TAMPERE UNIVERSITY OF TECHNOLOGY

Yuan Liu

INCREMENTAL LEARNING IN DEEP NEURAL NETWORKS

Master of Science Thesis

Examiners: Prof. Joni-Kristian Kämäräinen
Dr. Ke Chen

Examiners and topic approved in the Computing and Electrical Engineering Faculty Council meeting on 6th May 2015

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Program in Information Technology
YUAN LIU: Incremental Learning in Deep Neural Networks
Master of Science Thesis, 67 pages, 5 Appendix pages
June 2015
Major: Pervasive Systems
Examiners: Prof. Joni-Kristian Kämäräinen, Dr. Ke Chen
Keywords: incremental learning, deep neural networks, computer vision, machine learning

Image classification is one of the active yet challenging problems in computer vision field. With the age of big data coming, training large-scale datasets becomes a hot research topic. Most of work pay more attention to the final performance rather than efficiency during the training procedure. It is known that it takes a long time to train large-scale datasets. In the light of this, we exploit a novel incremental learning framework based on deep neural networks to improve both performance and efficiency simultaneously.

Generally, our incremental learning framework is in a manner of coarse-to-fine. The concept of our idea is to utilise the trained network parameters with low-resolution images to improve the initial values of network parameters for images with high resolution. There are two solutions to implement our idea. One is to use the networks with scaled filters. The size of filters in deep networks is extended by upscaling parameters from the previous trained network with lower-resolution images. The other is to add convolutional filters to the network. We not only extend the size of filters by scaling the weights of filters, but also increase the number of filters. The same transformed method with scaled filters can be used for the same number of filters, whereas we initialise parameters of other new added filters. Incremental learning can help neural networks to keep the learned information from coarse images network for initialising the following more detail level network, and continue to learn new information from finer images.

In conclusion, both of these two solutions can synchronously improve accuracy and efficiency. For the networks with scaled filters, the performance is not raised too much, while it can save nearly 40% training time. For the networks with added filters, the performance can be respectively increased from 10.8% to 12.1%, and from 14.1% to 17.0% for the ImageNet dataset and the Places205 dataset, and reducing about 30% training time. In the view of our results, adding new layers to deep neural networks and progressing from coarse to fine resolution is a promising direction.

PREFACE

This Master's thesis work has been conducted in the Computer Vision Group at the Department of Signal Processing in Tampere University of Technology.

First of all, I would like to express my gratitude to my supervisor, Professor Joni-Kristian Kämäräinen, for giving me a precious opportunity to work on this trendy topic which matches my interest. His patient guidance, encouragement and support throughout my study and research.

I am specially grateful to Dr. Ke Chen who gives me quite a lot of invaluable advices to my research and good suggestions for materials to read.

I also wish to thank our Computer Vision Group members who have supported me during my research. Everyone is nice and friendly. I really enjoy being in the group.

Finally, I would like to thank my family for their understanding and support during my study in Finland.

Tampere, June 1st, 2015

Yuan Liu

CONTENTS

1 INTRODUCTION 1

1.1 Motivation 1

1.2 Objectives 2

1.3 Structure of the Thesis 3

2 LITERATURE REVIEW 4

2.1 Image Classification 4

2.2 Datasets 6

2.2.1 Animals with Attributes Dataset 7

2.2.2 Fine-Grained Classification Datasets 7

2.2.3 CIFAR-10 and CIFAR-100 Datasets 8

2.2.4 Places205 Dataset 8

2.2.5 ImageNet Dataset 9

2.3 Neural Networks 10

2.3.1 Multi-Layer Perceptron (MLP) 11

2.3.2 Back-Propagation (BP) 13

2.3.3 Recurrent Neural Networks 14

2.4 Deep Neural Networks 16

2.4.1 Convolutional Neural Networks 17

2.4.2 Deep Belief Networks 20

2.4.3 Deep Recurrent Neural Networks 21

2.5 Deep Learning Toolboxes 22

2.5.1 Caffe 22

2.5.2 MatConvNet 24

2.5.3 OverFeat 24

2.6 Summary 25

3 INCREMENTAL TRAINING OF DEEP CONVOLUTIONAL NEURAL NETWORKS 26

3.1 Our Concept of Incremental Deep Learning 26

3.2 Generating ImageNet-Tiny Datasets 28

3.3 Rescaling Deep Networks for Incremental Training 30

3.4 Training Deep Networks 32

4 UPSCALING DEEP NETWORKS 35

4.1 The Networks with Scaled Filters 35

4.1.1 Convolutional Layer 35

4.1.2 Inner Product Layer 38
4.2 Adding Convolutional Filters 41
4.2.1 Initialisation of Zero 41
4.2.2 Initialisation of Random Gaussian Values 41
4.2.3 Initialisation of Flipping the Existing Filters 42
4.3 Fine-Tuning 44
4.4 Summary 45

5 EXPERIMENTS AND RESULTS **46**
5.1 Evaluation of Incremental Learning with Scaled Filters 46
5.2 Experiments on Adding and Initialising New Convolutional Filters 49
5.3 Evaluation of Incremental Learning with Added Filters 51
5.4 Summary 53

6 CONCLUSIONS **55**

REFERENCES **57**

APPENDIX A. An Example of Network Configuration File for DNN_8 **63**

APPENDIX B. An Example of Network Solver File **67**

List of Figures

1.1 The main idea of incremental learning in DNNs. 2
2.1 The pipeline of image classification. 4
2.2 A comparison of supervised learning and unsupervised learning[1]. . . . 5
2.3 Examples from the Places205 dataset. 9
2.4 Examples from the ILSVRC image classification dataset. 10
2.5 A simple MLP with one hidden layer. 12
2.6 An example of RNN with one hidden layers [2]. 15
2.7 A comparsion between shallow and deep neural networks. 16
2.8 Convolutional layer structure. 17
2.9 Pooling layer structure. 18
2.10 Fully connected layer structure. 19
2.11 A simple structure of DBN. 20
2.12 Schematic illustration of a DRNN [3]. 22
2.13 Left: layer computation and connections; Right: an example of Caffe network [4]. 23
3.1 The relation between low- and high-resolution images. 26
3.2 One procudure of incremental learning of CNNs. 27
3.3 An example of full size image from class "kit fox" uses to generate 8×8, 16×16, 32×32 tiny images. 29
3.4 Results of training I_8 with net-1, net-2, net-3 and net-4 for the ImageNet dataset. 31
3.5 Net-3 architecture. 32
3.6 Caffe log output during network initialisation. 33
3.7 Caffe log output during training. 33
3.8 Caffe log output during training and testing. 34
3.9 Caffe log output during training completion. 34
4.1 Parameters transformation in convolutional layer of DNNs. 37
4.2 Parameters transformation in inner product layers of DNNs. 39
4.3 The first convolutional layers filters by Krizhevsky et al [5]. 42
4.4 Flipping filters. 43
4.5 Caffe log output during fine-tuning. 44
5.1 Comparison of training curves of the traditional method and incremental learning with scaled filters on the ImageNet dataset. 46
5.2 Comparison of training curves of the traditional method and incremental learning with scaled filters on the ImageNet dataset. 47

5.3 Comparison of training curves of the traditional method and incremental learning with scaled filters on the Places205 dataset. 48
5.4 Results on initialising new added filters as 0, random Guassian values, random Guassian values divided by 10, 100 and 1000 on the ImageNet dataset. 49
5.5 Results on initialising new added filters as flipping horizontally, vertically, diagonally and centrally on the ImageNet dataset. 50
5.6 Results of the traditional method and incremental learning with added and scaled filters for the ImageNet dataset. 51
5.7 Results of the traditional method and incremental learning with added and scaled filters for the Places205 dataset. 52
5.8 Results of the traditional method and incremental learning method for the ImageNet dataset (top) and the Places205 dataset (bottom). 54

List of Tables

3.1 The scaling procedure of DNNs for incremental learning. 30
3.2 Summary of parameters in convolutional layer in DNN_8, DNN_{16}, and DNN_{32}. 31
4.1 Summary of initialising parameters. 41
5.1 Final accuracies of the traditional method and incremental learning with scaled filters on the ImageNet dataset. 48
5.2 Final accuracies of the traditional method and incremental learning with scaled filters on the Places205 dataset. 49
5.3 Final accuracies of the traditional method and incremental learning with added filters on the ImageNet dataset. 51
5.4 Final accuracies of the traditional method and incremental learning with added filters on the Places205 dataset. 52

