error rate per task for the single task networks. Figure (6.2) clearly shows that the multitask network with BN outperforms the 10 single task networks in every task.

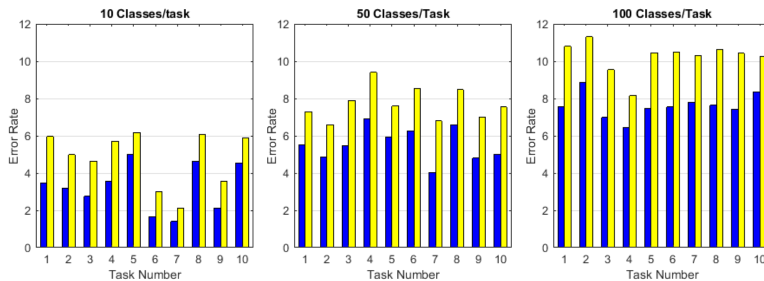

Figure 6.2: The figure shows detailed results per task. The blue bars show the error rate per task obtained for the multitask network, while the yellow bars show the error rates for single task networks.

The tasks (datasets) in the first experiment are small (10 classes per task), and therefore the reported results are the averages of these 5 runs. The tasks in the second and third experiments are much bigger (50 and 100 classes per task) and therefore the reported results are the average of 3 runs. Only the standard deviation for the multitask network is reported, because the variance for the overall performance of the 10 single task network will reflect the variance between the tasks and not the variance between the 3 or 5 runs. To measure the statistical significance of the improvements achieved using the multi-task network we will use a paired two-tailed t-test, where each task has a pair of results, one achieved using the single-task network, and one obtained using the multi-task network. The p-values in the first column in table(6.5) clearly shows that the improvements of the multi-task network in all three experiments are statistically significant with p-values that are much smaller than $\alpha = 0.05$. We also conducted a more strict test to see if the improvements for each individual task is statistically significant. In this case a standard t-test is performed for each individual task independently and the worst (highest) p-value across all tasks is reported. The second column in table(6.5) shows that not only the overall performance of the multi-task network (measured using paired t-test) is statistically significant but also the gains for each individual task is also statistically significant. This more strict test is a complementary test, it is not necessary to prove that the gains of the multi-task network are statistically significant, but it rather shows that the gains are spread across all tasks.

6.6 Training the Multitask Network on more Tasks

In these experiments the multitask network with BN will be trained using more tasks. The entire 1000 classes of the ImageNet dataset are evenly and randomly divided to create 10, 20, 30 and 50 tasks with 100, 50, 33, and 20 classes per task respectively. Therefore,the experiment will be repeated 4 times where the multitask network will be trained on 10, 20, 30 and 50 tasks, and its performance will be compared to the performance of the corresponding 10, 20, 30, and 50 single task networks respectively. We have already established in the first experiment that the multitask network will not work without BN and therefore this experiment will only be carried out with BN. The goal of this experiment is to see how the performance of the multitask network will be affected by increasing the number of tasks. The hyperparamters in this experiment are similar to those used in the previous experiment. One noticeable difference is for the multitask network trained on 50 tasks, where training has failed to converge when starting from random weights. To go around this problem, we started the training process with 30 tasks and then quickly added the remaining 20 tasks in the first 3 epochs.

	10 Tasks 100 Classes/Task	20 Tasks 50 Classes/Task	30 Tasks 33 Classes/Task	50 Tasks 20 Classes/Task
Single Task	10.24%	8.34%	7.13%	5.94%
Multi-Task	7.61% ±0.164	5.21% ±0.157	4.1% ±0.204	3.06% ±0.178

Table 6.4: The 1000-classes of ImageNet are divided into 10, 20, 30 and 50 tasks. Then the average performance of single task networks is compared to the performance of a single multitask network.

	p−value overall gains using paired t−test sample size is = no. of tasks	worst (highest) p−value per task sample size is = 3 or 5
10 tasks 10 classes per task	4.48×10^{-6}	0.0091
10 tasks 50 classes per task	2.64×10^{-8}	0.0137
10 tasks 100 classes per task	5.14×10^{-8}	0.0105
20 tasks 50 classes per task	2.97×10^{-15}	0.0087
30 tasks 33 classes per task	6.58×10^{-21}	0.0098
50 tasks 20 classes per task	3.48×10^{-25}	0.0238

Table 6.5: The first column shows the p-values obtained using a paired-t-test that measures the statistical significance of the overall performance of the multi-task network in comparison to the overall performance of the single-task networks, for 6 different experiments. The second column shows the worst (highest) p-value across all tasks, and it shows that the gains are significant for each task. All results are statistically significant for a level of significance $\alpha = 0.05$.

Table(6.4) compares the performance of the multitask network with the average performance of the single task networks in all 4 experiments (the results for the first experiment with 10 tasks are borrowed from the previous section). Tables (B.3) and (B.4) in Appendix A show the detailed results per task. The overall gains of the multi-task networks measured using a paired t-test are statistically significant for all 4 experiments as the first column of table(6.5) shows. Also, the gains for each individual task are statistically significant, as the second column of table(6.5) shows. Interestingly the gains of the multitask network increase as the number of tasks learned by the network increases. Compared to the average performance of the single task networks, the multitask networks trained on 10, 20, 30 and 50 tasks have reduced the error rate by 26%, 37.5%, 42.5%, and 48.4% respectively. These results clearly show that increasing the number of tasks causes the multitask network to perform better, which points to added transfer learning. These results imply that the gains obtained from the multitask network are attributed to the transfer of knowledge between the different tasks, and such transfer increases as the number of tasks learned by the network increases. Although the trend is clear, one caveat that needs to be taken into consideration is that the make up and the size of the tasks in these four experiments is different, which also can be a source of variation in the results. The

next experiment eliminates unwanted sources of variation and demonstrates that increasing the number of tasks improves the performance of the multitask network, and thereby provides clearer evidence for the hypothesis of transfer learning.

6.6.1 Testing the hypothesis of Transfer Learning

This experiment tries to provide more evidence that increasing the number of tasks improves the performance of the multitask network. Five tasks (each made of 50 classes) are used in four different experiments. In the first experiment each of the 5 tasks was learned separately using a single task network. In the second experiment, a multitask network was used to learn all 5 tasks. In the third experiment another multitask network was used to learn 10 tasks that include amongst them the 5 tasks used in the first and second experiment. In the forth experiemnt another multitask network was used to learn 20 tasks that include amongst them the 5 tasks used in the first, second, and third experiment. Therefore, for each of the main 5 tasks that are included in all 4 experiments, the difference in performance can only be attributed to increasing the number of tasks learned by the network from 1 to 5 to 10 to 20 tasks. Finally, all tasks in all 4 experiments have the same number of classes (50 classes).

	Single Task	MuliTask (5 Tasks)	MuliTask (10 Tasks)	MuliTask (20 Tasks)
Task 1	7.3%	6.44%	5.5%	4.95%
Task 2	6.6%	6.04%	4.86%	4.56%
Task 3	7.9%	6.68%	5.46%	4.72%
Task 4	9.4%	7.92%	6.9%	5.81%
Task 5	7.6%	6.54%	5.94%	5.27%
Average	**7.76%**	**6.72%** ±0.241	**5.73%** ±0.257	**5.06%** ±0.21

Table 6.6: Error rates for 5 tasks obtained using multitask networks trained on a variable number of tasks. The results show more gains by increasing the number of tasks learned by the multitask network, which point toward stronger transfer learning.

Table (6.6) shows the results of the 4 experiments for each of the 5 main tasks. For each of the 5 tasks, table (6.6) clearly shows that the performance of the multitask network improves as the number of tasks learned by the network increases. Compared to the average performance of the single networks, the average error rate for the 5 tasks was decreased by 13.4%, 26.3%, and 34.8% as the number of tasks learned by the multitask network increases from 5 to 10 to 20 tasks. For the 5 main tasks, the overall gains of the 10-task network are significantly better (p-value = 0.0083) than the gains of the 5-task network, and the overall gains of the 20-task

network are significantly better (p-value = 0.0266) than the gains of the 10-task network. For tasks that are uniformly sampled from ImageNet, this experiment demonstrates that increasing the number of tasks improves the performance of the multitask network. This experiment also provides strong evidence that the gains of the multitask network are caused by the transfer of knowledge (transfer learning) between the tasks learned by the network.

6.6.2 Another experiment on the Hypothesis of Transfer learning

All the tasks that have been used so far in this chapter are made of datasets that were constructed by randomly selecting a number of classes from ImageNet using a uniform random distribution. ImageNet is a dense dataset with many similar classes, and when the classes of ImageNet are uniformly divided among tasks, there is a good chance that those similar classes will also be uniformly distributed among the different tasks. Therefore, a proportion of the classes in each task will (probably) be similar to classes from other tasks which in turn facilitates the transfer of knowledge among these tasks. This is validated by the results in the previous section which clearly show that increasing the number of tasks improves the performance of the multitask network. In this section we will try to provide further evidence for this theory of transfer learning using different means, by training the multitask network using tasks that are clearly different from each other (tasks that do not share many common features). If the gains of the multitask network are truly caused by transfer learning (transfer of knowledge between similar tasks), then learning dissimilar tasks that do not share many common features should cause the amount of knowledge transfer to drop, and therefore cause the gains obtained from the multitask network to drop also. The dataset used in this experiment is the hierarchical dataset used in chapter 5, which is made of 100 classes that are naturally divided into 10 coarse categories. The multitask network is trained using 10 tasks, where each task represents a coarse category (each task is constructed using the 10 classes of a single coarse category). We found that the same hyper-parameters that were used in the previous sections for uniformly constructed datasets work well for the hierarchical dataset used in this experiment. Again, to assess the performance of the multitask network, its performance was compared to the performance of the 10 single task networks.

Table (6.7) compares the results of the multitask network to the results of the single task networks for each of the 10 tasks. Compared to the average performance of the 10 single task networks, the overall gains of the multitask network were minimal, where the average error rate for all tasks was only reduced by 2.5% (from 17.03% to 16.6%). Compare that to the multitask network that was trained on 10 uniformly constructed tasks with the same size where the error rate was reduced by 32% (table(6.3)). This shows that when the degree of similarity between the tasks decreases the transfer of knowledge between the tasks also decreases, and as a result

the gains from the multitask network decrease (and vice versa). This again provides strong evidence that the gains obtained from the multitask network are attributed to the transfer of knowledge between tasks. The gains from the multitask network can be considered an indicator of the kind of relationship that exists between the tasks (degree of similarity or dissimilarity between tasks).

	Single Task	MuliTsak
Task 1	16.6%	16.7%
Task 2	12.6%	11.8%
Task 3	18.1%	17.8%
Task 4	22.9%	22.1%
Task 5	2.43%	2.21%
Task 6	10.4%	10.15%
Task 7	16.4%	15.6%
Task 8	22.7%	21.8%
Task 9	23.4%	23.5%
Task 10	24.5%	23.9%
Average	**17.03%**	**16.6%** ± 0.23

Table 6.7: Error rates for 10 tasks obtained using multitask networks trained on a variable number of tasks. The results show more gains by increasing the number of tasks which point toward stronger transfer learning.

6.7 Discussion

The experiments in this chapter showed that a deep convolutional network that implements BN can be trained on multiple tasks by circulating through them in a round robin fashion, and be able to significantly outperform the single task networks. If BN was omitted, then the network struggles to separate between the tasks and it treats all of them as a single big task. The results also show that the gains obtained from the mutitask network are caused by the transfer of knowledge between the tasks, and that the amount of knowledge transfer depends on the similarity between the tasks and on the number of tasks. In this section we will try to look into the role of BN and the role of transfer learning in the operation of the multitask network.

In the inference stage, the experiments show that the multitask network that implements BN can efficiently separate the tasks if it uses a unique set of fixed normalization statistics (means and variances) for each task. For each task the normalization statistics are calculated using the training images of that task. If the wrong set of means and variances are used in the inference stage for a certain task, then the multitask network will not be able to correctly classify the images of that task. Figure (6.3) shows the confusion matrix for a multitask network with BN trained on 10 tasks, with 50 classes per task (from the first experiment). The confusion matrix

was constructed by using the right set of means and variances for each task, and it shows perfect separation between the tasks (all confusion happens internally between the member classes of each task). It seems that this efficient separation between the tasks (implemented through the normalization statistics of BN) allows the network to benefit from the transfer of knowledge between the tasks without sacrificing any performance in terms of more confusion as a result of increasing the number of classes 10 times. Here we will try to take a closer look at these sets of normalization statistics that are used to separate between the tasks in the inference stage, and how it is possible for the network to separate between the tasks, and at the same time allow the transfer of knowledge to happen between these tasks.

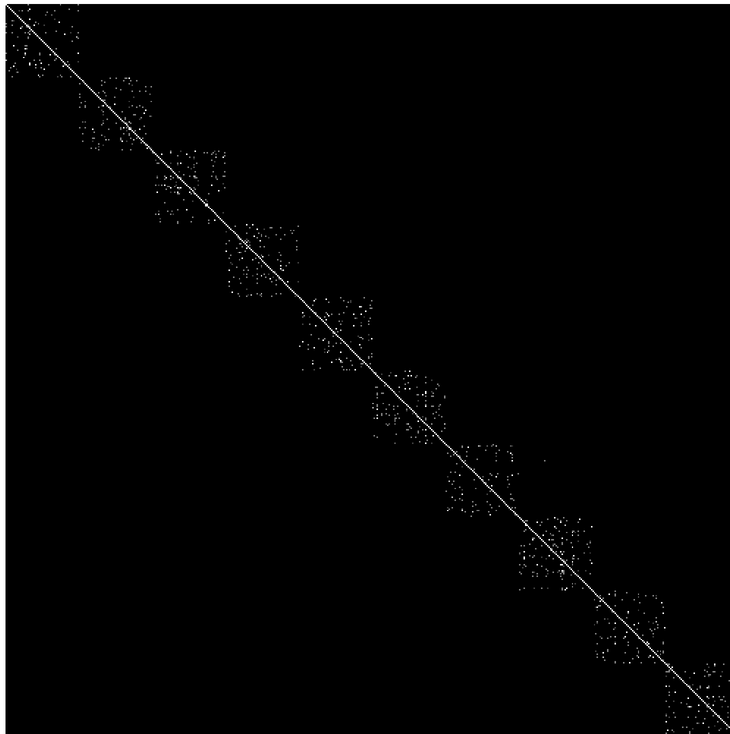

Figure 6.3: the confusion matrix for a multitask network with BN that is trained on 10 tasks that are uniformly constructed from ImageNet, with 50 classes per task. The confusion matrix shows a perfect separation between the tasks.

For a network trained on 10 tasks, there will be 10 different sets of means and variances that will be used in the inference stage. For each set (each task), the number of mean-variance pairs at each convolutional layer is equal to the number of output channels (between 96 and 768 in our implementation). The total number of mean-variance pairs per task is too high to be investigated individually (put into into tables), and therefore we will try to draw these 10 sets of means and variances in a meaningful way that might tell us how the network is utilizing these statistics. For each convolutional layer we will put the first 64 mean values for each of the

10 tasks in a 64×10 matrix, transfer these values into the grey region (between 0 and 255) and then draw this matrix as a grey image. The same is done for the variance values. The reason for drawing the means and variances separately, is to provide a clear contrast between the mean values used for each of the 10 tasks, and a clear contrast between the variance values used for each of the 10 tasks. The reason for only drawing the first 64 normalization values for each convolutional layer allows us to draw these values for multiple layers in the same figure, which in turn allows us to visually compare the values used for different layers. Figure (6.4) draws the mean values, and figure (6.5) draws the variance values for 14 convolutional layers spread throughout the network. For each of the 14 rectangular images (each of the 14 layers), a single row shows the mean (or variance) values for all 10 tasks (for one output channel in that layer). Therefore, a row that shows a single colour (horizontal line) implies that all 10 tasks are using similar mean (or variance) values at that output channel. On the other hand if a single row shows multiple different grey colours, then different means and variances are used for different tasks. Therefore, if the 64×10 grey picture that represents the mean (or variance) values for a certain layer is made of horizontal lines, then that implies that all 10 tasks are using similar normalization statistics at that layer. Figures (6.4, and 6.5) clearly show that early and middle layers are using similar means and variances for all tasks, while later layers are using different means and variances for different tasks.