ABBREVIATIONS AND SYMBOLS

ANN	Artificial Neural Network
AwA	Animals with Attributes
BP	Back-Propagation
BPTT	Back-Propagation Through Time
CNN	Convolutional Neural Network
DBF	Deep Belief Network
DNN	Deep Neural Network
DRNN	Deep Recurrent Neural Network
EKF	Extended Kalman Filtering
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
MLP	Multilayer Perceptron
RBM	Restricted Boltzmann Machines
RNN	Recurrent Neural Network
RTRL	Real-Time Recurrent Learning

α	the momentum constant
$\delta_j(n)$	the local gradient of jth node in iteration n
Δw_{ij}	the weight correction
η	the learning rate
φ	the activation function
$b_j^{(h)}$	the bias of jth node in h layer
$\boldsymbol{b}$	the vector of the bias
DNN_8	the DNN used in I_8
$\boldsymbol{h}$	the set of hidden nodes
h_i	the ith hidden node
I_8	the dataset of 8×8 images
I_{16s}	the dataset of 16×16 sample images
$\boldsymbol{K_2}$	a 2×2 filter matrix
m	the number of input signals
M_8	the trained model for I_8 using DNN_8
M_{16t}	the transformed model for I_{16} to fine-tuning
m_i^{n-1}	the ith input feature map
m_j^n	the jth output feature map
n	the number of iteration
p	the number of output nodes
q	the number of hidden nodes
t	time
$\boldsymbol{t}$	the set of target output
v	the induced local field of neuron
$w_{ij}^{(h)}$	the weight of jth node connected to i^{th} node in h layer
$\boldsymbol{W}$	the matrix of weights
x_i	the ith input
$\boldsymbol{x}$	the vector of input
y_i	the ith output
$\boldsymbol{y}$	the vector of output
$\boldsymbol{Y}$	the matrix of output
z^{-1}	unit-delay

1 INTRODUCTION

Computer vision attempts to emulate the human visual system in general, and extracts useful information from input, such as images or video sequences. This has proved a surprisingly challenging task; it has occupied thousands of intelligent and creative minds over the last four decades, and despite this we are still far from being able to build a general-purpose "seeing machine" [6]. On the basis of this, image classification is one of the essential tasks in computer vision. It deals with images whose contents are predicted using a classifier represented by the features. In this thesis, we explore a novel method for the improvement of image classification. In the following, there are the motivation and several main objectives of the thesis work, as well as the structure of the thesis.

1.1 Motivation

Thanks to new technologies, there is a rapid growth of data in the real world, and we live in an era called "the age of big data". However, many machine learning algorithms require more resources that grow faster than data. In other words, majority of methods in machine learning, especially to deal with the large-scale datasets, always require a longer time to train a model. Consequently, there is a need to develop novel learning approaches to handle large-scale training data to improve efficiency.

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) provides a large-scale dataset derived from ImageNet[1]. Commonly, deep learning is a class of machine learning techniques that exploit many layers of non-linear information processing for supervised or unsupervised feature extraction and transformation, and for pattern analysis and classification [7]. Deep learning aims at learning features from higher levels of the hierarchy formed by the composition of lower level features [8]. There is an implementation for the ILSVRC proposed by Krizhevsky et al. [5] for which every 20 iterations takes about 26.5 seconds on a machine with high-end GPU Nvidia K40 by Caffe[2] [4]. For the entire training procedure, the overall time is close to one week. Similar situation of training the Places205 dataset[3], it takes 6 days to finish $300,000$ iterations [9].

Evidently, it is quite time consuming to train a set of optimised parameters of a large deep network. Therefore, we intend to seek a new solution to improve the efficiency.

[1] http://image-net.org/
[2] http://caffe.berkeleyvision.org/
[3] http://places.csail.mit.edu/index.html

Although training a large-scale dataset needs a lot of time to be completed, we attempt to separate the entire training process into several parts and train the model in a coarse-to-fine fashion to increase the efficiency. The main motivation of this thesis is to find methods of implementing incremental learning to reduce the training time and increase the performance in a unique framework.

1.2 Objectives

The main objective of this thesis is to propose a novel method based on deep neural networks (DNNs), incremental learning, to improve the performance and efficiency of image classification. Incremental learning by using DNNs is a challenging research direction, which has not been investigated in the existing literature. Our main idea is to transform the "coarse" parameters from "tiny images network" to initial parameters in the following "larger images network" incrementally, as explained in Figure 1.1. To achieve our goal, it

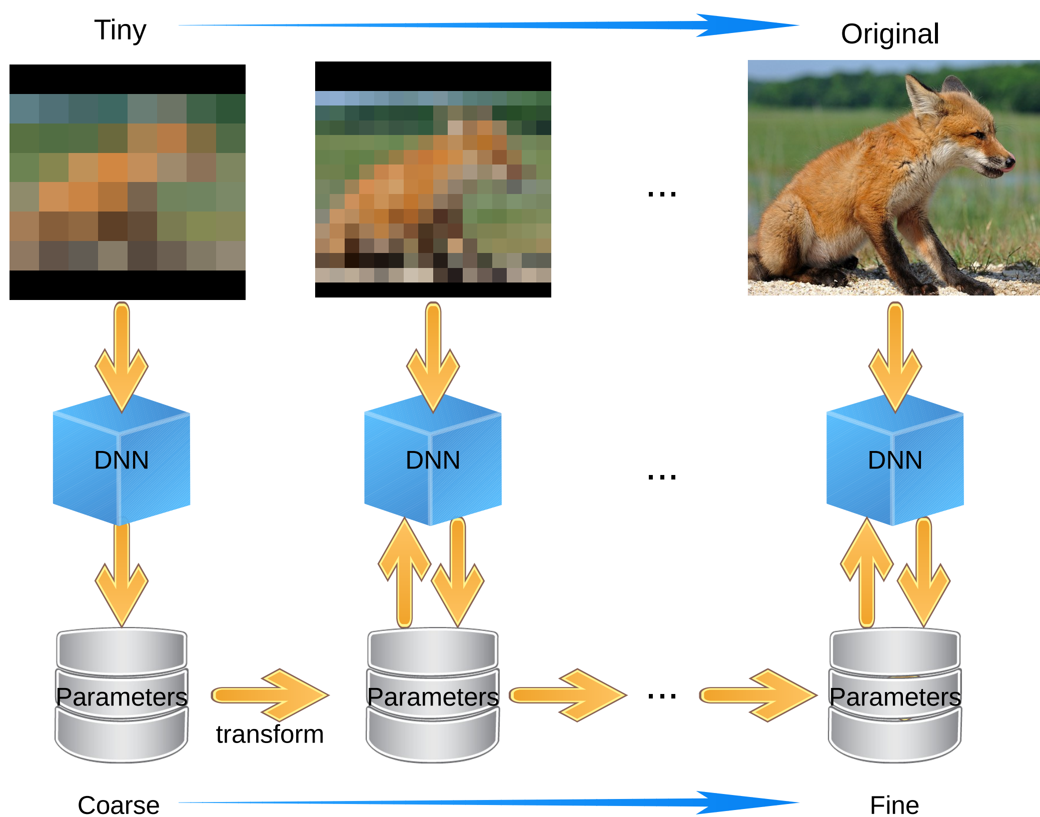

Figure 1.1. The main idea of incremental learning in DNNs.

can be divided into several questions: how to scale and add filters from tiny images to full size images network, how to transform parameters from coarse to fine through the scaled methods, and how to initialise new parameters. Based on the main objective, there are several supplementary purposes of the thesis.

The first purpose of this research is to learn neural networks along with DNNs and understand their working principles. This is the theoretical foundation, which includes learning how each layer works, and how parameters have an effect on input and output.

Moreover, the second purpose is to implement DNNs by Caffe, which is one of the popular toolboxes. Learning how to install it, use it and extract neural weights from DNNs can be useful to the experiments.

Finally, based on the theory and with using the toolbox, we need to employ them to find the relationship between the coarse and fine weights of filters, transform them, and implement them with Caffe to achieve our main objective.

1.3 Structure of the Thesis

The rest of the thesis is organised as the following. Chapter 2 describes literature reviews on image classification, several popular datasets, neural networks, DNNs and related toolboxes. Chapter 3 presents the preparatory works, which consist of the concept of incremental learning, data pre-processing, and setting network architecture before transforming. Chapter 4 provides incremental learning methods used in the thesis to improve the efficiency. Chapter 5 shows the details of the experiments. These consist of results, analysis and comparisons among different methods. Finally, the conclusion of the thesis work is given in Chapter 6.

2 LITERATURE REVIEW

This chapter starts from introducing the general task of image classification. Then, we introduce some popular datasets used in recent works. Furthermore, the basic theories of the used methods are presented, which include different types of neural networks. Finally, some trending toolboxes for training the neural networks are presented.