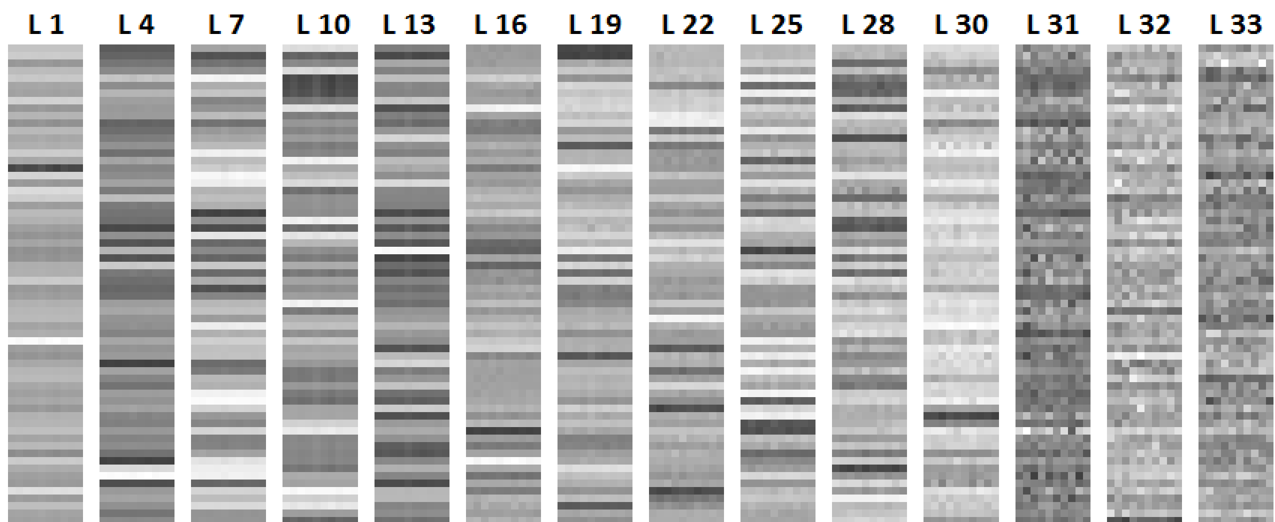

Figure 6.4: each rectangle shows the **mean** values per layer, where each row represents an output channel, and each column represents one of the 10 tasks. Horizontal lines show similar mean values for all tasks. Earlier layers show similar values while later layers show more discrepancy between tasks.

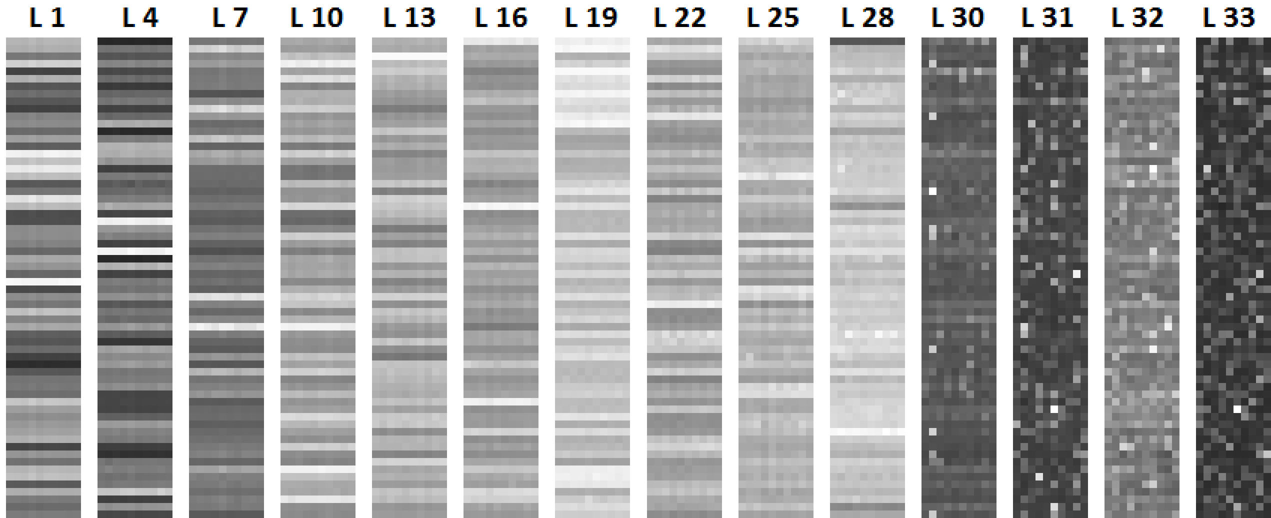

Figure 6.5: each rectangle shows the **variance** values per layer, where each row represents an output channel, and each column represents one of the 10 tasks. Horizontal lines show similar variance values for all tasks. Earlier layers show similar values while later layers show more discrepancy between tasks.

It is well known [99] that early stages in convolutional networks develop general (low level) feature extractors that are class agnostic (work across many tasks), while later stages develop feature extractors that are more class specific (therefore task specific). The transfer of knowledge between the different classes is usually stronger in those early layers that tend to develop class agnostic feature detectors. Interestingly, this is reflected in figures (6.4) and (6.5) which show that all tasks are using similar normalization statistics (means and variances) in the early layers, and that these statistics become different and more task specific in the later layers. The job of these 10 sets of means and variances is to separate between the 10 tasks (eliminate inter-task confusion). The figures show that this separation is less strict in the early stages and more enforced in the later stages. The fact that the network is not separating between the different tasks in the early stages (by using similar sets of means and variances for all tasks) can be thought of as an indicator that the network is developing common feature detectors that work across all tasks in those early layers. This allows for the transfer on knowledge in the early stages, and at the same time allows the network to separate between the tasks in the later stages. Therefore, the multitask network is relying on the transfer of knowledge between the tasks in the early and middle stages of the network in order to develop better feature detectors that can outperform the single task networks, while relying on normalization statistics of BN (by using a different set of means and variances for each task) to differentiate between tasks in the later stages of the network.

CHAPTER 7

Final Conclusions

7.1 Introduction

In this research we investigated different aspects of deep convolutional networks which today dominate the field of image recognition. This final chapter summarizes the main ideas and experiments, and the final results and conclusions. In regard to each of the main experiments we also discuss the main obstacles and challenges that we have faced, and the limitations that might require further investigation. The first (and main) section summarizes the main contributions of this research which covers the main conclusions of the main chapters. The section is divided into four subsections to cover the material in the four main chapters which cover all the experimental bodywork of this study. Each of these four subsections tries to clarify the contributions presented in each of the four main chapters by briefly summarizing the main results and comparing them to similar work in the literature. Also each subsection discusses the context and scope of our conclusions and the weaknesses and limitations of our implementations, and tries to suggest further investigation to remedy these limitations. The second section briefly summaries a few less successful experiments and discusses the reason behind their failure, and tries to draw some useful conclusions.

7.2 Summary of contributions

7.2.1 Utilizing batch structure through BN

Deep CNNs are trained using batches of images which are usually constructed randomly from the training set. The performance of the network is usually affected by the size of the training batches, but not by the structure of these batches. Our results in chapter three indicate that when BN is added to the network structure, the performance of the network is affected by the structure of non-random training batches. The results show that the network performs much better when inference was carried out using the same non-random batch structure that is used in the training phase (compared to carrying out inference on individual image or random batches). This implies that the network is incorporating the structure of the test batches when classifying the test images. The non-random batch structure that we used in chapter three is balanced batches. The results show significant improvement when both training and testing were carried out using balanced batches. Further visualization shows that the network has encoded the batch structure using the shared normalization statistics of BN, and further analysis indicates that the network has constructed an extra logic based on the structure of balanced batches. Similar behaviour was noticed by the authors of BN [7], but they gave a negative example where the performance of a batch normalized network suffers by training the network using non-random batches. Therefore, and to the best of our knowledge, we are the first to report that a batch

normalized network can use the structure of non-random batches to improve the performance of the network (specifically for balanced batches).

Our results show that utilizing the structure of balanced test batches improves the performance of the network by up to 80% for datasets with small number of classes, and up to 33% for datasets with a large number of classes. However, it is difficult to translate these big conditional gains achieved by training and testing the network using balanced batches into unconditional gains that can be achieved in practice. This is because structuring the test batches as balanced batches requires the labels of the test images. Our results also conclude that using the labels generated using standard inference to structure the test batches does not work. In general, the main challenge in utilizing the batch structure in improving the performance of the network is finding a way to structure the test batches without requiring the test image labels. Both batch structures used in our experiements (balanced batches), and used by Ioffe et al [7] require the image labels, and therefore it is difficult to use them to structure test batches. Therefore, the main challenge, that is still unresolved, is to find a method to structure the batches in an unsupervised way, that only depends on the nature of the input images, and not on the image labels. The subject of utilizing the batch structure to improve the performance of the network is not thoroughly investigated, and based on the big conditional gains achieved in our results, we believe that it is worth further investigation to translate such big gains into unconditional gains.

7.2.2 Training the network using a variable input size

The proposed implementation exploits the weight sharing property in convolutional layers where the size of the weight filters in such layers is independent of the size of the input channels. This allows convolutional layers to be trained using variable image sizes with minimal changes to the standard implementation. In order to train the whole convolutional network using a variable input size, we used a variable average pooling after the last convolutional layer to fix the number of parameters in fully connected layers. Training the network using a variable square input size allows us to inject more scale augmentation in the training process, more than what it can be achieved by training the network using a fixed input size. Our experiments in chapter 4 show better multi-crop performance, and much better single-crop performance compared to a standard network implementation with a comparable training time. Our implemenation is similar to the SPP network implementation [67]. The SPP network uses a variable pyramid max-pooling to fix the number of parameters in the fully connected layers, while our implementation uses a simple variable average pooling which makes it easier to implement when such pooling needs to be applied many time per image (as in the task of object detection). Also, the SPP model was implemented using the Caffe toolbox which (as they have stated [67]) restricted the model flexibility because the toolbox does not explicitly support training the network using a variable

input size. On the other hand, our model was implemented using CUDA which gave us much more flexibility to implement and train the model while at the same time allowed us to use optimized CUDA libraries. There is one major difference between our findings and the findings reported in the SPP implementation [67]. Our results show that the single-crop performance gains are much bigger than the the multi-crop performance gains when training the network using a variable input size, while their results did not report such discrepancy.

To simplify the CUDA implementation, a unified input size was used for all images in a single batch. This in turn allows us to use standard and optimized CUDA libraries to implement the convolutional layers. The down side of using a unified input size for the whole batch is increasing the memory footprint of the model. This restricted the value of the maximum input size that we were able to use in our implementation to 480×480. One way to solve this problem is to increase the flexibility of the model further by using a different input size for each image in the batch. This allows for a bigger maximum input size and therefore bigger scale augmentation, while at the same time maintaining a lower average size for the whole batch and therefore a lower memory footprint. The reason for not going this far in our implementation, is because it is very time consuming to construct such complex CUDA implementation of the convolutional layer from scratch, and also because that will prevent us from using the latest optimized CUDA libraries for standard convolutional layers, and therefore significantly increase the training time of our model (which is already very time consuming). We still believe that we struck a good balance between model flexibility and complexity; however, this is an interesting area of further investigation to see if it possible to further increase the amount of scale augmentation without increasing the memory requirement of the model.

7.2.3 Deep CNNs can efficiently classify hierarchical data and large datasets

Deep CNNs are very effective in solving large recognition problems. Our experiments in chapter five measure how the error rate increases as the size of the dataset increases. The results roughly show that the error rate increases only two fold as the number of classes in the dataset increases ten fold. Large and dense datasets of natural images usually follow hierarchical patterns where the classes can be grouped into coarser categories as we go up the hierarchical tree. The main results in chapter five show how deep CNNs deal with hierarchical datasets, which might help clarify why deep CNNs are effective in classifying large (hierarchical) datasets of natural images. A hierarchical dataset made of ten coarse categories was constructed from ImageNet and then used to train a deep CNN. The network was able to achieve a recognition rate for each of these coarse categories similar to that achieved by learning each category separately (using a separate network). In fact, the overall recognition rate of the network trained on all

coarse categories marginally surpassed the overall recognition rate of the 10 networks trained on individual categories. A further analysis to the confusion matrix shows that the confusion rarely crosses the coarse category lines, and mostly happens between the member classes of a single coarse category. These findings indicate that standard convolutional networks are able to discover hierarchical structures in the training data by making the confusion patterns follow the hierarchical patterns in the data. More importantly, these findings indicate that these networks are very effective in classifying hierarchical data, where increasing the size of the dataset by adding more coarse categories does not decrease the recognition of the network.

A t-SNE visualization shows that the network is using the early and middle layers to break the hierarchical data into coarse categories, and then use the latest layers to break each coarse category into individual classes. It is interesting to see this two-step processing of our two-level hierarchical data by the network. Our hierarchical dataset has a shallow and balanced tree structure with only two levels, where the root node divides the data into coarse categories and then the leaf nodes divide each coarse category into individual classes. It is exponentially harder to construct a balanced hierarchical dataset with more levels, and it will very interesting to verify these results using deeper hierarchical structures. It will be very interesting to see if the multi-step classification process, where the number of steps is equal to the number of levels in the hierarchy, can be verified using a deeper hierarchy and see which convolutional layers are dedicated by the network to each level in the hierarchical tree.

Finally, we incorporated the hierarchical structure of our dataset into the network implementation by adding an extra output layer that predicts the coarse category label of the image. Based on the t-SNE visualizations this extra output layer was incorporated at an earlier stage compared to the main output layer which predicts the class labels. The results show that predicting the coarse category label at an earlier stage produces better results than predicting both the class and category labels at the same output layer. Our findings in chapter five are to some extent similar to those by Bilal et al [68] and to those by Deng et al [70]. The main differnece with Bilal et al [68], is that their findings were achieved using a generic dataset and indirect visualization, while our results were achieved using a hierarchical dataset and a more direct visualization. The main differences with Deng et al [70], is that their findings were achieved using shallow linear classifiers while our achieved using deep CNNs. However, reaching similar conclusions using different means further verify these conclusions.

7.2.4 Multitasking with deep CNNs

In chapter six we introduced a multitasking model that is capable of efficiently learning multiple homogeneous classification tasks on the same deep CNN. The model uses a deep residual convolutional network [1] that implements BN as a baseline network. The model was able

to learn multiple tasks simultaneously by circulating through them in a round robin fashion and its performance significantly surpassed the overall performance of the single task networks. The separation between the tasks is implemented through BN by using a separate set of normalization statistics per task at the inference stage. Our analysis shows that tasks are using similar statistics in the early and middle layers, and different normalization statistics in the latest layers. The results show that removing BN makes the network unable to separate between the tasks. The main difference with the mainstream implementation (Caruana's model), is that in our implementation each task has its own dataset (its own inputs and outputs), while in Caruana's model all tasks share the same inputs and each task has its own outputs. Our results indicate that the gains of the multitask network were caused by the transfer of knowledge between the tasks. In Caruana's model all tasks share the same inputs, and therefore the transfer of knowledge between the tasks comes only from the output labels, while in our model each task has its own inputs and outputs, and therefore the transfer of knowledge comes from both the input images and the output labels. Because the input image is significantly larger and has much more information than the output label, it is plausible that most of the knowledge transfer between the tasks comes from the input side. Therefore, and based on our results and analysis in chapter six, we believe that using a whole dataset for each task increases the transfer of knowledge between the tasks and as a result increases the gains of the multitask network.

Our model uses a deep CNN as a baseline network where all convolutional layers are shared across all tasks, and only the output layer is task specific. Our experiments were restricted to homogeneous image classification tasks with equal sizes. Therefore, the next logical step is to try to extend it to multiple heterogeneous tasks from the domain of image recognition. This has already been done using Caruana's model on a limited scale in the case of object detection [93] where the network predicts both the class identity and the bounding box coordinates for each object. It is not a trivial task to extend our implementation to accommodate multiple tasks that are fundamentally different in nature even if these tasks belong to the same domain. For example, image classification and image segmentation require different network structures. In the case of image classification the network is made of multiple consecutive convolutional layers, while in the case of image segmentation the network is made of multiple convolutional layers followed by multiple deconvolution layers. Circulating through such tasks in a round robin fashion means that task-specific components will only be updated when the corresponding task is passed through the the network.

The model in its current form can only be applied in the domain of image recognition because it is using a deep CNN as a baseline model. CNNs are designed specifically for image recognition. Applying this model to a different domain, requires (at least) replacing the baseline convolutional network with a different baseline network that is appropriate for the target domain. The new baseline network needs to include BN in its implementation for the model to work.

Applying the model to multiple tasks from multiple domains is much harder, because different domains have different input representations and they require different models with different core components. The implementation by [77] shows that a multi-domain multitask model requires translating the inputs from different domains into a unified representation and then designing a unified multitasking model that includes the important core components required for those domains.

7.3 Less successful experiments

This section briefly summarizes less successful experiments and the conclusions that can be drawn from their results. These experiments investigated the role of the convolutional layer and the role of the activation function, as well as the idea of curriculum learning.

7.3.1 Reducing the computation cost of the convolutional layer

A single convolutional layer may require tens of millions of 2D convolutions in a single forward pass. Therefore, it makes sense to try to make changes to the implementation of the convolutional layer in order to reduce computations. To achieve this objective we changed the design of the convolutional layer by using large filters with the same size as the input channels, and therefore significantly reducing the number of multiplications and additions. The new model was a midway between simple feed-forward networks and CNNs, and its performance was also better than that of feed-forward networks and wose than that of standard CNNs. The new implementation achieved significant speed up, but significantly worst performance than standard convolution. Later on in 2017, a new convolutional network model called the Xception network [26] used depthwise separable convolutions to significantly reduce the amount of computations in the convolution layer without sacrificing any performance loss. We believe that the reason why the Xception model has succeeded while our approach has failed is because the Xception model did not change the fundamentals of the standard convolutional layer. The main principle behind the Xception model is to approximate the standard convolution by separating the spatial convolution and the channel projection into two consecutive stages and therefore maintaining the strengths of standard convolution. On the other hand, the failure of our implementation is caused by changing the fundamentals of the standard convolutional layer, and therefore losing the strengths of the standard implementation. This unsuccessful trial further points to the importance of using small filters in the convolutional layer (compared to the size of the input channels) especially in the early layers, which in turn forces these filters to extract small, frequent, and position-invariant low level features.