2.1 Image Classification

In artificial intelligence, machine learning is an important research topic, and classification is one of the most fundamental problems in this area. Image classification is the task of assigning an input images one label from a fixed set of categories according to the content of the image, such as face recognition in biometrics [10] and inspection system in product factory [11]. An algorithm that implements classification is known as a classifier, which maps the input data to a category [12]. There are lots of techniques pro-

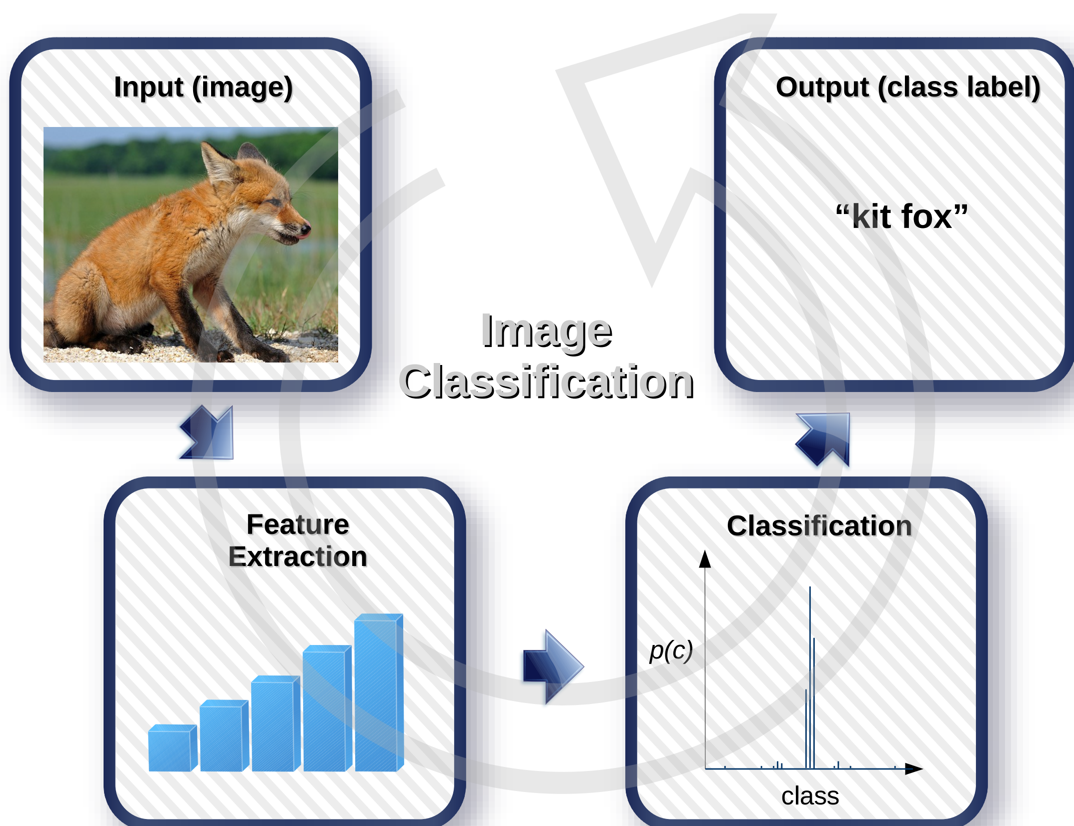

Figure 2.1. The pipeline of image classification.

posed to complete the task of image classification. The most popular pipeline of image classification is based on the assumption that the image depicts one or more features and that each of these features belongs to one of several distinct and exclusive classes [13], shown in Figure 2.1. The classes may be specified a priori by an analyst (as in supervised classification) or automatically clustered (i.e. as in unsupervised classification) into sets of prototype classes, where the analyst merely specifies the number of desired categories [13].

Image classification is a critical task for both humans and computers. It is one of the forms of data analysis that can be used to extract a model or classifier constructed to predict categorical labels [14]. A training set, a test set and a validation set are exploited for discovering and evaluating the relationships of model. For example, if the most suitable classifier is explored, the training set is used to train the candidate algorithms, the validation set is used to compare their performances and decide which one to take, and the test set is used to obtain the performance characteristics such as accuracy [15]. Generally, the techniques of image classification are made up of supervised learning and unsupervised learning, illustrated in Figure 2.2.

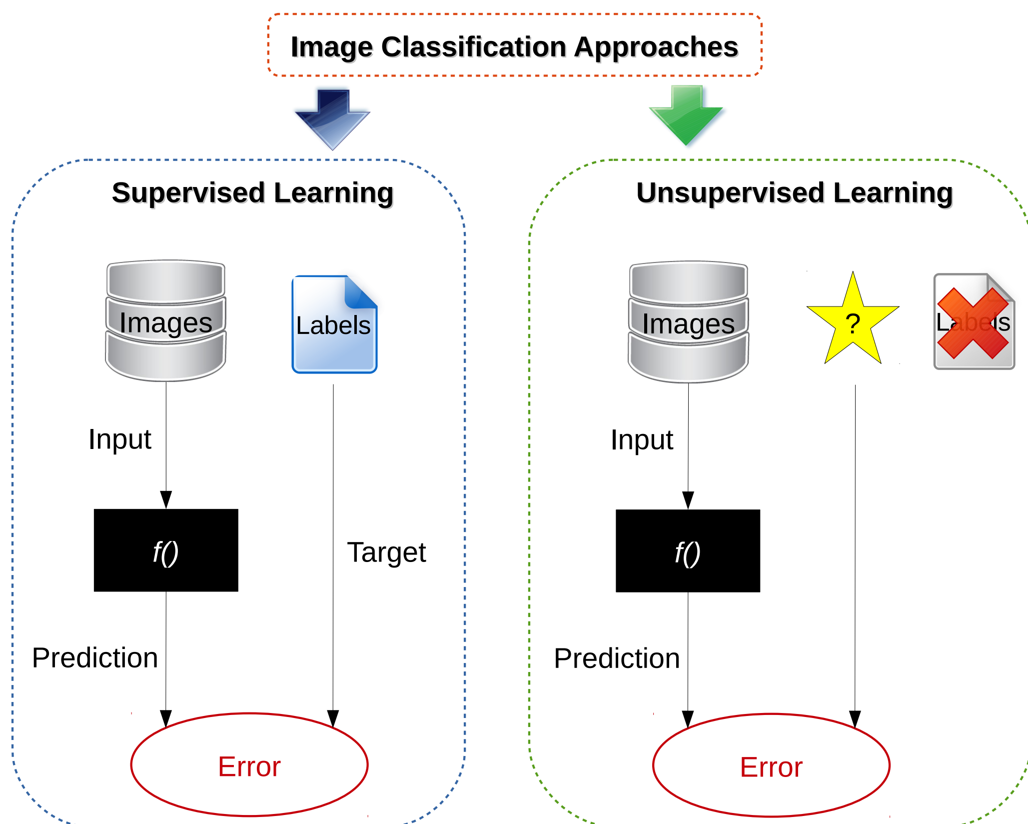

Figure 2.2. A comparison of supervised learning and unsupervised learning[1].

Supervised Learning

Conceptually, the supervised learning is considered as the knowledge of the environment, where the knowledge being denoted as a set of input-output examples. In other words, each training data is a pair of an input and its corresponding target output. During the training phase, a learning algorithm builds the classifier by analysing or "learning from" a training set made up of database and their associated class labels [14]. The desired output represents the optimum action which is performed by the classifier. The parameters of a classifier are enforced to minimise the error between predicted labels and ground truth. The knowledge of the environment available in the training set is transferred to the classifier through the training as fully as possible. When this condition is reached, we may then dispense with training set and let the classifier to deal with the environment completely by itself [2]. The experiments in the thesis are based on supervised learning for coping with image classification.

Unsupervised learning

On the contrary, an alternative solution is to utilise the data with no class label of each training set in an unsupervised learning. In the training phase, it tries to auto-associate information from the input with an intrinsic reduction of data dimensionality or total amount of input data [16] with the similarity across training samples. Once the network has become tuned to the statistical regularities of the input data, it develops the ability to form internal representations for encoding features of the input and thereby to create new classes automatically [2].

2.2 Datasets

With the development of computer vision, the increasing number of datasets is collected to solve the task of image classification. The dataset can be utilised to solve different classification problems, such as classification based on attribute, object categories, size and amount of images. There are several widely used datasets of images, and the last two are used in this thesis.

2.2.1 Animals with Attributes Dataset

The Animals with Attributes (AwA) dataset[4] [17, 18] is a dataset for attribute based classification with disjoint training and testing classes. The split of the classes is not done randomly, but much of the diversity of the animals in the dataset (water/land-based, wild/domestic, etc.) is reflected in the training as well as in the test set of classes [18]. It is aligned with all 50 Osherson [19] and Kemp [20] animal classes by querying the image search engines of Google, Microsoft, Yahoo, and Flickr, thus providing 85 numeric attribute values for each class. The collection consists of $24,295$ images of 40 classes for training, and $6,180$ images of the remaining 10 classes for testing. Each image has six pre-extracted feature representations. It is formed by combining the collected images with attribute-matrix. The dataset can be served as zero-data learning in computer vision, and it is identified based on a high-level description that is phrased in terms of semantic attributes, such as the object's colour and shape [18].

2.2.2 Fine-Grained Classification Datasets

The Fine-Grained datasets are employed for fine-grained object categorisation, which means to distinguish the breed of objects from an image, such as Chihuahua, Maltese dog, and Blenheim spaniel. The datasets include many categories of aircraft, birds, cars, dogs, and shoes. There are many datasets used for fine-grained visual categorisation including Stanford Dogs dataset [21], Oxford-IIIT-Pet dataset [22], Caltech-UCSD Birds-200-2011 [23], and Birdsnap [24]. We present two examples of Fine-Grained datasets as below.

The Stanford Dogs dataset[5] is one of the Fine-Grained datasets with 120 classes. There are $20,580$ images of dogs annotated with a bounding box and object class labels. Both of images and bounding boxes are from ImageNet [25], which we will introduce later. The classification of dogs is a difficult problem. First, there is little inter-class variation, such as different dog categories may share very similar facial characteristics but differ significantly in their color. In addition, dogs in the same class could have different ages, poses and even colours. Furthermore, a large proportion of images are leading to greater background variation [21].

[4] http://attributes.kyb.tuebingen.mpg.de/

[5] http://vision.stanford.edu/aditya86/ImageNetDogs/

The Caltech-UCSD Birds-200-2011 dataset[6] is another fine-grained categorisation used to distinguish species of birds. It contains $11,788$ images from 200 species. Each image annotated by 15 part locations, 312 binary attributes, and one bounding box. The classification of bird species is a challenging problem, because different bird species can vary dramatically in shape and appearance (e.g. pelicans vs. sparrows). At the same time, even for expert bird watchers, some pairs of bird species are nearly visually indistinguishable, such as the sparrow species. Intraclass variance is high due to variation in lighting and background and extreme variation in pose (e.g. flying birds, swimming birds, and perched birds that are partially occluded by branches) [23].

2.2.3 CIFAR-10 and CIFAR-100 Datasets

The CIFAR-10 dataset[7] [26] is composed of $60,000$ natural images of 10 classes. There are random-selected $5,000$ images from each class as training set, whereas the remaining $10,000$ images form the testing set. Each image is a 32×32 RGB image. The ten classes include airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck.

There is an extension of CIFAR-10 dataset, named CIFAR-100 dataset [26]. It has 100 classes (fine label) and 20 superclasses (coarse label). There are 500 training images and 100 testing images for each class. Each image is a 32×32 RGB image.

2.2.4 Places205 Dataset

The Places205 dataset [9] is a large-scale dataset provided by MIT Computer Science and Artificial Intelligence Laboratory. It is used as the benchmark of scene recognition methods. The total size of the dataset is around 2.5 millions images of 205 scene categories. There are randomly selected $2,448,873$ images as the training set and the remaining 100 images per category as the validation set. There are distinct $41,000$ images in the separate testing set. The Places205 dataset is designed to represent places and scenes found in the real world, examples shown in Figure 2.3.

[6] http://www.vision.caltech.edu/visipedia/CUB-200-2011.html
[7] http://www.cs.toronto.edu/~kriz/cifar.html

airfield

hospital

sauna

tree house

woodland

yard

Figure 2.3. Examples from the Places205 dataset.

2.2.5 ImageNet Dataset

The ImageNet dataset [25] is a large-scale database with millions of images from more than $20,000$ categories. It is a useful resource for image classification, and object detection.

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [27] has become the benchmark in large-scale image classification as well as object detection. The main task in this thesis is image classification, and therefore we choose the image classification dataset provided by ILSVRC as the experimental target. It is a subset of the ImageNet and collected from the Internet through several image search engines. The dataset consists of $1,000$ classes, and each image contains one ground truth label. There are 1.2 million images for training, $50,000$ images for validation, and $100,000$ images for testing. Figure 2.4 shows the example images from 5 classes of ILSVRC image classification dataset.

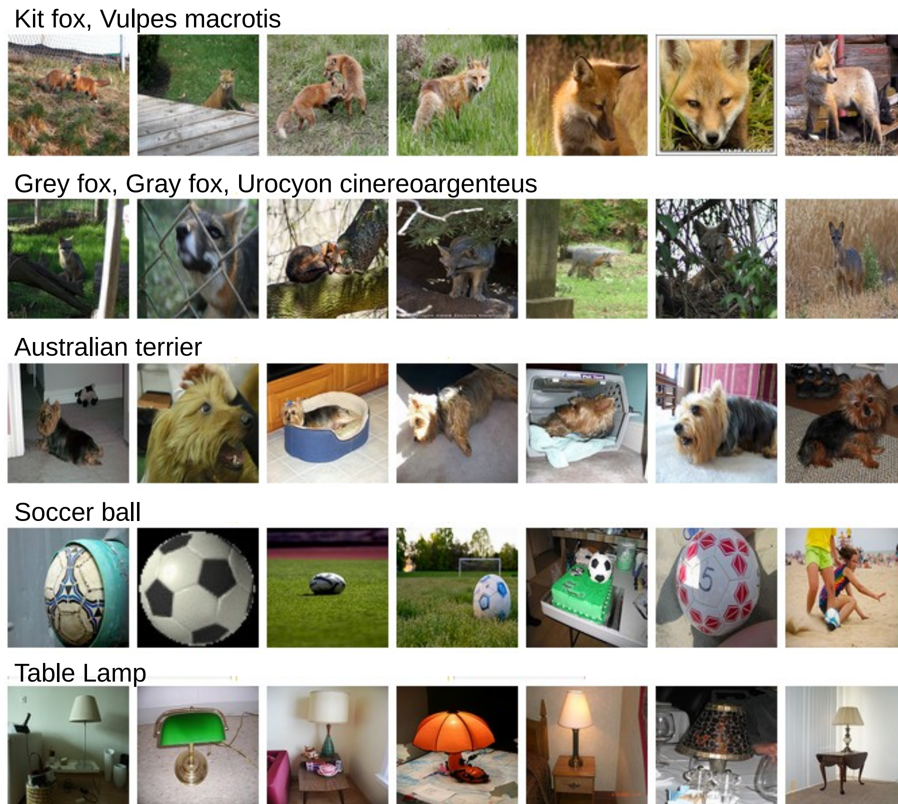

Figure 2.4. Examples from the ILSVRC image classification dataset.

2.3 Neural Networks

A neural network is a massively parallel distributed machine learning method that contains simple processing units, which has a natural propensity for storing experiential knowledge and making it available for use [28], also called artificial neural network (ANN). In other words, it has the computing power through parallel distributed structure and ability to learn. Basically, the general idea of a neural network is inspired by biological neural networks, and is presented as a combination of neurons which can calculate values from input. A neuron is defined as an information-processing unit that is fundamental to the operation of a neural network [2]. However, neural networks are motivated by the goal of obtaining highly effective machine learning algorithms, independent of whether these algorithms mirror biological processes [29].

Neural networks have variety of useful properties and capabilities, like nonlinearity, input-output mapping, adaptively and so on [2]. Their learning can be robust to errors in the training data and has been successfully applied to problems such as interpreting visual scenes [30], speech recognition [31], and learning robot control strategies [32, 29].

In the following subsections, a type of network architectures called multilayer perceptron (MLP), a common algorithm for training neural networks and recurrent neural network (RNN) are introduced.

2.3.1 Multi-Layer Perceptron (MLP)

The purpose of any supervised learning algorithms is to find a function that adequately maps between the input and output. Simple mapping can be learned by a single neuron, but it cannot learn nonlinear separable data. The MLP overcomes this limitation as it can create internal representations as layers and learn different features in each layer [33]. A generic feedforward MLP neural network can adapt to difficult training data as presented by Rumelhart et al. [34]. Generally, a MLP whose nodes are neurons with logistic activation, is a finite directed acyclic graph, which consists of an input layer with sensory units, one or more hidden layers and an output layer with computational nodes. The input signal propagates through the network to forward layer-by-layer direction. There are three main characteristics of MLPs [2]:

- Each neuron in a MLP has a nonlinear activation function. The presence of nonlinearities is important because otherwise the input-output relation of the network could be reduced to that of a single-layer perceptron [2]. Generally, the sigmoidal nonlinearity defined by logistic function, who is biologically motivated, is used to the nonlinearity, as

  $$\varphi(v) = \frac{1}{1 + exp(-v)} \tag{2.1}$$

  where v is the induced local field of neuron, and $\varphi(v)$ is the output of the neuron [35].