7.4 Parametric Rectified Linear Functions

The reintroduction of the rectified linear function $max(x, 0)$ is one of the reasons behind the resurgence of deep CNNs. The rectified function $f(x) = max(x, 0)$ approximates nonlinearities by piecing together the two lines $y_1 = x$, and $y_2 = 0$. He et al. [19] achieved better results by generalizing the line equations and piecing together $y_1 = x$, and $y_2 = a\ x$, and called it the parametric rectified function. In our experiments we tried to generalize these line equations further using different variants of the general line equation $y = a\ x + c$. Our results showed significant improvements compared to the standard rectified function, and small (but not statistically significant) improvements compared to He et al. [19]. Interestingly, adding BN (which was introduced later) to the network implementation, completely erases any advantages of these more sophisticated functions over the original rectified linear function $max(x, 0)$. This is a repetitive trademark that we have noticed, where the improvements obtained using different methods are often not orthogonal. For example our early experiments showed that using simple data augmentation (random cropping) erases the advantages of using stochastic pooling [56] over max pooling, the results by Simonyan & Zisserman [10] show that using more convolutional layers eliminates the need for local contrast normalization layers, and the results by Ioffe et al [2] show that using BN reduces the benefits of dropout regularization [46]. This interdependency between components that were designed for completely different purposes is one of the fundamental difficulties when trying to improve the performance of deep CNNs by adding new components or changing the structure of old ones.

7.4.1 Curriculum learning with deep CNNs

The basic idea of curriculum learning is to start from easier curriculum (data) and gradually expose the network to harder curriculum (which is similar to the way humans learn). The experiment required the construction of a large dataset from scratch by downloading more than 150,000 images from the internet and then dividing those images into multiple groups based on how difficult they are to classify. Training the network on simpler images first should makes it easier for the network to develop the primary concepts that should help it to learn more challenging images in later stages. Although this idea has worked in the past for smaller and more controlled experiments [101, 58], we were unable to apply it successfully on this large and more realistic scale. There were two main obstacles in applying curriculum learning when training the network on real images. First, most of the images downloaded from the internet were nice and clean, and there was not enough challenging images even when using search phrases that were constructed specifically to retrieve more challenging images. Second, it is very challenging to use the human eye to construct an adequate curriculum by dividing the

data into groups based on image complexity, because what is difficult for humans might not be difficult for artificial neural networks. Among the factors that we used to divide the images is the size of the main object and the amount of clutter in the image, and even with this simple criterion there were many images that exist in the grey area where putting such images in a certain group is mainly subjective. We still believe that the logic behind curriculum learning is sound, but applying it requires more control over the experiment (e.g. constructing the images for the purpose of curriculum learning rather than using or downloading existing ones).

APPENDIX A

Technical details of the network implementation

A.1 Introduction

This appendix contains technical details related to the implementation of deep convolutional networks. Such details include the optimization methods used to train deep CNNs, practical implementations of the expensive convolutional layer, design principles for complex deep neural networks trained using back-propagation, and choosing the appropriate software tool to design fast and efficient deep CNNs. Finally, at the end of this appendix some results and conclusions about the usage of parametric rectified functions are presented.

The first section discusses different SGD optimization methods, and which ones are being extensively used with modern deep neural networks. Modern deep CNNs have many convolutional layers that use the rectified linear function as an activation function. For such networks reducing computations and memory footprint is paramount, and therefore simpler SGD that only utilize first order gradients are often used. The non-saturating rectified linear function simplifies the active mapping between the inputs and outputs, and for such activation functions, first order SGD optimizers are sufficient. The next section discusses practical implementations for the convolutional layer, which may require up to millions of 2D convolutions in each forward or backward pass. Two implementations are discussed. The first implementation transfers all 2D convolutions in a convolutional layer into a single matrix-matrix multiplication. This transformation does not reduce the actual number of computations but rather it minimizes the number of memory read/write operations which allows for maximum utilization of thousands of GPU cores. The second implementation [63] used Winograd's minimal filtering [64] to reduce the actual number of computations required in the convolutional layer. The next section discusses the implementation of deep neural networks trained using the back-propagation of first order derivatives. Such models are designed as acyclic graphs, and new components can be plugged in by propagating the input signal forward through those components and by using the chain-rule of derivatives to propagate the error signal backward through those components. This section also discusses the CUDA software platform which we have used to implement and run our models on fast NVIDIA GPUs. Finally, at the end of this appendix we included experiments about parametric rectified functions, and about the normalization statistics of BN. The results about parametric rectified functions can be used to understand the mechanics of standard rectified linear functions, and further justify the choice of $max(x, 0)$ (or $min(x, 0)$) as an activation function for deep feed-forward neural networks. The experiments about the normalization statistics of BN show a correlation and interaction between these statistics and the residual connections of deep residual networks [1].

A.2 Optimization methods and Loss function

As the error signal propagates back it will be multiplied by the derivative of the activation function at each layer, and for saturating activation functions this derivative will be zero or close to zero most of the time. This problem was handled for all hidden layers (convolution and fully connected) by using the rectified linear activation function $max(x, 0)$ which has a unity derivative value when its input is greater than 0. The rectified linear function can not be used for the output layer, because when the input of the function is less than 0 the output will be a hard 0, and therefore no error signal will be propagated back from that output neuron even if the difference between the ground truth and actual output is maximum. The most widely used activation function for the output layer is the softmax. However, with the mean square error MSE loss function the derivatives of the softmax can still have small values which in turn can slow convergence as the back propagated error signal is multiplied by those derivatives. For this reason, most implementations use the cross-entropy loss function instead of the mean square error loss function. When the cross-entropy loss is used the derivatives of the softmax get canceled out of the gradient decent equations and the backpropagated signal at the output layer will only be proportional to the difference between the actual output and the ground truth. Equation (A.1) shows the MSE loss function and the corresponding update equation for a weight $w_{i,k}$ at the output layer, while equation (A.2) shows the cross-entropy loss function and the corresponding update equation for a weight $w_{i,k}$ at the output layer. These equations are for a single example to simplify notation. Figure (A.1) shows the worst-case scenario for the MSE loss function, when the actual output is $y_i = 1$ and the ground truth is $t_i = 0$, which means the error is maximum, but because the activation function is saturated, the derivative of the activation function $\acute{\sigma}(z_i)$ will be zero. Therefore, if the MSE loss function is used, the weight's update $\frac{\partial loss}{\partial w_{i,k}}$ at the output layer can be zero, even if the error is maximum. The cross-entropy loss function will not have this problem because the derivative of the softmax activation function is not multiplied to the backpropagated signal. In conclusion, the rectified linear function eliminates the problem of small derivatives in the hidden layers, while the cross-entropy loss function eliminates this problem in the output layer.

$$MSE\ loss = \frac{1}{2} \sum_i (y_i - t_i)^2 \qquad \rightarrow \qquad \frac{\partial loss}{\partial wi,k} = (y_i - t_i)\ \acute{\sigma}(z_i)\ x_{i,k} \tag{A.1}$$

$$CE\ loss = -\sum_i t_i \ln y_i \qquad \rightarrow \qquad \frac{\partial loss}{\partial wi,k} = (y_i - t_i)\ x_{i,k} \tag{A.2}$$

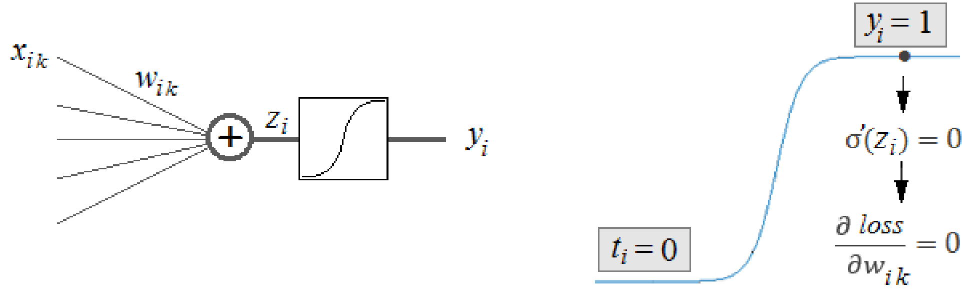

Figure A.1: the backpropagated MSE of a saturating output activation function is blocked because the derivative $\acute{\sigma}(z_i)$ equals to 0, even though the error $y_i{-}t_i = 1$ is maximum.

The networks studied in this this research use supervised training to optimize the network parameters. This supervised training is based on the simple gradient decent algorithm that uses first order derivatives to update the network weights. Before the era of deep neural networks, more sophisticated optimization techniques were used with shallow neural networks, such as conjugate gradients shown in equation (A.3), and second order derivatives that try to approximate the Hessian matrix such as the Levenberg-Marquardt method shown in equation (A.4).

$$\Delta w_k = \alpha\ p_k \qquad p_k = -g_k + \beta_k\ p_{k-1} \qquad \beta_k = \frac{\Delta g_{k-1}{}^T\ g_k}{\Delta g_{k-1}{}^T\ p_{k-1}} \tag{A.3}$$

$$\Delta w_k = -\alpha \left(J_k{}^T\ J_k + \mu_k\ I \right)^{-1} J_k{}^T\ e_k \tag{A.4}$$

Current neural networks use simpler optimization techniques that are based only on first order derivatives. The update equation that is used frequently is based on gradient decent with momentum and L2 weight decay as show in equation(A.5).

$$\Delta w_k = \gamma\ \Delta w_{k-1} - \alpha\,(g_k + \lambda\ w_k) \tag{A.5}$$

With simple gradient decent the amount of update to the network parameters is proportional to the gradient, which can be very small, causing slow convergence. Adding momentum can help, but if a long sequence of updates had small values then the momentum also becomes small. An optimization method called Resilient Propagation (Rprop), uses only the sign of the gradient, and ignores its magnitude. The updates start from a small predefined value, and if the gradient changes signs between two consecutive updates that value is reduced (multiplied by 0.5), and if the sign doesn't change that update value is increased (multiplied by 1.2). This method works well with small datasets when all data are included in each update. If the dataset is large, as is often the case with deep convolution networks, then the data is divided into batches, and Stochastic Rprop will give a poor approximation to Rprop, as the contribution of

each batch will be arbitrary and not proportional to its importance. RMSProp (Root Mean Square Propagation) is a stochastic gradient decent algorithm that works on minibatches, and tries to utilize the concepts introduced by Rprop. For each parameter, the update value (current derivative) is divided by the running average of the derivative for that parameter. This will reduce the impact of the magnitude of the current gradient on the update value, and also because the running averages change slowly, it will treat different batches fairly, and thus allow it to work on bigger datasets. The running average is calculated as a root mean square as shown in equations (A.6)(A.7).

$$v_t = \gamma \ v_{t-1} + (1-\gamma)\Delta {w_k}^2 \tag{A.6}$$

$$w_k = w_{k-1} - \frac{\alpha}{\sqrt{v_t}}\Delta w_k \tag{A.7}$$

So why do deep neural networks use simpler optimization methods to update their parameters compared to those used by older shallower networks? The first apparent reason is because more sophisticated methods require much more computation and can only be applied to smaller shallower networks. For example, conjugate gradient algorithms require an expensive line search to find an appropriate learning rate value α, and methods that approximate second order derivatives such as the Levenberg-Marquardt have storage and computation requirements that can be as high as the square of the number of parameters in the networks. But not all deep neural networks are big neural networks, and current available computation resources can overcome this computation hurdle. The more scientific reason is because most of the current deep neural networks use the simple activation function $max(x, 0)$, and this function only decides which neurons are on or off in the current update cycle. Once the on and off neurons have been decided, the on neurons form a simple linear relationship between the inputs and outputs, and such linear relationship doesn't require complex optimization methods. Even the softmax activation function at the output layer can be neutralized by using the cross-entropy loss function as descried in a previous section. Figure (A.2) shows a simple network with a single hidden layer that has two neurons where only one neuron is on in each cycle. Each case will map a different linear function to approximate the relationship for that specific input. Equations (A.8) and (A.9) model the relationship for the cases shown in figure (A.2), which are purely linear relationships if we ignore the softmax function at the output layer.

$$y_1 = \sigma(w_3 w_1 x_1 + w_3 w_2 x_2) \tag{A.8}$$

$$y_2 = \sigma(w_6 w_4 x_1 + w_6 w_5 x_2) \tag{A.9}$$

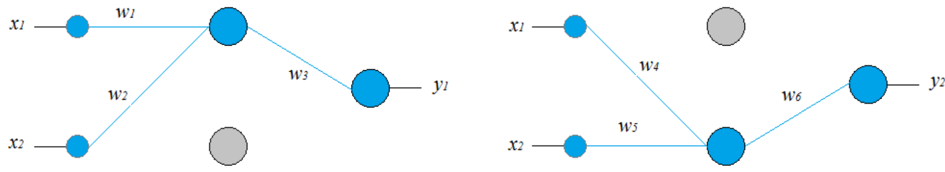

Figure A.2: shows how the ReLUs decide which linear system maps the inputs to the outputs.

A.3 Practical Convolution Implementation

The number of convolutions required in each convolution layer equals the number of input channels multiplied by the number of output channels in that layer. As figure (2.4) shows for the VGG and Residual models the last set of convolution layers have 512 input channels and 512 output channels, therefore the total number of convolutions in those layers is $512 \times 512 = 262144$ entire operations. Training is mostly done on batches of images, and if the batch size is 100 images then the total number of convolutions for the forward pass at each of these layers is $512 \times 512 \times 100 = 26214400$. That is more than 26 million entire convolutions needed in each forward pass for just one convolution layer, and the number of mathematical operations is almost doubled in the backward pass to update the filter's weights and to propagate back the error signal.

The number of mathematical operations needed to update the weights of a deep convolution network in each pass is astronomical, and using a single or a couple of CPU processors is not enough, and will cause training to become computationally infeasible. Adequate options include using a cluster of computers, a supercomputer, or GPUs. Using GPUs is the chosen method in this research and it is widely used with deep learning. NVIDIA GPUs provide a good software platform called Compute Unified Device Architecture or CUDA, that allows engineers to write general purpose software that can run on NVIDIA GPUs. Implementing the convolution operation in the classical way of sliding the weight filters on 2D input channels might be appropriate in convolving a single image, or a few images, but it is not appropriate for convolution networks, when there are hundreds of thousands to tens of millions of convolutions in each convolution layer. GPUs have thousands of cores or processors, and implementing small convolutions separately will not utilize all the cores and keep them busy all the time. Even if each core is assigned an entire 2D convolution to keep the GPU busy, the redundant and repeated load and store of the 2D input and output channels will overwhelm the GPU, and starve the cores.

The next sections present two practical implementations of the convolution operator for deep convolution networks. The first section presents a simple and elegant algorithm that transforms the many 2D convolutions needed in each layer into a single very big matrix-matrix

multiplication. An efficient and fast GPU version of matrix multiplication is implemented by the CUBLAS library provided by NVIDIA. CUBLAS is the CUDA version of the original BLAS library (Basic Linear Algebra Subprograms).

A.3.1 Convolution as Matrix Multiplication

Theoretically, the number of calculations performed using this algorithm is the same as the number of additions and multiplications done by the classical implementation of convolution by sliding the weight kernel squares throughout the 2D channels. However, in practice doing a single big matrix multiplication is much faster than doing hundreds of thousands or millions of small 2D convolutions. This algorithm starts each convolution layer by transforming all 2D input channels into a single 2D matrix called the **Toeplitz** matrix. Figure (A.3) shows the transformation of a single 2D input channel. Basically, if the filter size is $n \times n$, then each possible $n \times n$ adjacent square of pixels is transformed into a single row in Toeplitz matrix, and the $n \times n$ weight filter is transformed into a single column matrix. Then multiplying the rows of the Toeplitz matrix by the single column that represents the weight filter produces the output channel. Figure (A.3) shows the results of convolving a single input channel with a single 2×2 weight filter.

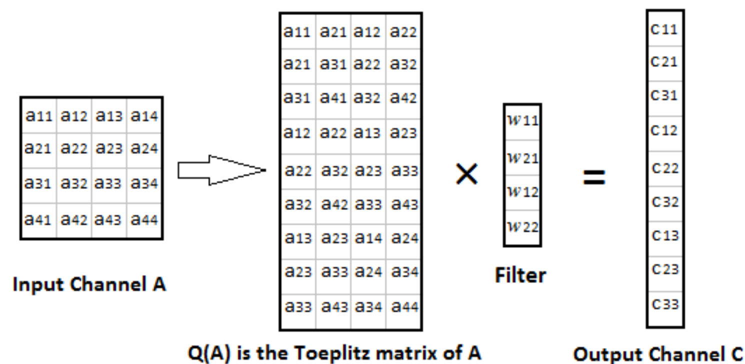

Figure A.3: a single 2D input channel is transformed into the Toeplitz matrix, then a single channel-filter pair convolution is done as a matrix multiplication.