- A MLP contains one or more layers of hidden neurons, which are not part of the input or output. These hidden neurons enable the network to learn complex tasks by extracting progressively more meaningful features from the input patterns [2]. In principle, each hidden layer performing a transformation of a scalar product of the output of the previous layer and the vector of weights linking the previous layer to a given node [36].

- The network reveals a high degrees of connectivity, determined by the synapses of the network. A change in the connectivity of a network requires a change in the population of synaptic connections or their weights [2].

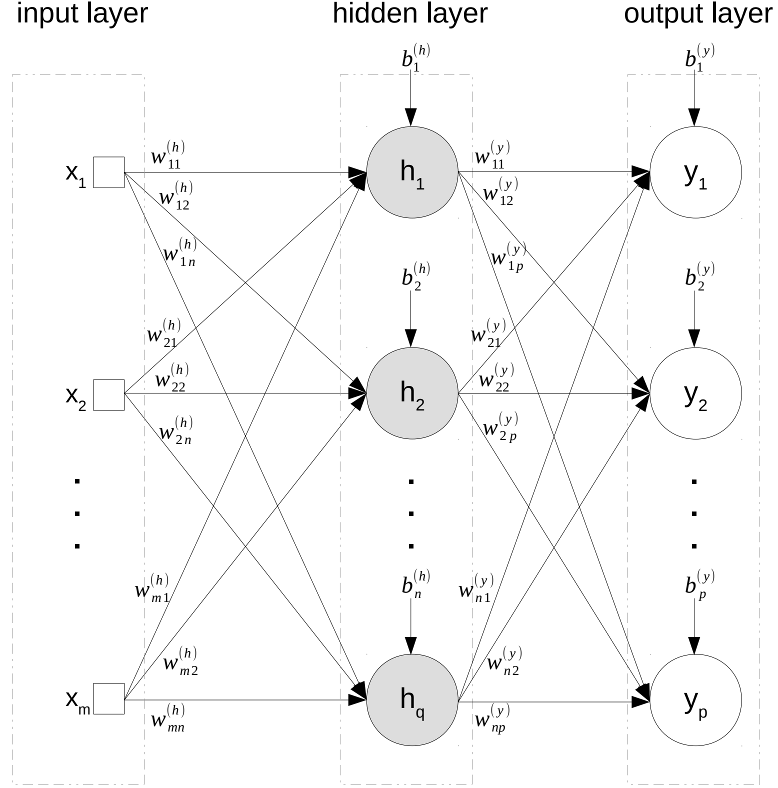

Figure 2.5. A simple MLP with one hidden layer.

A simple MLP with one hidden layer is illustrated in Figure 2.5. Here, we denote m input signals as a vector $\boldsymbol{x} = [x_1, x_2, ..., x_m]^T$, q hidden nodes as $\boldsymbol{h} = [h_1, h_2, ..., h_q]^T$, and p output nodes as $\boldsymbol{y} = [y_1, y_2, ..., y_p]^T$. The superscript is set to represent the layer of parameters, so that the output of hidden neurons h_j can be written as

$$h_j = \varphi(\sum_{i=1}^{m} w_{ij}^{(h)} x_i - b_j^{(h)}) \tag{2.2}$$

where $w_{ij}^{(h)}$ is the weight of jth node in the hidden layer h connected to the ith node from the previous layer, $b_j^{(h)}$ is the bias of jth node in the hidden layer, and φ is the activation function in each neuron. Based on the equation (2.2), the final result y_k can be written as

$$y_k = \varphi(\sum_{j=1}^{q} w_{jk}^{(y)} h_j - b_k^{(y)}) = \varphi(\sum_{j=1}^{q} w_{jk}^{(y)} \varphi(\sum_{i=1}^{m} w_{ij}^{(h)} x_i - b_j^{(h)}) - b_k^{(y)}) \tag{2.3}$$

Similarly, $w_{jk}^{(y)}$ and $b_k^{(y)}$ are presented the weight and bias located in the output layer y. Likewise, the general output of MLP with more hidden layers can be denoted as

$$y_j^{(l)} = \varphi(\sum_{i=1}^{m} w_{ij}^{(l)} y_i^{(l-1)} - b_j^{(l)}) \tag{2.4}$$

The MLP is a very flexible model, which gives good performance on a wide range of problems in discrimination and regression [36], and applies in a variety of fields. MLPs with back-propagation (BP) algorithm are the standard algorithm for numerous supervised learning tasks.

2.3.2 Back-Propagation (BP)

BP algorithm is the most popular supervised method in training MLPs. It is described as a multi-stage dynamic system optimization method by E. Bryson and Yu-Chi Ho [37, 38]. Since 1974, BP algorithm has been further developed by Paul Werbos [39], and David E. Rumelhart et al. [33, 40]. The intention and motivation for developing the BP algorithm is to find a way to train a multi-layered neural network which can learn the appropriate internal representations to allow it to learn any arbitrary mapping of input to output. The derivative of an error function can be calculated efficiently in MLPs. In other words, it is based on error-correction learning rule [2] to minimize error and adjust parameters of the network.

For further description of the algorithm, let a training vector as $\boldsymbol{x} = [x_1, x_2, ..., x_m]^T$, and its target output is $\boldsymbol{t} = [t_1, t_2, ...t_p]^T$. Normally, the BP algorithm can be divided into two passes, forward pass and backward pass [2, 29, 41]:

Forward pass

An input vector $\boldsymbol{x}$ can be applied to the network to forward propagate layer by layer, and obtain a set of output $\boldsymbol{y} = [y_1, y_2, ..., y_p]^T$ as the actual response of the network by the equation (2.4). During this pass, parameters of the network are not changed. In conclusion, forward propagation can take a training data through the network for purpose of generating the current output.

Backward pass

After forward pass, there is an error between the output $\boldsymbol{y}$ and the desired response $\boldsymbol{t}$. When neuron j is located in the output layer of the network, it is supplied with a desired response of its own. However, when neuron j is located in a hidden layer of the network, the error signal would have to be determined recursively in terms of the error signals of all the neurons to which that hidden neuron is directly connected [2]. Therefore, we can write the local gradient $\delta_j(n)$ as [2, 36]

$$\delta_j(n)^{(L)} = e_j^{(L)}(n)\varphi^{'}(v_j^{(L)}(n)) \text{ for } j \text{ in output layer L} \quad (2.5)$$

$$\delta_j^{(l)}(n) = \varphi^{'}(v_j^{(l)}(n)) \sum_k \delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n) \text{ for } j \text{ in hidden layer l} \quad (2.6)$$

In the time of backward pass, the error is propagated inversely through the network to adjust parameters according to the error-correction rule. On the basis of the delta rule [42, 43], the weight correction Δw_{ij} is applied between the neuron i and neuron j defined as

$$\Delta w_{ij}(n) = \eta \delta_j y_i + \alpha \Delta w_{ij}(n-1) \quad (2.7)$$

where η is the learning rate, δ_j is the local gradient, y_i is the input signal of neuron j, α is the momentum constant, and n is the iteration. The BP algorithm provides an "approximation" to the trajectory in weight space computed by the method of steepest descent [2]. The smaller learning rate η leads to the smaller changes to the synaptic weights in the network, while the larger η forms the unstable network. The momentum constant α can increase the rate of learning yet avoid the danger of instability [2]. It controls the feedback loop. Finally, the weight correction can be utilised to adjust weights as

$$w_{ij}(n+1) = w_{ij}(n) + \Delta w_{ij}(n) \quad (2.8)$$

In a general sense, backward propagation can acquire the difference between the output and the target of training data in order to generate the Δw of all output and hidden neurons to tune the weights of the network. The development of BP algorithm become a milestone in the neural network field. When training networks in practice, derivatives should be evaluated using BP, because this gives the greatest accuracy and numerical efficiency [35].

2.3.3 Recurrent Neural Networks

We have introduced MLPs previously whose connections do not form cycles. If the cyclical connections are allowed, a RNN can be acquired, which is a class of ANNs and con-

tains at least one feedback loop. In the light of this, it is known the simplest type of a RNN is a MLP plus extra feedback loops. In general, a MLP can only map from input to output, whereas a RNN can in principle map from the entire history of previous input to each output. The key point is that the recurrent connections allow a "memory" of previous input to persist in the network's internal state, and thereby influence the network output [44]. The ability to use internal memory for processing arbitrary input sequences makes it powerful on certain tasks such as handwriting recognition [45], and many other sequential problems.

RNNs are powerful tools which allow neural networks to handle arbitrary length sequence data. They can learn many behaviours/sequence processing tasks/algorithms/programs that are not learnable by traditional machine learning methods [46]. In Figure 2.6, we visualise an example of RNNs with only one hidden layer, which feedbacks connect to both hidden neurons and output neurons. Moreover, there is a set of unit-delay elements denoted as z^{-1} are used in feedback loops [2]. In other words, it uses feedbacks at time t as the input at time $t + 1$, where forms directed cycles between neurons in the RNN.

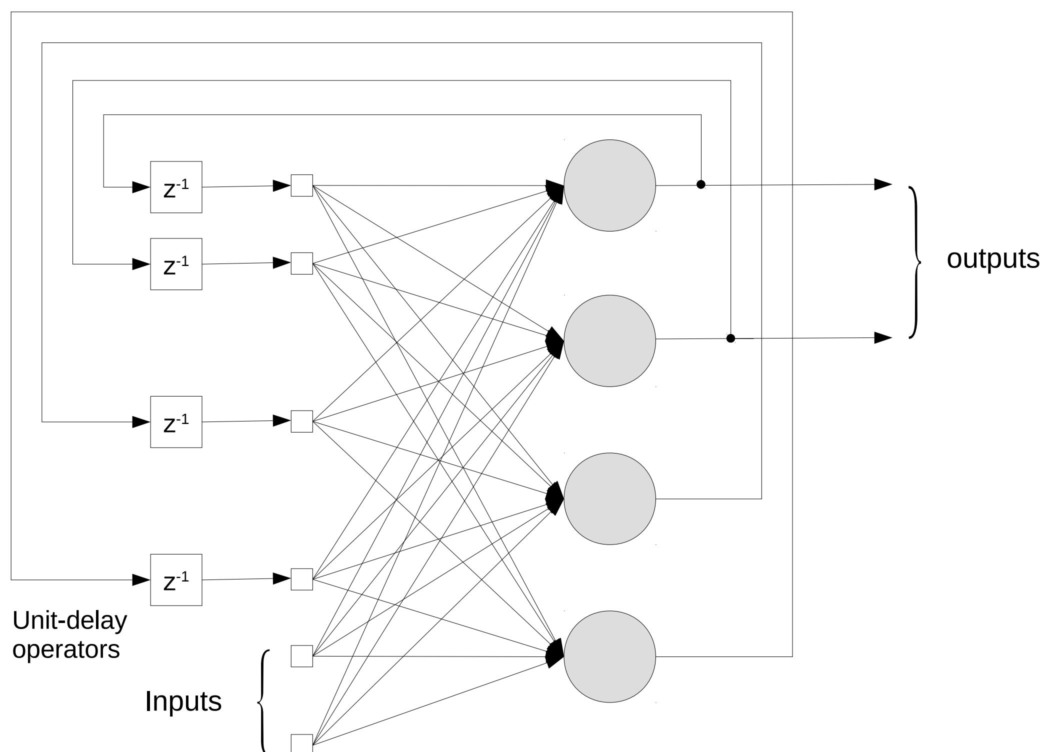

Figure 2.6. An example of RNN with one hidden layers [2].

With feedforward networks, many varieties of RNNs have been proposed, such as Elman networks, and Jordan networks [44]. RNNs are dynamical systems with temporal state representations [16] with several methods for supervised training, like back-propagation through time (BPTT), real-time recurrent learning (RTRL), and extended Kalman filtering (EKF). Furthermore, BPTT and RTRL algorithms are based on gradient descent to minimize error, while EKF is more effective on using higher-order information. EKF can converge faster than BPTT and RTRL algorithms, but requires higher computational complexity [47].

2.4 Deep Neural Networks

Background on a "deeper" network are described from the recalling literature of neural networks. The depth is defined in the case of feedforward neural networks as multiple nonlinear layers between input and output [48]. The general difference between neural networks and DNNs is illustrated in Figure 2.7. A deep architecture is one in which there

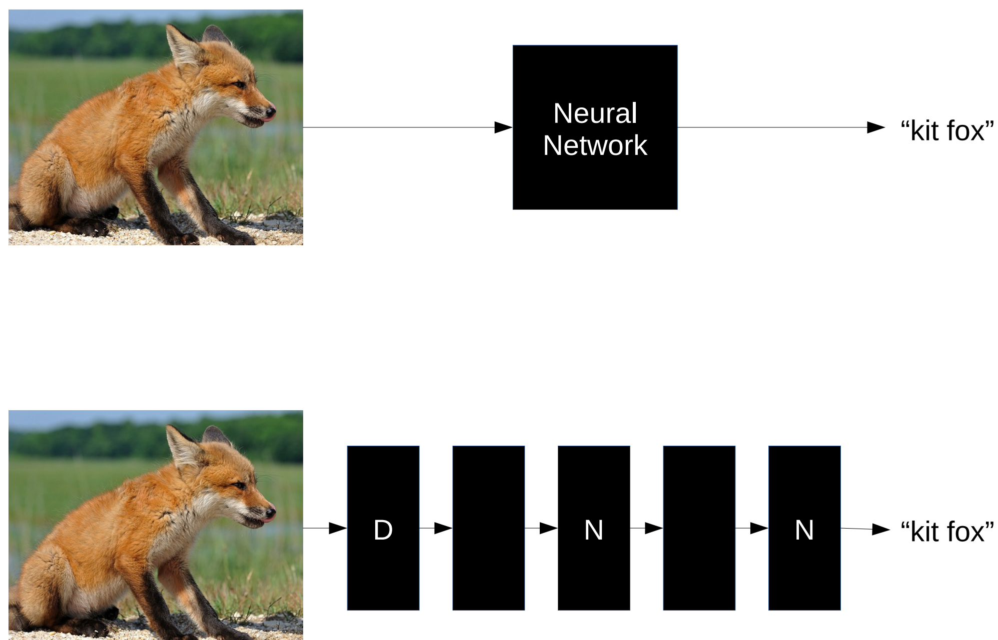

Figure 2.7. A comparsion between shallow and deep neural networks.

are multiple levels of representation, with higher levels built on top of lower levels (the lowest being the original raw input), and higher levels representing more abstract concepts defined in terms of less abstract ones from lower levels [49].

2.4.1 Convolutional Neural Networks

A convolutional neural network (CNN) is a type of feedforward neural network using convolution instead of general multiplication, inspired by biological processing and firstly introduced by Kunihiko Fukushima in 1980 [50]. A CNN consists of one or more convolutional layers and then followed by one or more fully connected layers as in one standard MLP structure. With the rise of efficient GPU computing, it is possible to train larger networks. CNNs are comprised of several layers, and we introduce the three main layers below.

Convolutional layer

In the convolutional layer, a small sub region of the image for a convolutional filter is

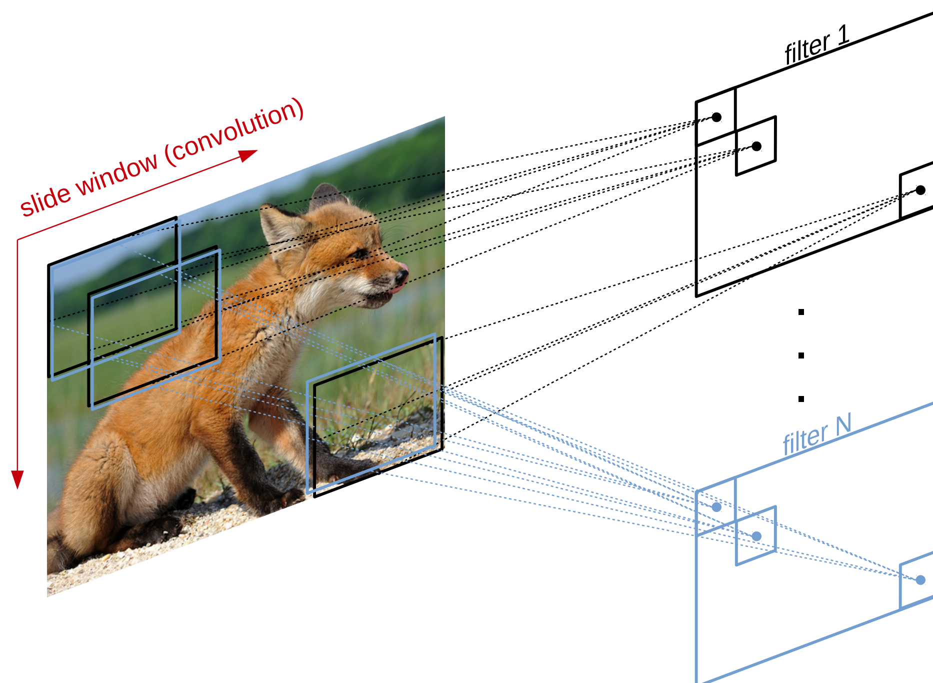

Figure 2.8. Convolutional layer structure.

applied to compute the output, also called feature map, shown in Figure 2.8. In other words, each neuron takes its synaptic input from a local receptive field in the previous layer to extract different features. Therefore, the function of the convolutional layers is to extract different features from the input, and it can be represented as [51]

$$m_j^{(n)} = max(0, \sum_{i=1}^{K} m_i^{(n-1)} w_{ij}^{(n)}) \tag{2.9}$$

where the superscript $n-1$ is the previous layer of n, m_j^n is the jth output feature map, m_i^{n-1} is the ith input feature map, and w_{ij}^n is the weight in the filter. Normally, there are many convolutional filters in each layer, and each filter shares weights through the entire image. For the first convolutional layer, it typically acquires low level features, such as edges, lines and etc. In the light of this, the more layers network has, the higher level features are obtained. Owing to the weight sharing, the evaluation of the activations of these units is equivalent to a convolution of the image pixel intensities with a filter comprising the weight parameters [35].