In a convolution layer, there is a batch of images, each with multiple input channels and multiple output channels. Figure (A.4) shows how multiple Toeplitz matrices like the one generated in Figure (A.3) are all arranged together to form a single Toeplitz matrix for all input channels and for all images within the current batch. Figure (A.4) also shows how many single column filters like the one generated in figure (1.11) are arranged together to form a single filter matrix for all input channels and output channels. Once the unified Toeplitz matrix and the unified filter matrix are created a single matrix multiplication will generate all output channels

for all images. The example shown in figure (A.4) is for a convolution layer that has 3 input channels and 8 output channels, and a batch size of 4 images. Software packages like Caffe, Torch, Theano, and others implement variations of this algorithm, where the Toeplitz matrix is created explicitly, or created implicitly to save memory.

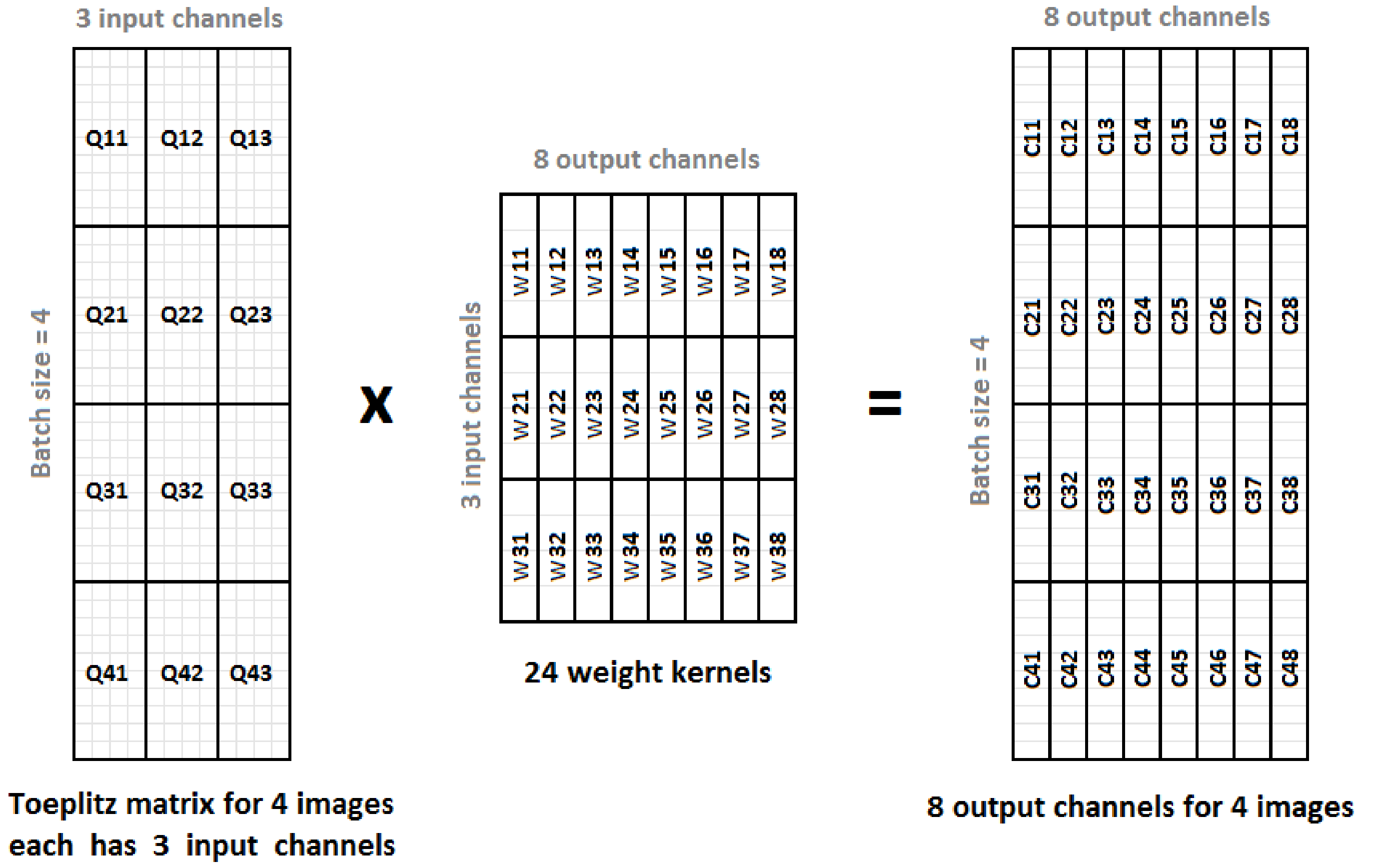

Figure A.4: A convolution layer, with 3 input channels and 8 output channels, implemented as a matrix multiplication between the Toeplitz matrix and the weight matrix, using a batch size of 4.

The complete gradient descent implementation of the convolution layer can be summarized in the following steps. In the forward pass, the input signals are propagated forward through each convolution layer as follows: -

- Forming a single big Toeplitz matrix for convolution layer i that includes all the input channels X_i for all the images in the current batch.

$$Q_i = Toeplitz(X_i) \tag{A.10}$$

- The convolution is implemented as a matrix-matrix multiplication between the Toeplitz matrix Q_i that contains all the input channels and the weight matrix W_i that contains all the weights filters in convolution layer i.

$$Y_i = Q_i \times W_i \tag{A.11}$$

In the backward pass, the error signals are propagated back until they reach the outputs of the convolution layer i. Once the error signals $\frac{\partial Loss}{\partial Y_i}$ are ready at the doorsteps of the convolution layer then: -

- The weight matrix of convolution layer i is updated using the following equation: -

$$\frac{\partial Loss}{\partial W_i} = {Q_i}^T \times \frac{\partial Loss}{\partial Y_i} \tag{A.12}$$

- The error signals are propagated back to the inputs of the convolution layer represented by the Toeplitz matrix using the following equation: -

$$\frac{\partial Loss}{\partial Q_i} = \frac{\partial Loss}{\partial Y_i} \times {W_i}^T \tag{A.13}$$

- The error signals of the Toeplitz matrix are then propagated back to the input channels before the Toeplitz transformation by doing an inverse Toeplitz transformation: -

$$\frac{\partial Loss}{\partial X_i} = InverseToeplitz\left(\frac{\partial Loss}{\partial Q_i}\right) \tag{A.14}$$

Where $Loss$ is the Cross-Entropy loss function between the ground truth outputs and the softmax outputs of the network. The forward signals include: X_i a 4D matrix that contains all the input channels to convolution layer i before the Toeplitz transformation, Q_i the Toeplitz transformation of X_i, and Y_i the output matrix of convolution layer i. The backward signals include $\frac{\partial Loss}{\partial Y_i}$ the error signal at the output Y_i, $\frac{\partial Loss}{\partial Q_i}$ the error signal at the Toeplitz matrix Q_i, $\frac{\partial Loss}{\partial X_i}$ the error signal at the input X_i, and $\frac{\partial Loss}{\partial W_i}$ the update values for the weights of convolution layer i. There is extra manipulation to these equations to include the biases in the weight matrices. However, in the case of using Batch Normalization biases will be included in the normalization step and not in the weight matrices. Also, the forward and backward signal propagation of the rectified linear activation function and the pooling stages are not included in these equations. The addition of padding is embedded in the implementation of the Toeplitz transformation. Except for the Toeplitz and inverse Toeplitz transformations, all the gradient decent equations for convolution layers are built as matrix-matrix multiplications, both in the forward and backward passes. CUBLAS library implements a fast matrix-matrix multiplication in CUDA that can run many times faster than the original BLAS library on the latest and fastest core i9 or Xeon Intel processors.

A.3.2 Winograd's minimal filtering Convolution

As figure (2.4) showed, the latest implementations (expect the Inception model) of deep convolution networks such as the VGG model [10], and residual model [1], use mainly small filters of size 3×3. Lavin et al. [63] 2015, used the concept of minimal filtering, and used the implementation from Winograd [64], to introduce a faster implementation of small sized 3×3 filters called Winograd's minimal filtering. This implementation theoretically calculates the same outputs as standard filters but with less mathematical operations. This is achieved by factorisations similar to $ax + ay$ and $a(x + y)$, where both terms calculate the same value, while the second term requires one less multiplication.

A minimal filter to compute m outputs with a one dimensional filter of size r is called $F(m, r)$ and requires $m + r - 1$ multiplications, while a regular filter requires $m \times r$ multiplications. Also a minimal filter to compute $m \times n$ outputs with a two dimensional filter of size $r \times s$ is called $F(m \times n, r \times s)$ and requires $(m + r - 1) \times (n + s - 1)$ multiplications, while a regular filter requires $m \times n \times r \times s$ multiplications. In order to calculate $m \times n$ outputs, a tile of size $(m + r - 1) \times (n + s - 1)$ data elements must be accessed, and therefore the number of multiplications using minimal filtering equals the number of accessed data elements.

F(2x2,3x3) implementation

The standard implementation of $F(2, 3)$ requires 6 multiplications, while using Winograd's minimal filtering requires 4 multiplications and the two outputs can be calculated as: -

$$\begin{pmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{pmatrix} \begin{pmatrix} g_0 \\ g_1 \\ g_2 \end{pmatrix} = \begin{pmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{pmatrix}$$

where: -

$m_1 = (d_0 - d_1)g_0 \quad m_2 = (d_1 + d_2)\frac{g_0+g_1+g_2}{2} \quad m_3 = (d_1 - d_3)g_2 \quad m_4 = (d_2 - d_1)\frac{g_0-g_1+g_2}{2}$

According to these equations the algorithm requires 4 multiplications which is equal to $m + r–1 = 2 + 3–1 = 4$ and so it is minimal. However, it also requires 4 additions involving the data d, 3 additions and 2 constant multiplications involving the filter g, and 4 extra additions to produce the final results. Why does replacing 6 multiplications and 5 additions with 4 multiplications, 2 constant multiplications, and 11 additions save computation? From the first look it seems there is an increase in the number of computations rather than reducing them, but it will soon be clear how this in fact reduces computation by a factor of 2 or even more

depending on the tile size. The filtering algorithm can be written in matrix form for clarity purposes.

$$Y = A^T \left[(G\ g) \cdot \left(B^T\ d \right) \right] \tag{A.15}$$

Where ($\cdot$) is the element wise multiplication and: -

$$B^T = \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \qquad G = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix} \qquad A^T = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{pmatrix}$$

$$g = (g_0\ g_1\ g_2)^T \qquad d = (d_0\ d_1\ d_2\ d_3)^T$$

Where B, G, and A are constant transformation matrices, and g is a 1D filter, and d is the 1D input data tile. The 1D minimal algorithm $F(2, 3)$ in Equation (A.15), can be nested with itself to obtain the 2D minimal algorithm $F(2 \times 2, 3 \times 3)$, in a matrix form as: -

$$Y = A^T \left[\left(G\ g\ G^T \right) \cdot \left(B^T\ d\ B \right) \right] A \tag{A.16}$$

Where G and B are the same matrices shown above, and g is a 3×3 filter and d is 4×4 input data tile. $F(2 \times 2, 3 \times 3)$ uses 16 multiplications while the standard algorithm uses $2 \times 2 \times 3 \times 3 = 36$ multiplications, which is a reduction with a factor of $\frac{23}{16} = 2.25$. However, the data transformation $\left(B^T\ d\ B \right)$ requires 32 additions, and the filter transformation $\left(G\ g\ G^T \right)$ requires 28 floating point operations, and the inverse transformation $A^T[\ ]A$ requires 24 additions. Again, it seems like the $2.25\times$ reduction in the number of multiplications is being overwhelmed by the data, filter, and inverse transformations. Before explaining why this is not the case we look back at the structure of the convolution layer, and how it was exploited to deal with this problem.

Assuming square filters, a convolution layer convolves $K \times C$ filters each with size $r \times r$ with $C \times N$ input channels each with size $H \times W$, to produce $K \times N$ output channels each of size $(H - r + 1) \times (W - r + 1)$, where C is the number of input channels per image, K is the number of output channels per image, and N is the number of images per batch. Denoting a single input channel as $D_{i,c}$, and a single filter as $G_{k,c}$, then a single output channel can be calculated as: -

$$Y_{i,k} = \sum_{c=1}^{C} D_{i,c} * G_{k,c} \tag{A.17}$$

Where i is the image index. To use the $F(m\times m, r\times r)$ algorithm to calculate the convolution of an input channel with size $H \times W$ with a filter of size $r \times r$, the input channel is divided into tiles of size $(m+r-1)\times(m+r-1)$ with $r-1$ overlap between neighboring tiles to yield $\lceil H/m \rceil \times \lceil W/m \rceil$ tiles per input channel. Rewriting equation (A.17) for a single tile with index $\tilde{x}$,$\tilde{y}$: -

$$Y_{i,k,\tilde{x},\tilde{y}} = \sum_{c=1}^{C} D_{i,c,\tilde{x},\tilde{y}} * G_{k,c} \tag{A.18}$$

Substituting $U = \left(G\ g\ G^T\right)$ and $V = \left(B^T\ d\ B\right)$ in equation (A.16) for the minimal 2D filtering algorithm $F(m \times m, r \times r)$ becomes:-

$$Y = A^T \left[U \cdot V\right] A \tag{A.19}$$

Substituting the minimal convolution in equation (A.19) with tile convolution in equation (A.18).

$$Y_{i,k,\tilde{x},\tilde{y}} = \sum_{c=1}^{C} A^T \left[U_{k,c} \cdot V_{c,i,\tilde{x},\tilde{y}}\right] A = A^T \left[\sum_{c=1}^{C} U_{k,c} \cdot V_{c,i,\tilde{x},\tilde{y}}\right] A \tag{A.20}$$

The inverse transform $A^T[\ ]A$ can be taken out of the summation, and calculated only once for all C channels. For a batch of N images the total number of tiles is $P = N \times \lceil H/m \rceil \times \lceil W/m \rceil$ tiles, and substituting the $i, \tilde{x}, \tilde{y}$ indices that point to a single tile with a single index p and ignoring the inverse transformation.

$$M_{k,p} = \sum_{c=1}^{C} U_{k,c} \cdot V_{c,p} \tag{A.21}$$

Where $(\ \cdot\ )$ is the element wise multiplication, and for the $F(3\times 3, 2\times 2)$ algorithm the size of the $U_{k,c}$ and $V_{c,p}$ matrices is 4×4. If each of the 16 elements of the element wise multiplication is labeled separately as (ξ, ν), and put in a separate equation then equation (A.21) can be substituted by 16 equations each of which represents a single element matrix multiplication to produce 1/16 of the results for a single output tile.

$$M_{k,p}^{\xi,\nu} = \sum_{c=1}^{C} U_{k,c}^{\xi,\nu} \cdot V_{c,p}^{\xi,\nu} \tag{A.22}$$

By arranging all the $U_{k,c}^{\xi,\nu}$ values, for all k and c in a single matrix $U^{\xi,\nu}$, and arranging all the $V_{c,p}^{\xi,\nu}$ values for all c and p in a single matrix $V^{\xi,\nu}$, equation (A.22) becomes 1 of 16 matrix

multiplications indexed by ξ, ν.

$$M^{\xi,\nu} = U^{\xi,\nu} \times V^{\xi,\nu} \tag{A.23}$$

Once the elements of U and V are calculated and arranged, the bulk of the convolution can be done using 16 matrix multiplications. Therefore, calculating a 2×2 tile using a 3×3 filter requires 16 multiplications for a single channel filter pair, while in the standard implementation requires $2 \times 2 \times 3 \times 3 = 36$ multiplications. Now going back to the 28 floating point operations required by the filter transformation $U = \left(G\ g\ G^T\right)$ and the 32 additions required by the data transformation $V = \left(B^T\ d\ B\right)$, will these extra operations to calculate the 2×2 tiles, overwhelm the reductions of the number of multiplications from 36 to 16. The answer is no, because these extra operations are not needed for each input and output tile. For example, the filter transformation $U_{k,c}$ in equations (A.21 and A.22) is only indexed by k and c that represent the indices of the filter matrix, while the input tile index p is missing, which means that the $K \times C$ filter transformations are common among all input tiles and need to be done only once. On the other hand, the data transformation $V_{c,p}$ in equations (A.21 and A.22) is only indexed by c and p that represent the indices of the input channels for all images in the batch, while the output channel index k is missing, which means that the $C \times P$ data transformations are common among all output channels, and need to be done only once. Therefore, the major component to calculate the convolutions using the minimal approach is the matrix multiplications which are reduced by a factor of 2.25. This is a complex algorithm but it is the fastest 3×3 convolution implementation for convolution layers, and we realised it with the CUDNN CUDA library, and it is about $2\times$ faster than other implementations.