Pooling layer

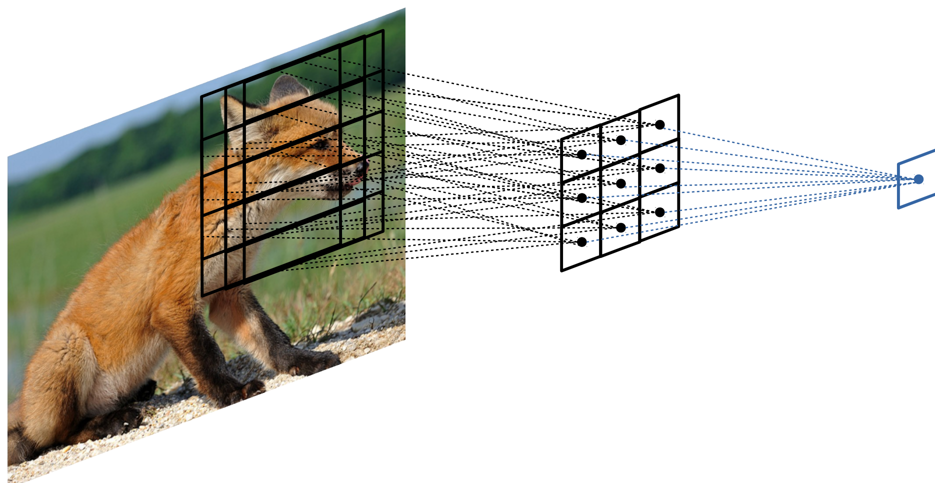

Figure 2.9. Pooling layer structure.

There can be a pooling layer after a convolutional layer, which is an important operation for object classification and detection. Pooling layers in CNNs summarise the output of neighbouring groups of neurons in the same filter map [5]. It is utilised to reduce variance by computing the mean (or max) value of a particular feature over a region of the image, visualised in Figure 2.9. These summary statistics are much lower in dimension (compared to using all of the extracted features) and can also improve results (less overfitting) [52]. The aggregation operation is called mean pooling or max pooling, which depends on the pooling operation applied. By pooling filter responses at different locations we gain robustness to the exact spatial location of features [51].

Fully connected layer

Ultimately, CNNs have one or more fully connected layers after several convolutional and pooling layers. It takes all output in the previous layer and connects to every single neuron, shown in Figure 2.10. Fully connected layers are not spatially located anymore, so there can be no convolutional layers after a fully connected layer.

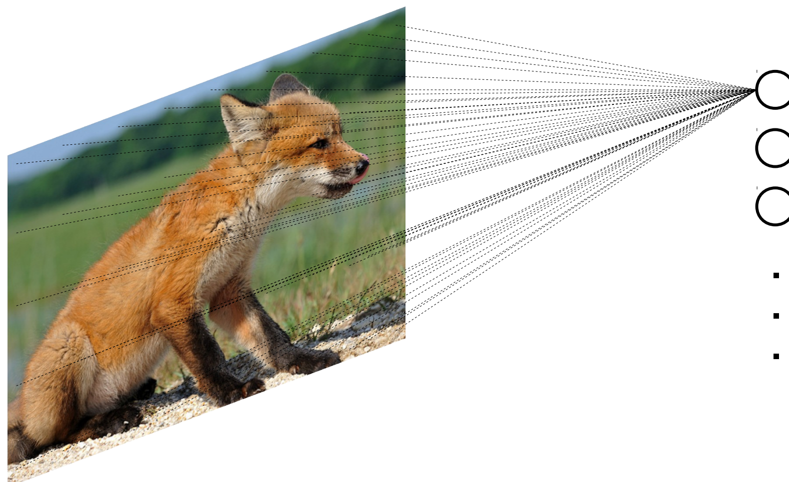

Figure 2.10. Fully connected layer structure.

In addition to the above three types of layers, CNNs also have others, like dropout layer, and loss layer [53]. Moreover, the whole network can be trained by the error minimization

using BP algorithm. The architecture is designed to take advantage of the 2D structure of an input image, which is achieved with local connections and tied weights followed by some form of pooling which results in translation invariant features [52]. Another major advantage of CNNs is the use of shared weight in convolutional layers, which means that the same filter is used for each pixel in the layer; this both reduces required memory size and improves performance [54]. On the basis of the shared weights, the number of weights in the network is smaller than if the network is fully connected. Therefore, the number of independent parameters to be learned from the data is much smaller, due to the substantial numbers of constraints on the weights [35].

2.4.2 Deep Belief Networks

A deep belief network (DBN) is a probabilistic generative model introduced by Geoffrey Hinton [55, 56], viewed as a composition of simple learning modules each of which is a Restricted Boltzmann Machine (RBM). It has been used for generating and recognizing

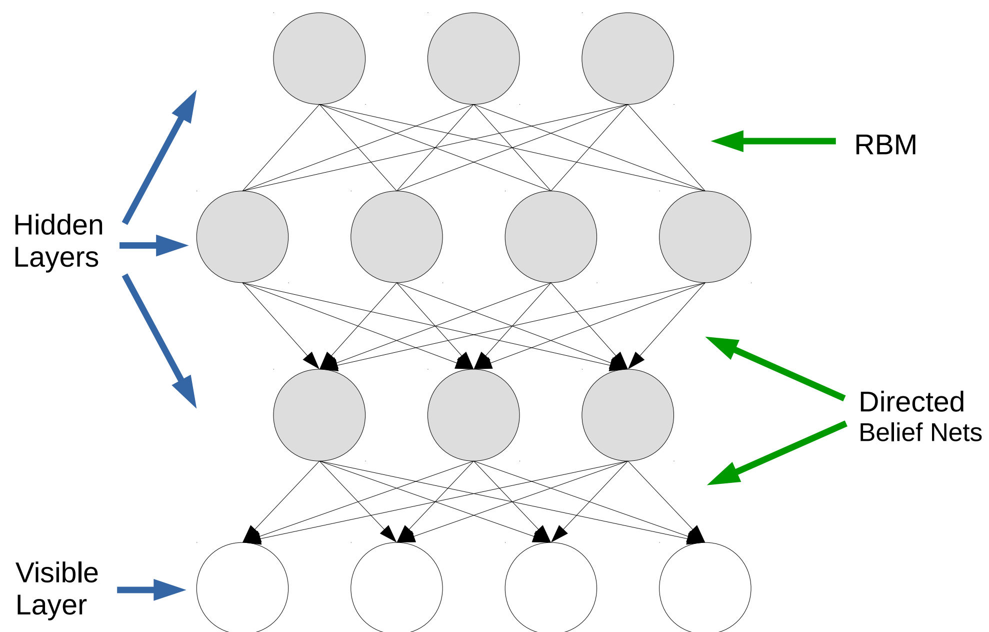

Figure 2.11. A simple structure of DBN.

images [56], and video sequences [57]. The structure of a DBN consists of multiple layers, which can be divided into visible layer and hidden layer. Furthermore, the layer of visible units represents the input data, while the layer of hidden units learns to represent features that capture higher-order correlations in the data [58].

A simple structure of DBN is shown in Figure 2.11, where the bottom layer represents data vectors and each unit in layer is a data element. The connections of the top two layers are undirected and form an associative memory, whereas other lower layers have top-down directed connections. There is a layer-by-layer procedure for learning the weights, which is determined by how the variables in one layer depend on the variables in the layer above. After learning, the values of the latent variables in every layer can be inferred by a single, bottom-up pass that starts with an observed data vector in the bottom layer and uses the generative weights in the reverse direction [58].

For the efficiency of DBNs, greedy learning can be followed by, or combined with, other learning procedures that fine-tune all of the weights to improve the generative or discriminative performance of the whole network [58]. Moreover, we can add a final layer to represent the desired output, and use BP algorithm to enforce the supervised learning. When networks with many hidden layers are applied to highly-structured input data, backpropagation works much better if the feature detectors in the hidden layers are initialised by learning a DBN that models the structure in the input data [55].