A.4 Implementation and design principles

Current implementations of deep convolution networks are complex, and designing one from scratch can be a challenging task. This is a very active area of research, and new additions are frequently made to standard implementations. Training is based on calculating first order derivatives to update the network trainable parameters. The efficient way to construct such complex networks is to consider them as graphs, where adding a new function or stage to the network is like adding a new node to the graph. For a new stage to be plugged-in into the network, that stage has to propagate the input signal forward from the previous stage to the next stage, and it has to propagate the error signal backward in the opposite direction. Figure (A.5) shows how to plug-in a new stage in the network. At each stage the backpropagated error signal has two functions; first it is used to update the stage trainable parameters, then it is propagated back to earlier stages. Equation (A.24) shows the forward pass where f(x)

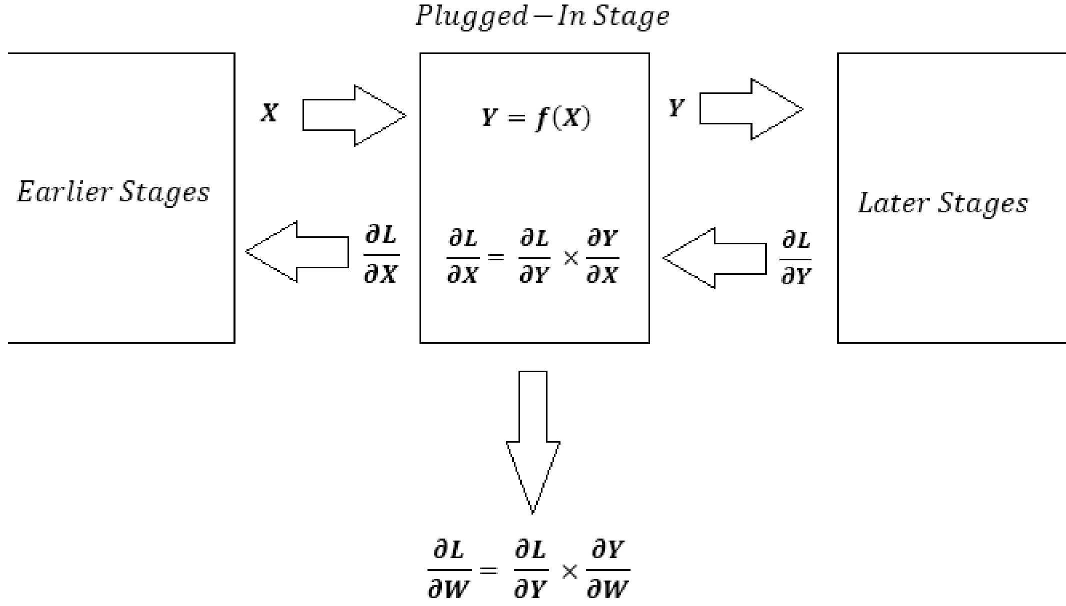

Figure A.5: a new stage is plugged-in into the network by propagating the input signal forward and error backward through that stage. The parameter update equations are not required if the stage has no trainable parameters.

represents the function implemented by that stage. Equation (A.25) shows how the error signal is propagated back to earlier stages, and equation (A.26) is used to update the stage trainable parameters. Both equations (A.25) and (A.26) use the chain rule of derivatives to propagate back the error signal.

$$Y = f(X) \tag{A.24}$$

$$\frac{\partial Loss}{\partial X} = \frac{\partial Loss}{\partial Y} \times \frac{\partial Y}{\partial X} \tag{A.25}$$

$$\frac{\partial Loss}{\partial W} = \frac{\partial Loss}{\partial Y} \times \frac{\partial Y}{\partial W} \tag{A.26}$$

The complexity of equations (A.25) and (A.26) used to propagate back the error signal depend on the complexity of the function $f(x)$ implemented by that stage. In many cases these equations can be divided into multiple steps as the case with the implementation of batch normalization. However, even if the backpropagation equations look very different from the forward function $f(x)$, the structure of the subroutines needed to implement these equations will still have some resemblance with subroutine needed to implement $f(x)$, and such resemblance can be exploited to help write those subroutines.

The backpropagated error signal $\frac{\partial Loss}{\partial Y}$ doesn't need to be stored as it is calculated on the fly, however the second term $\frac{\partial Loss}{\partial W}$ in the parameters update equation (A.26) depends on the inputs of the stage, or the output of the previous stage, and those outputs need to be stored.

Therefore, the output channels of each convolution layer calculated in the forward pass need to be stored, and then used in the backward pass to update the network parameters. With bigger batch sizes, the total size of these output channels is much bigger than the number of parameters in the network, and therefore most of the memory required by the network is used to store these output channels. In the case of adding batch normalization to the implementation of these convolution layers these output channels need to be stored twice, before and after the normalization.

As many layers, operators, functions, and stages can be added to the structure of deep convolution networks, it is practical to have a method that can test the correctness of the update equations which are based on first order derivatives. The standard method is to compare these equations with the numerical derivatives calculated using equation (A.27).

$$\frac{\Delta Loss}{\Delta w_i} = \frac{f(w_i + \Delta w_i) - f(w_i)}{2\Delta w_i} \tag{A.27}$$

Where $f(\ )$ is the function that maps the input image pixels to the output labels of the network. Numerical differentiation represented by equation (A.27) is easier to implement because it only requires the forward pass to calculate $f(\ )$. However, it requires a whole forward pass to calculate the numerical derivative of one parameter, and therefore it is not a practical substitution to the gradient decent equations. If the update equations of all the parameters in the network agree with their numerical counterparts, then the implementation is correct.

A.4.1 CUDA as an implementation tool

Convolution layers can require up to millions of full 2D convolutions per batch, which requires significant computation resources. The latest GPUs with thousands of cores provide about an order of magnitude more computation resources compared to the latest desktop and server CPUs. Nvidia titan x GPUs are used in this research, and they pack 3584 CUDA cores with about 11 TF of single precision compute power, and 12GB of RAM. CUDA programs are highly parallel, and debugging can be a challenge. Subroutines or functions are called CUDA kernels and they can be launched from CPU programs, or other CUDA kernels. The main difference between a CUDA program, and a CPU program, is that the number of threads launched by a CPU program reflect the number of independent tasks that need to be executed, while the number of threads launched in a CUDA kernel is proportional to the size of the input and output data of the kernel. Software tools that run CUDA programs provide debugging tools, however CUDA kernels are often debugged by comparing their results to a CPU version of the code. Therefore, when writing a complex CUDA program with many CUDA kernels, like the implementation of deep convolution networks, each kernel will be written and debugged

separately, and then all the kernels are plugged-in together to form the main program. Once the implementation is complete, the correctness of the program can be verified by comparing its results to another program that uses a numerical implementation of the derivatives, as it has been explained in the previous section.

CUDA Kernels

The number of threads that will execute a CUDA kernel are given as an input argument when the kernel is launched. Each thread will execute the same kernel on a different section of the input data, and the index of the thread will decide which part of the data (e.g. matrix) will be computed by that thread. Threads are divided into blocks, where the number of threads per block doesn't exceed the maximum block size, and where the global thread index is calculated using the block index, and local thread index within the block. The design of a good kernel depends mainly on how efficiently it utilizes the memory hierarchy in the CUDA environment. The programmable GPU memory hierarchy consists mainly of the main GPU memory which is visible to all threads, the shared memory (L1 cash) which is only visible to a single block of threads, and the register file (fastest) which is visible to a single thread. Data structures that reside in the main GPU memory are defined in the CPU code that launches the main kernels, shared memory is defined inside CUDA kernels and proceeded by the key word *shared*, and data that reside in registers are defined directly in CUDA kernels with a predefined size that can be resolved at compile time, and doesn't exceed the maximum available registers per thread. Data that are accessed multiple times by a single thread should be stored in the register file, data that are accessed by all threads in a single block should be stored in the shared memory, and the main input and output data structure of the kernel should be stored in the main GPU memory.

The following are two very simple examples of CUDA kernels, where the first calculates the rectified linear function $max(x, 0)$, and the second calculates the parametric version of the rectified linear function with 2 trainable parameters $a_1\ max(x, 0) + a_2\ min(x, 0)$ presented in section (1.4) in this chapter. These activations need to be computed for each element in the output channels of a convolution layer. For a convolution layer all output channels for all images in the current batch are stored as a single matrix. For the first kernel all elements in the matrix are treated equally, and therefore the thread indexing scheme is straight forward, where the the input and output buffers are treated as 1D matrices. For the second kernel, each output channel (across all images in the batch) has its own parameters a_1, and a_2, and therefore the thread indexing scheme should access the appropriate pair of parameters for each output channel.

Before launching a kernel, the block size (number of threads per block), and grid size (number of blocks) are defined using a simple CUDA data structure with 3 members of type integer,

called `dim3`. Therefore, the block size can have up to 3 dimensions (`x`, `y`, and `z`, and indexed inside the kernel by the predefined CUDA parameters `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`), and the grid size can have up to 3 dimensions (`x`, `y`, and `z`, indexed by `blockIdx.x`, `blockIdx.y`, and `blockIdx.z`). Often a block size of a single dimension `x` is used, while the number of grid size dimensions depends on the complexity of the indexing scheme used in the CUDA kernel.

For the first kernel that computes $max(x, 0)$, all elements of the input matrix are treated equally, and therefore a single block dimension, and a single grid dimension are sufficient. If the used block size is N threads, and the total number of elements in the output channels is $SIZE = ChSize * NumChs * BatchSize$ ($ChSize$ is the size of a single output channel, $NumChs$ is the number of output channels, and $BatchSize$ is the number of images in the current batch), then the block and grid sizes are defined as follows: -

```
dim3 blockSize(N, 1, 1);
dim3 gridSize((SIZE + N - 1)/N, 1, 1);
```

Only `blockSize.x` is used and is equal to N, while `blockSize.y`, and `blockSize.z` are not used and are equal to 1. Also, only `gridSize.x` is used and is equal to the number of thread blocks, while `gridSize.y` and `gridSize.z` are not used and are equal to 1. The reason for adding $N - 1$ to the numerator before the integer division is to guarantee that there are enough blocks to compute all the needed outputs. The kernel is launched using the following command:

```
ReLU<<<gridSize, blockSize>>>(Y, X, SIZE);
```

Where X and Y are the input and output buffers that store the output channels before and after applying the activation. The CUDA kernel is defined as follows:

```
__global__ void ReLU(float *Y, float *X, int SIZE)
{
   int indx = threadIdx.x + blockIdx.x*blockDim.x;
   if(indx<SIZE)
   {
      Y[indx] = fmaxf(X[indx],0);
   }
}
```

Where `indx` is the global thread index, `threadIdx.x` is the thread index within the block, `blockIdx.x` is the block index, and `blockDim.x` is the block size. Because the number of launched threads equals the number of elements in the input/output buffers, each thread will calculate a single element defined by the global index of that thread. The statement `if(indx<SIZE)` guarantees that the number of active threads do not exceed the total number of elements in the output channels, especially when the number of elements is not a multiple of the block size.

For the second kernel that computes $a_1\ max(x, 0) + a_2\ min(x, 0)$, each output channel should access a different part of the parameters matrix A that holds all the parameter pairs a_1, a_2 for all output channels in that convolution layer. Therefore, a more complex indexing scheme is needed, and the block and grid sizes are defined as follows:

```
dim3 blockSize(N, 1, 1);
dim3 gridSize((ChSize + N - 1)/N, NumChs, BatchSize);
```

Because the channel index that goes from 0 to $NumChs$ - 1 is needed to access the appropriate pair of parameters a_1, a_2, the arguments $ChSize$, $NumChs$, and $BatchSize$ need to be defined and used individually and not to be combined into a single argument $SIZE = ChSize * NumChs * BatchSize$, as is the case in the first kernel. The following command launches the kernel:

```
ParametricReLU<<<gridSize, blockSize>>>(Y, X, A, ChSize, NumChs);
```

Where A is the matrix that stores parameters a_1 and a_2 for all output channels in a convolution layer. The kernel is defined as follows:-

```
__global__ void ParametricReLU(float *Y, float *X, float *A, int
ChSize, int NumChs)
{
   __shared__ float a[2];
   int elem_ix = threadIdx.x + blockIdx.x*blockDim.x;
   int ch_ix = blockIdx.y;
   int batch_ix = blockIdx.z;

   if(threadIdx.x<2) a[threadIdx.x] = A[2*ch_ix + threadIdx.x];
```

```
    __syncthreads();

    if(elem_ix<ChSize)
    {
       int indx = batch_ix*NumChs*ChSize + ch_ix *ChSize + elem_ix;
       Y[indx] = a[0]*fmaxf(x[indx],0) + a[1]*fminf(x[indx],0);
    }
}
```

The reason for storing the parameter pair in a shared memory $a[2]$ is because all threads in a single block access elements from the same output channel, and therefore share these parameters. Also, because each channel has its own pair of parameters, the main parameter matrix A is accessed using the channel index ch_ix. Once the appropriate pair of parameters are accessed by each thread, the global thread index $indx$ is calculated. The calculation of $indx$ depends on the layout of the input matrix, and in the setup used here, the first dimension is the output channel size $ChSize$, the second dimension is the number of output channels $NumChs$, and the third dimension is the number of images in the current batch $BatchSize$. $indx$ is calculated using the element index $elem_ix$ (which goes from 0 to Ch_Size), the channel index ch_ix (which goes from 0 to Num_Chs), and the batch index $batch_ix$ (which goes from 0 to $Batch_Size$). These kernels only implement the forward pass for these activation functions. The function $max(x, 0)$ only requires propagating back the error signal from $\frac{\partial Loss}{\partial Y}$ to $\frac{\partial Loss}{\partial X}$, and doesn't require updating any parameters. On the other hand, the function $a_1\ max(x, 0) + a_2\ min(x, 0)$ requires both propagating back the error signal and updating the parameters a_1, and a_2.

The kernels presented here are simple to implement compared to more complex kernels such as the ones used to implement data augmentation, batch normalization, convolution operator etc. However, the same principles apply, and any CUDA kernel can be efficiently implemented by combining the appropriate thread indexing scheme with the appropriate usage of the memory hierarchy of the GPU. Implementing most of the network from scratch using CUDA kernels allows for greater flexibility in trying different implementations and ideas, compared to using ready software packages such as TensorFlow to implement the whole network.

Convolution implementation

The matrix-matrix multiplication implementation of the convolution operator presented in section (1.5.1) is the first implementation that has been tried in this research. The only tricky part was to implement the CUDA kernels for the Toeplitz transformation (equation (A.10)) that transfers the input channels into the Toeplitz matrix, and the inverse-Toeplitz transformation (equation (A.14)) that propagates the error signal back from the Toeplitz matrix to the input channels. These kernels are complex and must be thoroughly tested before they are

integrated into the main program. All the other operations are implemented as matrix-matrix multiplication, where an optimized function from the CUBLAS CUDA library is used to do the multiplication. The implementation was fairly fast and has worked well for medium size networks. However, the Toeplitz matrix was explicitly constructed, and it consumes much more memory than the output channels, and therefore it was unable to support larger networks. The second convolution operator used in this research is based on Winograd's minimal filtering convolution presented in section (1.5.2) and an optimized implementation by the CUDNN CUDA library is used. The reason for using Winograd's convolution is because most of the latest models use mainly small filters of size 3×3, and this implementation is about two times faster than the matrix-matrix multiplication for filters of size 3×3. All the other parts of the network were implemented as CUDA kernels, and in order for this convolution provided by CUDNN library to be integrated with the other kernels, the layout of the input/ouput channels of the convolution layers must be the same. CUDNN also provides implementations for the other major parts of the network such as pooling, batch normalization, softmax, etc. and comparing their results to the results of the corresponding CUDA kernels provides another way to test the correctness of those kernels.

A.4.2 Experiments on Parametric Linear Rectified Functions

He et al. [19] proposed a slightly different rectified function with a single parameter $f(x) = max(x, 0) + a\ min(x, 0)$, where a was initialized to be 0.25, and updated using gradient decent. Their results showed better performance than the original function $f(x) = max(x, 0)$. However, it was victory short lived, as the introduction of Batch normalization [2] has erased this advantage. The experiments presented here do not aim to design a superior activation function, which will be extraordinary, but rather to understand how linear rectified functions work. The rectified function $f(x) = max(x, 0)$ pieces together two lines $y_1 = x$, and $y_2 = 0$, and the function that will be tested here will generalize these two lines into $y_1 = a_1\ x$ and $y_2 = a_2\ x$ as equation (A.28) shows:

$$f(x) = a_1\ max(x, 0) + a_2\ min(x, 0) \tag{A.28}$$

If $a_1 = a_2$ then $f(x)$ collapses into a straight-line $f(x) = x$. There will be two experiments on the CIFAR10 data set using a deep convolution network with 19 convolution layers and one fully connected layer (the output layer). The network also uses two max pooling layers and one average pooling layer. In the first experiment, the starting values of the parameters of the activation function are set to $a_1 = 1$, $a_2 = 0.25$. These parameters are updated using gradient descent as training progresses and the update is not restricted, so a_1 and a_2 can end up having any real positive or negative value. Each output channel in each convolution layer

has its own parameters, which means that by the end of training there will be as many different activation functions as the number of output channels in the network. In the second experiment min pooling replaces max pooling, and the starting values of the parameters of the activation function are reversed to $a_1 = 0.25$, $a_2 = 1$. We believe that using $f(x) = max(x, 0)$ as activation function which passes the positive part of x is compatible with using max pooling. On the other hand, the activation function $f(x) = min(x, 0)$ which passes the negative part of x will be naturally compatible with min pooling. This explains why a_1 had a bigger starting value in the first experiment that uses max pooling, because a_1 is multiplied by the positive part $max(x, 0)$, and why a_2 had a bigger starting value in the second experiment that uses min pooling, because a_2 is multiplied by the negative part $min(x, 0)$.

	Error Rate
max pooling	5.014%
min pooling	4.97%

Table A.1: Shows the results of using parametric ReLU **(1)** with max pooling and starting from $a_1 = 1$, $a_2 = 0.25$ **(2)** with min pooling and starting from $a_1 = 0.25$, $a_2 = 1$.

Table (A.1) shows the results of the two experiments with max pooling and with min pooling, and shows that both designs are identical and had nearly the same error rate. Table (A.2) shows the average values of the parameters a_1 and a_2 for both experiments and for all 19 convolution layers. Each convolution layer has multiple output channels and each output channel has its own parameters, therefore these are the average parameter values per layer. Figure (A.6) shows the average activation values for 3 out of the 19 convolution layers. The first observation from table (A.2) and figure (A.6) is that the two experiments end up with symmetrically opposite activation functions. For example, the first layer in the network that uses max pooling ends up with $a_1 = 1.0565$ and $a_2 = 0.3117$, while the first layer in the network that uses min pooling ends up with $a_1 = 0.3197$ and $a_2 = 1.0521$. The same is true for all other 18 convolution layers. The second observation is that with max pooling a_1 always ends up with a higher value than a_2, while with min pooling a_2 always ends up with a higher value than a_1. This is expected because a_1 is multiplied by the positive part $max(x, 0)$, while a_2 is multiplied by the negative part $min(x, 0)$ in equation (A.28). The last observation is that despite being updated during training without constrains, the parameters a_1 and a_2 ended up with values that make the shape of the activation function similar to the original rectified linear function $f(x) = max(x, 0)$ when max pooling is used, and similar to $f(x) = min(x, 0)$ when min pooling is used. Actually, looking at table (A.2), as the network goes deeper and after each pooling layer, the values of a_2 with max pooling and the value of a_1 with min pooling goes close to zero and the activation function becomes very similar to the original $f(x) = max(x, 0)$ (and $f(x) = min(x, 0)$ with

min pooling).

The conclusion out of these three observations are. First, the choice of $f(x) = max(x, 0)$ is not unique, but it is compatible with max pooling because $max(x, 0)$ passes the positive part of the input x, and this function can easily be replaced with $f(x) = min(x, 0)$ when min pooling is used, to give the same results. Second, when max pooling is used the gradient decent algorithm pushes down the negative part $min(x, 0)$ of the input x by lowering the value of a_2 and ends up with an activation function that almost only passes the positive part $max(x)$. This shows that the choice of the $f(x) = max(x, 0)$ is natural and logical when combined with max pooling, and vice versa with min pooling. Thus the exact shape of the activation function is not important as long as there are two intersecting lines to approximate a nonlinear function, where either the positive or the negative part of the input is eliminated based on the used pooling operator.

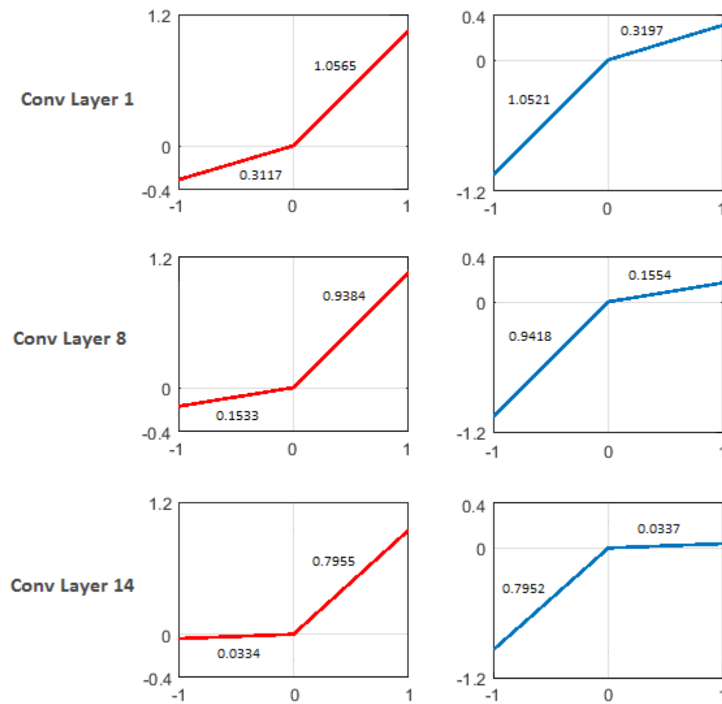

Figure A.6: shows the final activation functions for the 1st, 8th, and 14th layers. **Left:** using max pooling, **right:** using min pooling.

	Using max pooling Starting values a_1=1.0, a_2=0.25		Using min pooling Starting values a_1=0.25, a_2=1.0	
	a_1	a_2	a_1	a_2
Conv Layer 1	1.0565	0.3117	0.3197	1.0521
Conv Layer 2	1.0502	0.2935	0.3022	1.0502
Conv Layer 3	1.0580	0.2856	0.2798	1.0606
Conv Layer 4	1.0730	0.2698	0.2612	1.0773
Conv Layer 5	1.0855	0.2821	0.2873	1.0862
Conv Layer 6	1.0901	0.2841	0.2749	1.0944
Conv Layer 7	1.0952	0.3353	0.3398	1.0915
	max pooling		min pooling	
Conv Layer 8	0.9384	0.1533	0.1554	0.9418
Conv Layer 9	0.9453	0.1162	0.1179	0.9489
Conv Layer 10	0.9520	0.1210	0.1186	0.9564
Conv Layer 11	0.9658	0.1223	0.1252	0.9649
Conv Layer 12	0.9604	0.1677	0.1655	0.9627
Conv Layer 13	0.9658	0.1932	0.1951	0.9660
	max pooling		min pooling	
Conv Layer 14	0.7955	0.0334	0.0337	0.7952
Conv Layer 15	0.8069	0.0218	0.0221	0.8091
Conv Layer 16	0.7972	0.0182	0.0179	0.7942
Conv Layer 17	0.7819	0.0151	0.0151	0.7846
Conv Layer 18	0.7973	0.0103	0.0102	0.7983
Conv Layer 19	0.8306	0.0216	0.0220	0.8294

Table A.2: shows the average values of a_1 and a_2 for all 19 convolution layers, for both, the max pooling model, and the min pooling model.

A.5 Interaction between residual connections and BN statistics

This analysis is part of of the experiments carried out in chapters two and six about BN. As we were trying to analyze the distribution of the shared normalization statistics of BN we calculated the average of the means, and the average of the variances at each convolutional layer (the number of mean-variance pairs at each layer is equal to the number of the output channels at that layer). Figure (A.7) shows the average mean and the average variance for all 34 convolutional layers for the standard network trained on the 10-class dataset in section (3.5). These are the averages per layer of the means and variances of the fixed set used in the inference stage, and calculated using the entire training set. The 34-layer residual network in figure (3.1) starts by a single convolutional layer followed by 16 residual blocks followed by the FC output layer. For the 16 residual blocks the identity connection jumps ahead two consecutive convolutional layers. Figure (A.7) also shows that the magnitude of these statistics forms a peak every two consecutive layers. To prove that the location of these peaks was caused by location of the residual jump-ahead connections, we wanted to repeat the experiment using different residual block sizes where the location of the residual connections vary (jump ahead different numbers of consecutive layers). For this purpose we used the 18-layer network version from [1] because its difficult to use bigger residual blocks (with more than two consecutive layers) with the 34-layer network in figure (3.1) without changing the number of convolutional layer at each resolution point (the change in structure should only include the location of the residual connections). We repeated the experiment using the same 10-class dataset used in section (3.5), and using the same hyperparameters used with the 34-layer network in section (3.5). The experiment was repeated 3 times, where in the first experiment the residual connections jump ahead 2 consecutive layers, in the second experiment the residual connections jump ahead 4 consecutive layers, and in the third experiment the residual connections were completely removed from the network. Figure (A.8), shows the average mean and the average variance per layer for all three experiments. Its clear that the location of the peaks in figure (A.8) follows to a great degree the location of the residual connections in each of the three experiments. This clearly points to a correlation between two components (batch normalization and residual connections) that were designed independently and for different purposes. Usually such inter-dependency between the functions of different components of CNNs can be noticed when the improvements achieved using such components are not orthogonal. These results were not in particular helpful in understanding how the network was using the shared normalization statistics of balanced batches in chapter two, or how the multi-task network were using the normalization statistics to separate between tasks in chapter six, which is the reason why we did not include them either in chapter two or six. However, i think its still interesting to quantitatively measure such

interaction and interdependency between two different network components that were designed for different purposes and were introduced separately and be able to plot it and visualize it as figures (A.7, A.8) showed.

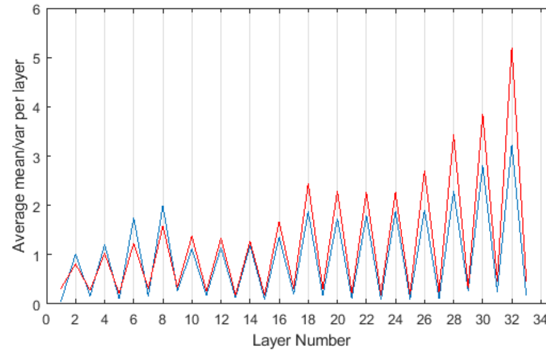

Figure A.7: **Left:** The average BN statistic (the average mean and the average variance) per layer for all 34 convolutional layers. It show an interaction between the magnitude of these statistics and the location of the residual connections which jumps 2 consecutive layers.

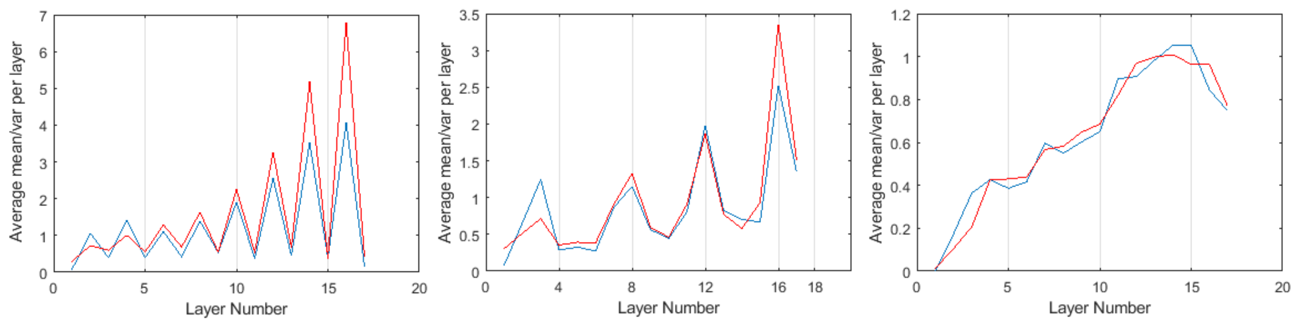

Figure A.8: **Left:** The average mean and the average variance per layer for all 18 convolutional layers. The experiment was repeated 3 times, **Left**: the residual connections jump-ahead 2 consecutive layers, **middle**: the residual connections jump-ahead 4 consecutive layers, and **right**: the residual connections are removed. Again the 3 figures indicate that there is an interaction between the location of the residual connections and the magnitude of the normalization statistics of BN.

APPENDIX B

Results and auxiliary tables

B.1 Introduction

This appendix includes tables for detailed results form chapter six, and also auxiliary tables that show include the folder names of classes from ImageNet used to construct multiple datasets with different sizes. Tables (B.1) to (B.4) contain detailed results per task for the multitask network from chapter six. The training images for the 1000 class objects of ImageNet are stored in 1000 folders. Tables (B.5) to (B.13) contain the folder class names used to construct the datasets to carry out experiments in this thesis.

	Without BN			With BN	
	Single Task	Multi Task		Single Task	Multi Task
Dataset 1	7.2%	15.6%		5.97%	3.47%
Dataset 2	5.8%	9.9%		4.98%	3.2%
Dataset 3	5.5%	10.3%		4.64%	2.75%
Dataset 4	6.7%	10.8%		5.72%	3.56%
Dataset 5	7.4%	12.6%		6.17%	5%
Dataset 6	3.8%	6.4%		3.02%	1.67%
Dataset 7	2.9%	7.8%		2.12%	1.4%
Dataset 8	6.9%	14.8%		6.08%	4.64%
Dataset 9	4.4%	7.5%		3.56%	2.12%
Dataset 10	6.4%	10.6%		5.9%	4.55%
Average	5.7%	10.6%		4.81%	3.23%

Table B.1: Results for the first experiment, 10 classes per task. The results show the multitask network with BN outperforming the 10 single task networks, while failing without BN.

	Without BN			With BN	
	Single Task	Multi Task		Single Task	Multi Task
Dataset 1	8.5%	15.7%		7.3%	5.5%
Dataset 2	8.1%	16.0%		6.6%	4.86%
Dataset 3	8.9%	18.4%		7.9%	5.46%
Dataset 4	10.3%	17.6%		9.4%	6.9%
Dataset 5	8.86%	16.4%		7.6%	5.94%
Dataset 6	9.94%	18.3%		8.54%	6.25%
Dataset 7	7.74%	16.1%		6.8%	4.03%
Dataset 8	9.82%	18.5%		8.47%	6.6%
Dataset 9	8.22%	16.7%		7.0%	4.8%
Dataset 10	8.74%	16.1%		7.54%	5.0%
Average	8.92%	16.7%		7.71%	5.53%

Table B.2: Results for second experiment, 50 classes per task. The results show the multitask network with BN outperforming the 10 single task networks, while failing without BN.

	Without BN			With BN	
	Single Task	Multi Task		Single Task	Multi Task
Dataset 1	11.64%	22.32%		10.8%	7.56%
Dataset 2	12.32%	23.86%		11.32%	8.88%
Dataset 3	10.26%	21.32%		9.54%	6.98%
Dataset 4	9.0%	21.48%		8.16%	6.44%
Dataset 5	11.3%	23.04%		10.46%	7.48%
Dataset 6	11.34%	22.36%		10.5%	7.54%
Dataset 7	11.06%	21.78%		10.32%	7.8%
Dataset 8	11.92%	22.52%		10.62%	7.64%
Dataset 9	10.5%	22.82%		10.42%	7.42%
Dataset 10	11.32%	21.1%		10.24%	8.36%
Average	11.07%	22.26%		10.24%	7.61%

Table B.3: Results for the third experiment, 100 classes per task. The results show the multitask network with BN outperforming the 10 single task networks, while failing without BN.

20 Tasks, 50 classes/task		
	ST	MT
Task 1	8.36%	4.73%
Task 2	8.6%	5.98%
Task 3	8.71%	5.28%
Task 4	10.13%	6.37%
Task 5	7.97%	4.63%
Task 6	6.7%	3.74%
Task 7	8.04%	4.77%
Task 8	5.86%	3.24%
Task 9	9.07%	4.95%
Task 10	9.03%	5.52%
Task 11	8.15%	5.66%
Task 12	9.28%	6.51%
Task 13	8.03%	5.54%
Task 14	7.48%	4.4%
Task 15	8.51%	5.37%
Task 16	8.89%	4.94%
Task 17	6.84%	4.26%
Task 18	9.35%	6.62%
Task 19	7.44%	5.52%
Task 20	10.3%	6.23%
Average	8.34%	5.21%

30 Tasks, 33 classes/task		
	ST	MT
Task 1	8.2%	4.64%
Task 2	5.16%	2.8%
Task 3	8.62%	5.32%
Task 4	9.34%	5.28%
Task 5	8.3%	5.75%
Task 6	5.78%	2.9%
Task 7	8.77%	5.26%
Task 8	4.83%	2.83%
Task 9	6.59%	3.1%
Task 10	5.98%	3.05%
Task 11	6.2%	3.57%
Task 12	5.19%	3.65%
Task 13	7.24%	4.27%
Task 14	7.96%	4.56%
Task 15	6.68%	4.13%
Task 16	5.47%	3.21%
Task 17	9.52%	4.94%
Task 18	8.26%	4.73%
Task 19	6.43%	4.15%
Task 20	7.25%	3.28%
Task 21	7.37%	4.55%
Task 22	6.9%	3.74%
Task 23	7.84%	4.66%
Task 24	7.93%	4.71%
Task 25	6.54%	3.68%
Task 26	5.29%	3.2%
Task 27	7.96%	4.5%
Task 28	6.63%	3.38%
Task 29	6.57%	4.16%
Task 30	8.98%	5.22%
Average	7.13%	4.1%

50 Tasks, 20 classes/task		
	ST	MT
Task 1	8.32%	4.57%
Task 2	3.16%	2.28%
Task 3	3.95%	2.41%
Task 4	5.53%	3.04%
Task 5	7.03%	3.3%
Task 6	6.24%	3.04%
Task 7	7.55%	3.55%
Task 8	6.06%	3.93%
Task 9	6.24%	2.92%
Task 10	8.01%	3.05%
Task 11	8.17%	4.82%
Task 12	3.43%	1.52%
Task 13	4.22%	2.28%
Task 14	5.36%	2.67%
Task 15	5.27%	2.28%
Task 16	2.98%	1.52%
Task 17	6.676%	3.42%
Task 18	4.57%	2.41%
Task 19	3.43%	0.91%
Task 20	5.18%	1.78%
Task 21	5.8%	2.79%
Task 22	5.71%	3.3%
Task 23	7.98%	4.18%
Task 24	7.2%	2.79%
Task 25	4.74%	2.41%
Task 26	6.07%	2.15%
Task 27	3.69%	1.65%
Task 28	8.61%	4.95%
Task 29	7.21%	4.57%
Task 30	5.01%	2.66%
Task 31	5.1%	3.55%
Task 32	7.82%	3.93%
Task 33	4.21%	2.03%
Task 34	6.59%	3.68%
Task 35	4.22%	1.52%
Task 36	6.94%	2.16%
Task 37	6.41%	3.17%
Task 38	6.76%	2.92%
Task 39	8.96%	5.2%
Task 40	4.48%	2.28%
Task 41	3.6%	2.41%
Task 42	4.75%	2.16%
Task 43	6.76%	5.84%
Task 44	5.18%	3.43%
Task 45	9.49%	4.7%
Task 46	4.66%	1.9%
Task 47	4.21%	2.66%
Task 48	6.32%	3.67%
Task 49	7.2%	3.42%
Task 50	9.3%	5.2%
Average	5.94%	3.06%

Table B.4: Results per task for the 20, 30 and 50 tasks obtained using single task networks vs using a single multitask network.

Table B.5: Folder names of 100 Classes sampled randomly from ImageNet ILSVRC 2015 and randomly divided into 10 equally sized datasets.

Part 1	n04270147	n04591713	n01978287	n04604644	n03602883	n01819313	n02114712	n03146219	n07753275	n07802026
Part 2	n02051845	n07871810	n07745940	n02892767	n04154565	n02006656	n02655020	n04562935	n04120489	n07718472
Part 3	n03661043	n01644373	n04330267	n13052670	n03733131	n03976467	n03908618	n02480495	n03633091	n02086240
Part 4	n03787032	n01631663	n02112137	n01677366	n01807496	n04192698	n03759954	n02134084	n09399592	n01632777
Part 5	n02676566	n02417914	n03933933	n04039381	n02091134	n02841315	n02692877	n03527444	n03770439	n03887697
Part 6	n02111129	n09468604	n03534580	n02028035	n01882714	n02859443	n04579432	n02206856	n03187595	n02099601
Part 7	n03857828	n02106382	n02869837	n03697007	n04326547	n04540053	n02992211	n01978455	n02002724	n07711569
Part 8	n04118776	n02105855	n04008634	n02102480	n04399382	n02219486	n02091467	n02105251	n03259280	n02769748
Part 9	n06785654	n04037443	n03110669	n02971356	n04335435	n02110806	n03788365	n03786901	n02326432	n03000247
Part 10	n01749939	n01689811	n02883205	n03840681	n04350905	n01914609	n02992529	n02672831	n01518878	n02129604

Table B.6: Folder names of 100 Classes sampled from ImageNet ILSVRC 2015 to form 10 categories, each category has 10 classes.

Part 1	n02701002	n02814533	n02930766	n03100240	n03594945	n03769881	n03770679	n03930630	n03977966	n04285008
Part 2	n01773797	n01774384	n01775062	n02165105	n02167151	n02168699	n02174001	n02177972	n02229544	n02233338
Part 3	n02123045	n02123159	n02123394	n02123597	n02124075	n02127052	n02128385	n02128757	n02128925	n02130308
Part 4	n03063599	n03063689	n03443371	n03775546	n03786901	n03950228	n04398044	n04522168	n07920052	n07930864
Part 5	n01530575	n01531178	n01532829	n01534433	n01537544	n01558993	n01560419	n01580077	n01592084	n01828970
Part 6	n07720875	n07742313	n07745940	n07747607	n07749582	n07753113	n07753592	n07768694	n12620546	n12768682
Part 7	n02870880	n03014705	n03016953	n03018349	n03125729	n03131574	n03201208	n03337140	n04099969	n04550184
Part 8	n01629819	n01630670	n01631663	n01632458	n01675722	n01682714	n01685808	n01687978	n01689811	n01693334
Part 9	n02483362	n02486261	n02486410	n02487347	n02488291	n02492035	n02492660	n02493509	n02493793	n02494079
Part 10	n01728572	n01728920	n01729322	n01734418	n01735189	n01740131	n01742172	n01753488	n01755581	n01756291

Table B.7: 1st **half:** folder names of 500 Classes sampled randomly from ImageNet ILSVRC 2015 and randomly divided into 10 equally sized datasets.

Part 1									
n03584829	n02391049	n13040303	n02102177	n03666591	n03445924	n02480855	n02009912	n01601694	n02607072
n02092339	n03884397	n02412080	n04317175	n03891332	n03223299	n02088466	n07715103	n02110958	n04560804
n02101006	n12057211	n01775062	n03529860	n02090721	n01675722	n03837869	n02279972	n03599486	n02437312
n03450230	n01580077	n04483307	n04074963	n03877472	n04118538	n02415577	n03388549	n03179701	n02974003
n02111277	n02106166	n02701002	n02100735	n04044716	n07695742	n01616318	n07753592	n01774384	n04009552
Part 2									
n07615774	n01735189	n04487394	n01537544	n07920052	n03929855	n02951358	n04285008	n04344873	n03355925
n01968897	n02097209	n02087394	n01665541	n01820546	n03271574	n04447861	n02236044	n02443114	n04553703
n06596364	n03485794	n04606251	n03814906	n02134418	n03445777	n02101556	n02113023	n03482405	n02865351
n07892512	n03127925	n03761084	n03042490	n02948072	n03032252	n02807133	n01756291	n03623198	n04515003
n02169497	n04435653	n02137549	n04141327	n02492660	n09332890	n02095570	n01860187	n02113712	n03598930
Part 3									
n03782006	n03457902	n01498041	n03937543	n04229816	n03733281	n01664065	n02259212	n03461385	n03584254
n02096051	n02105505	n03347037	n02504458	n03109150	n02097298	n02027492	n03871628	n03676483	n04204238
n01797886	n02037110	n01774750	n02536864	n02085936	n04026417	n04372370	n04461696	n02834397	n03657121
n07753113	n02018795	n02119022	n09246464	n02687172	n07716358	n04532670	n04366367	n02231487	n01682714
n02514041	n02395406	n04275548	n03724870	n02490219	n03930630	n01872401	n02177972	n04019541	n03983396
Part 4									
n07614500	n02966193	n02115641	n04367480	n01824575	n07760859	n03028079	n04462240	n02190166	n04589890
n03729826	n02110185	n02786058	n04131690	n02776631	n02111500	n02927161	n02120079	n03443371	n02256656
n02483362	n03141823	n04065272	n02110063	n03691459	n03345487	n03980874	n12768682	n04252077	n03014705
n09421951	n02091244	n07754684	n02892201	n02804414	n02979186	n01644900	n09428293	n02325366	n02093256
n03134739	n03670208	n02100236	n03207743	n01833805	n03590841	n01843065	n02788148	n02114548	n03452741
Part 5									
n02443484	n07768694	n04136333	n02111889	n02277742	n03877845	n01629819	n03891251	n02009229	n02930766
n02730930	n07693725	n02112018	n02804610	n03417042	n01532829	n01734418	n03873416	n02843684	n07579787
n02124075	n01795545	n04442312	n02093754	n02106662	n01855672	n02108551	n01784675	n15075141	n01742172
n03710721	n02119789	n01632458	n02107574	n03026506	n10565667	n02457408	n04081281	n03535780	n07565083
n02099849	n01630670	n01990800	n02965783	n03794056	n03459775	n02493509	n01692333	n03594945	n03188531

Table B.8: 2nd **half:** folder names of 500 Classes sampled randomly from ImageNet ILSVRC 2015 and randomly divided into 10 equally sized datasets.

Part 6									
n02096294	n02437616	n03743016	n02910353	n03290653	n02125311	n04228054	n02445715	n12267677	n02977058
n02281787	n02749479	n02098286	n01798484	n04258138	n03868863	n03047690	n03018349	n02790996	n11879895
n07720875	n02423022	n02097047	n03492542	n03637318	n02094114	n04548362	n03384352	n02483708	n03337140
n03481172	n07684084	n02837789	n03777568	n07697537	n02120505	n04286575	n02276258	n02106550	n04149813
n03063689	n03125729	n07749582	n02442845	n03785016	n03388183	n02643566	n01980166	n02504013	n04296562
Part 7									
n02109525	n04371774	n04254120	n04252225	n02128925	n02825657	n04141076	n04486054	n04482393	n03065424
n07697313	n04116512	n02025239	n02870880	n04417672	n03196217	n02280649	n02403003	n01796340	n03126707
n02113186	n01944390	n03124043	n02802426	n03673027	n03902125	n02489166	n02486261	n01755581	n03888257
n02085620	n02641379	n02086646	n11939491	n02879718	n02104029	n02123045	n04505470	n03495258	n02361337
n02264363	n02093991	n09472597	n02497673	n03494278	n03532672	n01514859	n02447366	n02108089	n02108000
Part 8									
n01740131	n04263257	n02092002	n03804744	n02094433	n02098413	n02130308	n02165456	n02100877	n04099969
n09835506	n02398521	n03720891	n02487347	n03180011	n02797295	n03544143	n04592741	n03530642	n07836838
n10148035	n02077923	n01494475	n03627232	n01694178	n01770393	n03617480	n03930313	n02999410	n03692522
n04235860	n02123159	n04067472	n02088364	n02102040	n09256479	n04356056	n04532106	n04398044	n13044778
n02666196	n07583066	n02167151	n03496892	n02793495	n02835271	n01817953	n09193705	n02795169	n02127052
Part 9									
n02108915	n02133161	n03272010	n04179913	n04243546	n02988304	n02099267	n02840245	n07880968	n02281406
n03325584	n03394916	n03977966	n03868242	n01768244	n04557648	n02268853	n02097474	n03393912	n02342885
n02088238	n04033995	n03763968	n02410509	n02321529	n02980441	n02229544	n02777292	n02408429	n06359193
n01693334	n04090263	n02113624	n02917067	n01592084	n04251144	n03992509	n01496331	n13054560	n01641577
n03404251	n04162706	n06794110	n03016953	n01697457	n02951585	n03982430	n07831146	n02815834	n03220513
Part 10									
n04476259	n04392985	n04141975	n02112350	n02791124	n03710193	n01669191	n02481823	n01685808	n04086273
n01729977	n02106030	n01443537	n02226429	n02389026	n03249569	n03255030	n03201208	n01871265	n01667778
n02510455	n03388043	n04599235	n03721384	n03662601	n04523525	n01748264	n04613696	n04254777	n04336792
n02441942	n02110341	n02808304	n03291819	n03792782	n07742313	n03935335	n02011460	n04465501	n07747607
n03788195	n02056570	n09229709	n02088094	n01877812	n04404412	n04517823	n02799071	n02013706	n01985128

Table B.9: **Parts 1 ans 2:** folder names of classes sampled randomly from ImageNet ILSVRC 2015.

Part 1									
n04141327	n03637318	n02963159	n04277352	n02091244	n02607072	n02877765	n01667778	n02447366	n03141823
n02791270	n04152593	n04023962	n02108422	n03124043	n01943899	n04325704	n01797886	n02165456	n02110185
n01669191	n02917067	n02492035	n03197337	n01833805	n09835506	n02086910	n01695060	n04540053	n01692333
n02119789	n04560804	n03976467	n02106166	n02025239	n02276258	n02108000	n02692877	n02105641	n02102177
n02992211	n01807496	n04350905	n01775062	n04335435	n03133878	n02966687	n04532670	n01614925	n03937543
n01756291	n04179913	n02927161	n03126707	n01685808	n02892201	n02113186	n04428191	n03584254	n02769748
n02090379	n02361337	n01484850	n04332243	n01737021	n13052670	n04536866	n04553703	n04562935	n04429376
n02094433	n01755581	n02364673	n01518878	n07754684	n04366367	n04493381	n04591713	n04501370	n01986214
n03063689	n04120489	n03888605	n01729322	n02086079	n02672831	n02977058	n03376595	n02099849	n09256479
n02422699	n03272010	n01694178	n01749939	n02096585	n01873310	n01955084	n03983396	n04286575	n01930112
Part 2									
n04111531	n03995372	n03895866	n01968897	n02442845	n01630670	n03544143	n04311174	n02655020	n01773797
n04033995	n01796340	n01980166	n03594734	n06785654	n03992509	n04483307	n03452741	n01776313	n02823750
n03482405	n02892767	n02444819	n04542943	n02105505	n02667093	n02776631	n03095699	n03075370	n04192698
n03887697	n01443537	n02120079	n04523525	n02281787	n03794056	n01608432	n07920052	n02268443	n03146219
n02514041	n04254120	n03781244	n02328150	n03404251	n02097130	n09428293	n03476684	n01795545	n13040303
n02106550	n07614500	n03908714	n02107683	n04461696	n03249569	n07717556	n04404412	n04328186	n03179701
n04037443	n04009552	n03658185	n01914609	n03777754	n04238763	n01855032	n04074963	n07742313	n01558993
n03840681	n03026506	n02091831	n01806567	n01675722	n02106030	n06794110	n02088238	n04347754	n02106382
n01770393	n02492660	n03759954	n04118538	n03770679	n04604644	n04592741	n02807133	n02091032	n07836838
n02077923	n01728920	n02168699	n03599486	n07930864	n03970156	n04435653	n02979186	n03041632	n04254680

Table B.10: **Parts 3 ans 4:** folder names of classes sampled randomly from ImageNet ILSVRC 2015.

Part 3									
n02747177	n02443114	n04081281	n03980874	n09421951	n04254777	n04090263	n04465501	n02105162	n03868242
n02099712	n01990800	n02280649	n07716358	n02999410	n07760859	n04116512	n02099601	n02097298	n07871810
n04515003	n02110341	n03290653	n02102318	n04311004	n04310018	n03803284	n03884397	n12985857	n04044716
n02256656	n02109961	n02104365	n02051845	n06359193	n03788195	n03814906	n04239074	n04479046	n03042490
n01753488	n01560419	n03843555	n02687172	n02229544	n02096177	n01917289	n01773157	n02114855	n02494079
n02056570	n02091467	n01592084	n02483708	n02264363	n07720875	n03089624	n02980441	n02326432	n02454379
n04204347	n04204238	n07590611	n02808440	n02102040	n04487394	n04266014	n02058221	n02701002	n02113624
n02013706	n04591157	n01748264	n02006656	n07613480	n03018349	n04517823	n03259280	n07565083	n04392985
n02389026	n02094114	n07684084	n03709823	n03976657	n02504013	n04507155	n02814533	n02808304	n02119022
n02097209	n09332890	n03602883	n07583066	n02690373	n02172182	n02396427	n02277742	n03530642	n03127747
Part 4									
n04554684	n03272562	n02125311	n03425413	n02087046	n04201297	n03857828	n02493793	n03733131	n02128757
n02113712	n02089973	n03710721	n03187595	n12620546	n04606251	n02112706	n04376876	n03673027	n02951358
n02906734	n04136333	n04208210	n02111277	n02480855	n02086240	n09229709	n02090721	n01665541	n13054560
n03916031	n03476991	n02123045	n02510455	n02403003	n01877812	n02093991	n07753592	n12144580	n04146614
n03982430	n01820546	n04579145	n02127052	n02509815	n03180011	n03291819	n03697007	n03131574	n03729826
n02009912	n02817516	n03347037	n02096437	n02116738	n01983481	n02100583	n02666196	n03942813	n02981792
n02102973	n01855672	n03594945	n03393912	n03891332	n02097658	n02090622	n03538406	n02974003	n03871628
n12057211	n02111500	n03791053	n09399592	n02443484	n02951585	n04552348	n03832673	n01828970	n02088094
n02843684	n03782006	n02489166	n01774750	n02085936	n03207743	n04099969	n03388183	n02841315	n04458633
n03874599	n04275548	n01514859	n04367480	n03657121	n02085620	n01734418	n03662601	n03495258	n03443371

Table B.11: **Parts 5 ans 6:** folder names of classes sampled randomly from ImageNet ILSVRC 2015.

Part 5									
n02860847	n03201208	n02259212	n15075141	n02910353	n03935335	n03776460	n01689811	n02099429	n02487347
n02098286	n03000684	n02643566	n02101556	n01751748	n03642806	n03787032	n03384352	n03792782	n02098105
n02110627	n02486261	n01496331	n07695742	n02097047	n04229816	n03876231	n02093428	n03930313	n02236044
n02105412	n03314780	n01601694	n04525305	n02123394	n02500267	n03325584	n02504458	n02894605	n13133613
n02107908	n02093256	n07753275	n01622779	n03450230	n03792972	n03788365	n01697457	n03000134	n03125729
n02948072	n02190166	n01770081	n04141975	n03866082	n02814860	n04579432	n02009229	n02325366	n09468604
n02096294	n02091635	n01582220	n04613696	n06596364	n02033041	n01532829	n03761084	n03666591	n02909870
n12768682	n02104029	n02869837	n02114548	n02128925	n03796401	n07715103	n02093754	n04040759	n03017168
n03196217	n03417042	n02791124	n02105251	n02086646	n01739381	n02106662	n02797295	n01514668	n07745940
n07615774	n02138441	n01491361	n03775546	n02169497	n03775071	n02879718	n01798484	n02883205	n07714990
Part 6									
n07831146	n02268853	n02111889	n03710637	n01632777	n02342885	n03854065	n04259630	n04487081	n07860988
n04243546	n02100735	n01784675	n02233338	n02640242	n02114367	n03032252	n03764736	n07873807	n02840245
n01818515	n02101388	n02165105	n03062245	n02099267	n02120505	n11939491	n03535780	n04033901	n02895154
n01440764	n07880968	n02091134	n04258138	n02012849	n03355925	n12998815	n07697537	n02871525	n02133161
n02930766	n02794156	n03720891	n03721384	n04153751	n02088466	n03825788	n04584207	n02167151	n02641379
n04548280	n02088364	n02134418	n04263257	n01631663	n02095570	n04418357	n04270147	n02123597	n04525038
n03085013	n04008634	n04069434	n04423845	n03991062	n01531178	n03388549	n02137549	n03680355	n02971356
n07718747	n03944341	n02422106	n03400231	n02490219	n07932039	n03649909	n02317335	n04509417	n03014705
n03873416	n04417672	n01740131	n01534433	n02071294	n01872401	n02480495	n04125021	n02669723	n03938244
n03216828	n04070727	n02795169	n02699494	n01945685	n02488291	n03676483	n01984695	n02410509	n02226429

Table B.12: **Parts 7 ans 8:** folder names of classes sampled randomly from ImageNet ILSVRC 2015.

Part 7									
n02391049	n03379051	n02018795	n04371430	n03733805	n03670208	n01735189	n04200800	n02088632	n02111129
n03028079	n04118776	n04326547	n03492542	n02412080	n03630383	n03345487	n04550184	n03388043	n02837789
n03584829	n03956157	n02007558	n03220513	n09193705	n04344873	n03690938	n03877845	n07718472	n03481172
n03769881	n02606052	n03773504	n04522168	n01843383	n03063599	n03127925	n02676566	n02093647	n03786901
n10148035	n04443257	n04039381	n03598930	n02100877	n04389033	n02108089	n03424325	n01682714	n02408429
n02206856	n02107142	n04590129	n04228054	n01910747	n02105056	n04548362	n02356798	n04004767	n02423022
n04154565	n02098413	n04371774	n02066245	n04409515	n01773549	n02134084	n04131690	n02415577	n04442312
n02109525	n02966193	n03888257	n03933933	n07747607	n01774384	n03841143	n02708093	n04557648	n02493509
n01664065	n04456115	n01742172	n02787622	n09246464	n02965783	n03250847	n02790996	n03065424	n02095889
n03967562	n03188531	n04252077	n02109047	n04372370	n03445777	n02129604	n07768694	n01806143	n02417914
Part 8									
n02788148	n04264628	n03527444	n02115913	n02110806	n03461385	n01847000	n07730033	n03920288	n04265275
n03134739	n01882714	n04597913	n04317175	n03109150	n07802026	n02174001	n03208938	n04273569	n03814639
n01729977	n01498041	n02115641	n02536864	n03496892	n03877472	n03529860	n03447721	n04485082	n03110669
n02834397	n02437312	n03255030	n07248320	n04467665	n03902125	n03710193	n04086273	n02110063	n04141076
n02094258	n09472597	n12267677	n04482393	n02782093	n03372029	n03929855	n01744401	n02219486	n03908618
n02087394	n03874293	n02128385	n03623198	n04026417	n02096051	n03483316	n02363005	n02497673	n02097474
n02027492	n04127249	n01693334	n01494475	n02704792	n02112018	n02802426	n03633091	n04486054	n10565667
n03627232	n02108551	n02777292	n03532672	n02988304	n04476259	n02018207	n02002724	n01728572	n02124075
n04209239	n02279972	n03903868	n02089867	n03447447	n02110958	n03763968	n02092002	n01829413	n03459775
n03961711	n02101006	n01641577	n02730930	n03337140	n07749582	n04251144	n03868863	n01883070	n04147183

Table B.13: **Parts 9 ans 10:** folder names of classes sampled randomly from ImageNet ILSVRC 2015.

Part 9									
n03950228	n01860187	n03837869	n02113978	n01537544	n04019541	n02321529	n02865351	n02108915	n02112350
n04065272	n03223299	n02112137	n02783161	n04049303	n02437616	n04346328	n03929660	n07753113	n02804414
n04356056	n02093859	n04589890	n07714571	n03124170	n01824575	n02085782	n03891251	n03924679	n07734744
n04162706	n02481823	n02037110	n02095314	n02132136	n02319095	n02835271	n04532106	n02346627	n01981276
n02107312	n01817953	n02441942	n02395406	n01698640	n03899768	n03977966	n03661043	n03804744	n03297495
n02028035	n03954731	n02870880	n02100236	n04336792	n04398044	n01677366	n03743016	n03016953	n03838899
n02107574	n04370456	n07716906	n01632458	n03160309	n01580077	n07579787	n03240683	n01871265	n03100240
n04149813	n01687978	n01616318	n01530575	n04005630	n06874185	n01667114	n03770439	n04285008	n02117135
n07875152	n02978881	n02727426	n02089078	n04252225	n04599235	n04296562	n01985128	n03793489	n04447861
n01978455	n03457902	n02939185	n02177972	n02231487	n02488702	n02916936	n03045698	n01978287	n11879895
Part 10									
n01644900	n04235860	n02130308	n04067472	n02483362	n03485407	n02114712	n03394916	n02002556	n02074367
n03742115	n03947888	n01768244	n04596742	n01944390	n02804610	n02092339	n03717622	n02398521	n02397096
n03534580	n03785016	n03724870	n13037406	n02486410	n03494278	n01644373	n04209133	n02825657	n03444034
n03777568	n02113799	n01924916	n04399382	n02992529	n13044778	n02445715	n03930630	n07693725	n03706229
n07892512	n02799071	n02950826	n02793495	n07717410	n07711569	n07697313	n02484975	n04380533	n02815834
n02859443	n02749479	n02102480	n03485794	n04357314	n03478589	n07584110	n03998194	n03344393	n04505470
n02786058	n03692522	n03467068	n01819313	n02129165	n03590841	n03498962	n03271574	n03445924	n04355933
n01950731	n09288635	n03595614	n02113023	n01843065	n04462240	n03000247	n01629819	n03733281	n01704323
n02823428	n02123159	n03207941	n03218198	n02105855	n03691459	n04133789	n03958227	n01688243	n02011460
n04355338	n04041544	n03617480	n02017213	n02457408	n02526121	n02281406	n03047690	n04330267	n04612504

Bibliography

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[2] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*, 2015, pp. 448–456.

[3] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on.* Ieee, 2009, pp. 248–255.

[5] T. Salimans and D. P. Kingma, "Weight normalization: A simple reparameterization to accelerate training of deep neural networks," in *Advances in Neural Information Processing Systems*, 2016, pp. 901–909.

[6] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[7] S. Ioffe, "Batch renormalization: Towards reducing minibatch dependence in batch-normalized models," in *Advances in Neural Information Processing Systems*, 2017, pp. 1945–1953.

[8] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 815–823.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[10] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *International Conference on Learning Representations*, 2015.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *European Conference on Computer Vision*. Springer, 2016, pp. 630–645.

[12] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*. IEEE, 2017, pp. 5987–5995.

[13] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.

[14] T.-Y. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie, "Feature pyramid networks for object detection." in *CVPR*, vol. 1, no. 2, 2017, p. 3.

[15] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *Computer Vision (ICCV), 2017 IEEE International Conference on*. IEEE, 2017, pp. 2980–2988.

[16] S. Zagoruyko and N. Komodakis, "Wide residual networks," *Proceedings of the British Machine Vision Conference*, pp. 87.1–87.12, 2016.

[17] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[18] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[20] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[21] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 2010, pp. 249–256.

[22] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *Proceedings of the 3rd International Conference on Learning Representations (ICLR 2015)*, 2015.

[23] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[24] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Highway networks," *arXiv preprint arXiv:1505.00387*, 2015.

[25] K. He and J. Sun, "Convolutional neural networks at constrained time cost," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 5353–5360.

[26] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[27] V. Vanhoucke, "Learning visual representations at scale," *ICLR invited talk*, 2014.

[28] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio, "An empirical investigation of catastrophic forgetting in gradient-based neural networks," *arXiv preprint arXiv:1312.6211*, 2013.

[29] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the National Academy of Sciences*, p. 201611835, 2017.

[30] R. Caruana, "Multitask learning," in *Learning to learn.* Springer, 1998, pp. 95–133.

[31] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning.* ACM, 2008, pp. 160–167.

[32] Z. Zhang, P. Luo, C. C. Loy, and X. Tang, "Facial landmark detection by deep multi-task learning," in *European Conference on Computer Vision.* Springer, 2014, pp. 94–108.

[33] X. Liu, J. Gao, X. He, L. Deng, K. Duh, and Y.-Y. Wang, "Representation learning using multi-task deep neural networks for semantic classification and information retrieval." in *HLT-NAACL*, 2015, pp. 912–921.

[34] H. Shimodaira, "Improving predictive inference under covariate shift by weighting the log-likelihood function," *Journal of statistical planning and inference*, vol. 90, no. 2, pp. 227–244, 2000.

[35] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger, "Deep networks with stochastic depth," in *European Conference on Computer Vision.* Springer, 2016, pp. 646–661.

[36] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2818–2826.

[37] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning." in *AAAI*, 2017, pp. 4278–4284.

[38] Y. Bengio *et al.*, "Learning deep architectures for ai," *Foundations and trends® in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[39] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[40] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proceedings of the 25th international conference on Machine learning.* ACM, 2008, pp. 1096–1103.

[41] D. Erhan, P.-A. Manzagol, Y. Bengio, S. Bengio, and P. Vincent, "The difficulty of training deep architectures and the effect of unsupervised pre-training," in *Artificial Intelligence and Statistics*, 2009, pp. 153–160.

[42] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, 2011, pp. 315–323.

[43] J. Bergstra, G. Desjardins, P. Lamblin, and Y. Bengio, "Quadratic polynomials learn better image features," Technical Report 1337, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, Tech. Rep., 2009.

[44] K. Jarrett, K. Kavukcuoglu, Y. LeCun *et al.*, "What is the best multi-stage architecture for object recognition?" in *Computer Vision, 2009 IEEE 12th International Conference on.* IEEE, 2009, pp. 2146–2153.

[45] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[46] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[47] D. Warde-Farley, I. J. Goodfellow, A. Courville, and Y. Bengio, "An empirical analysis of dropout in piecewise linear networks," *arXiv preprint arXiv:1312.6197*, 2013.

[48] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision.* Springer, 2014, pp. 818–833.

[49] D. E. Rumelhart, G. E. Hinton, R. J. Williams *et al.*, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1, 1988.

[50] Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning from data.* AMLBook New York, NY, USA:, 2012, vol. 4.

[51] H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin, "Exploring strategies for training deep neural networks," *Journal of Machine Learning Research*, vol. 10, no. Jan, pp. 1–40, 2009.

[52] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998.

[53] G. E. Hinton, "Training products of experts by minimizing contrastive divergence," *Training*, vol. 14, no. 8, 2006.

[54] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio, "Contractive auto-encoders: Explicit invariance during feature extraction," in *Proceedings of the 28th international conference on machine learning (ICML-11)*, 2011, pp. 833–840.

[55] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout networks," *Proceedings of the 30th International Conference on Machine Learning*, pp. 1319–1327, 2013.

[56] M. D. Zeiler and R. Fergus, "Stochastic pooling for regularization of deep convolutional neural networks," *Proceedings of the International Conference on Learning Representation (ICLR)*, 2013.

[57] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *International Conference on Learning Representations (ICLR2014), CBLS, April 2014*, 2014.

[58] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proceedings of the 26th annual international conference on machine learning.* ACM, 2009, pp. 41–48.

[59] Y.-L. Boureau, J. Ponce, and Y. LeCun, "A theoretical analysis of feature pooling in visual recognition," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 111–118.

[60] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Advances in neural information processing systems*, 2007, pp. 153–160.

[61] Y. Bengio, Y. LeCun *et al.*, "Scaling learning algorithms towards ai," *Large-scale kernel machines*, vol. 34, no. 5, pp. 1–41, 2007.

[62] C. Poultney, S. Chopra, Y. L. Cun *et al.*, "Efficient learning of sparse representations with an energy-based model," in *Advances in neural information processing systems*, 2007, pp. 1137–1144.

[63] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4013–4021.

[64] S. Winograd, *Arithmetic complexity of computations.* Siam, 1980, vol. 33.

[65] D. Arpit, Y. Zhou, B. U. Kota, and V. Govindaraju, "Normalization propagation: A parametric technique for removing internal covariate shift in deep networks," *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, pp. 1168–1176, 2016.

[66] T. Cooijmans, N. Ballas, C. Laurent, Ç. Gülçehre, and A. Courville, "Recurrent batch normalization," *International Conference on Learning Representations*, 2017.

[67] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition," in *European conference on computer vision.* Springer, 2014, pp. 346–361.

[68] A. Bilal, A. Jourabloo, M. Ye, X. Liu, and L. Ren, "Do convolutional neural networks learn class hierarchy?" *IEEE transactions on visualization and computer graphics*, vol. 24, no. 1, pp. 152–162, 2018.

[69] Z. Yan, H. Zhang, R. Piramuthu, V. Jagadeesh, D. DeCoste, W. Di, and Y. Yu, "Hd-cnn: hierarchical deep convolutional neural networks for large scale visual recognition," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 2740–2748.

[70] J. Deng, A. C. Berg, K. Li, and L. Fei-Fei, "What does classifying more than 10,000 image categories tell us?" in *European conference on computer vision.* Springer, 2010, pp. 71–84.

[71] G. Griffin and P. Perona, "Learning and using taxonomies for fast visual categorization," in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on.* IEEE, 2008, pp. 1–8.

[72] M. Marszalek and C. Schmid, "Semantic hierarchies for visual object recognition," in *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*. IEEE, 2007, pp. 1–7.

[73] R. Caruana, "Multitask learning," *Machine learning*, vol. 28, no. 1, pp. 41–75, 1997.

[74] S. Ruder, "An overview of multi-task learning in deep neural networks," *arXiv preprint arXiv:1706.05098*, 2017.

[75] I. Misra, A. Shrivastava, A. Gupta, and M. Hebert, "Cross-stitch networks for multi-task learning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 3994–4003.

[76] M. Long, Z. Cao, J. Wang, and S. Y. Philip, "Learning multiple tasks with multilinear relationship networks," in *Advances in Neural Information Processing Systems*, 2017, pp. 1594–1603.

[77] L. Kaiser, A. N. Gomez, N. Shazeer, A. Vaswani, N. Parmar, L. Jones, and J. Uszkoreit, "One model to learn them all," *arXiv preprint arXiv:1706.05137*, 2017.

[78] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1. IEEE, 2005, pp. 886–893.

[79] S. Lazebnik, C. Schmid, and J. Ponce, "Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories," in *null*. IEEE, 2006, pp. 2169–2178.

[80] G. Csurka, C. Dance, L. Fan, J. Willamowski, and C. Bray, "Visual categorization with bags of keypoints," in *Workshop on statistical learning in computer vision, ECCV*, vol. 1, no. 1-22. Prague, 2004, pp. 1–2.

[81] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through ffts," *International Conference on Learning Representations (ICLR2014), CBLS, April 2014*, 2014.

[82] T. Highlander and A. Rodriguez, "Very efficient training of convolutional neural networks using fast fourier transform and overlap-and-add," *Proceedings of the British Machine Vision Conference (BMVC)*, pp. 160.1–160.9, 2015.

[83] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," *Proceedings of the British Machine Vision Conference*, 2014.

[84] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber *et al.*, "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies," 2001.

[85] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[86] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *arXiv preprint arXiv:1409.1259*, 2014.

[87] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734, 2014.

[88] C. Laurent, G. Pereyra, P. Brakel, Y. Zhang, and Y. Bengio, "Batch normalized recurrent neural networks," in *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on.* IEEE, 2016, pp. 2657–2661.

[89] G. A. Miller, "Wordnet: a lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[90] L. Zelnik-Manor and P. Perona, "Self-tuning spectral clustering," in *Advances in neural information processing systems*, 2005, pp. 1601–1608.

[91] M. J. Fine, D. E. Singer, B. H. Hanusa, J. R. Lave, and W. N. Kapoor, "Validation of a pneumonia prognostic index using the medisgroups comparative hospital database," *The American journal of medicine*, vol. 94, no. 2, pp. 153–159, 1993.

[92] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[93] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.

[94] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7132–7141, 2018.

[95] M. Lin, Q. Chen, and S. Yan, "Network in network," *arXiv preprint arXiv:1312.4400*, 2013.

[96] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[97] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision.* Springer, 2016, pp. 21–37.

[98] Y and Jia, "Caffe: An open source convolutional architecture for fast feature embedding," *http://caffe.berkeleyvision.org/*, 2013.

[99] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *Advances in neural information processing systems*, 2014, pp. 3320–3328.

[100] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[101] J. L. Elman, "Learning and development in neural networks: The importance of starting small," *Cognition*, vol. 48, no. 1, pp. 71–99, 1993.

[102] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *arXiv preprint arXiv:1511.07289*, 2015.

[103] F. Schilling, "The effect of batch normalization on deep convolutional neural networks," 2016.