2.4.3 Deep Recurrent Neural Networks

A deep recurrent neural network (DRNN) is a combination of DNN and RNN in principle, and it can be exploited in, for example speech recognition [59], and character-level language modelling [3]. It is mentioned that RNNs are inherently deep in time, since their hidden state is a function of all previous hidden states [59]. Each network update, new information travels up the hierarchy, and temporal context is added to each layer [3]. There is a simple DRNN visualised in Figure 2.12, which shows RNNs in the hierarchy, and the previous layer sends the hidden state to the corresponding subsequent layer.

Furthermore, a DRNN also can be established by another way, like stacking multiple recurrent hidden states on top of each other. It potentially allows the hidden states at each level to operate at different timescale [3, 48]. Moreover, based on the paper of Razvan et al. [48], there are two designs provided to extend RNNs to DRNNs, which are deep transition RNN and deep output RNN respectively. Therefore, DRNNs combine the

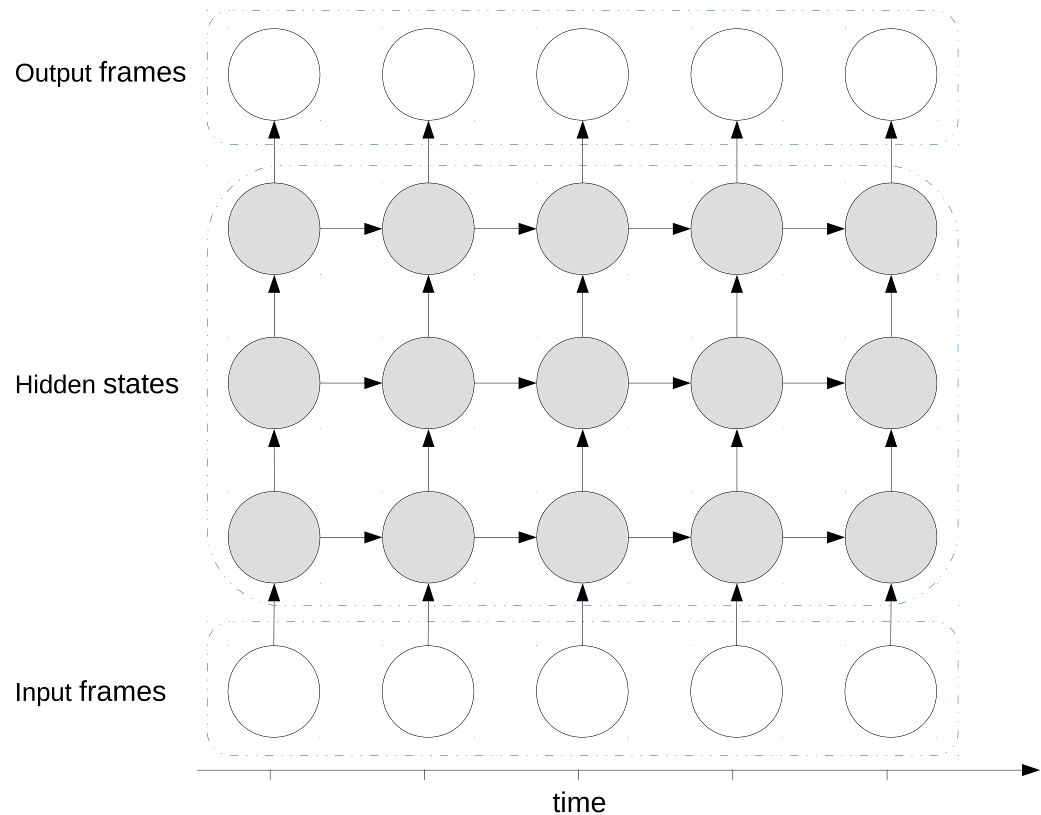

Figure 2.12. Schematic illustration of a DRNN [3].

multiple levels of representation that have proven effective in deep networks with flexible use of long range context that empowers RNNs [59].

2.5 Deep Learning Toolboxes

Deep learning toolboxes are utilised to implement the experiment with the datasets. Here we describe several widely used deep learning frameworks.

2.5.1 Caffe

Caffe [4] is a framework based on C++ library with Python and MATLAB interfaces for deep learning. It provides a complete package for training, testing, fine-tuning, and deploying models. There are several positive features in Caffe. Firstly, it is a modular open framework, easy to extend to new data formats, network layers, and loss functions.

Besides, models are defined in configuration files, which lead to separate representation and implementation. In addition, Caffe also provides pre-trained reference models for researchers.

Caffe defines a net layer-by-layer in its own model schema. The network defines the entire model bottom-to-top from input data to loss. The Caffe network consists of blobs, layers and networks. In Figure 2.13, there is the layer computation and connections in the left, as well as an example of Caffe network in the right.

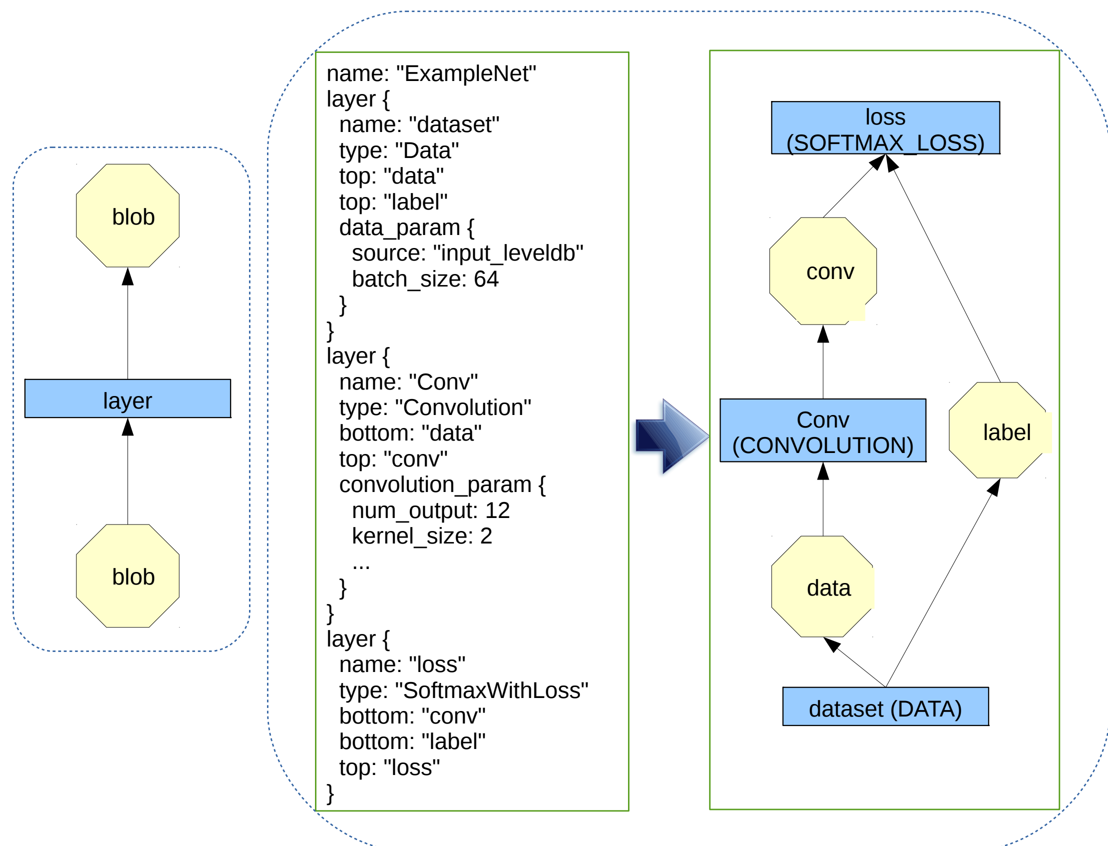

Figure 2.13. Left: layer computation and connections; Right: an example of Caffe network [4].

- Blob: a 4-dimensional array utilised for store data and parameters, which can be employed to check dimensions, read, transform and save parameters. The octagon denotes a blob in Figure 2.13.
- Layer: a neural network layer by using blobs as input and output, presented by rectangles. For operating networks, the layers take responsibilities to both forward and backward pass. Caffe provides a complete package of layers in different types to design networks in the experiments.

- Network: a set of layers connected in a computation graph.

The solver is also need to be defined, which orchestrates model optimisation by coordinating the network's forward inference and backward gradients to form parameter updates that attempt to improve the loss [4]. Firstly, it is used to scaffolds the optimization bookkeeping and creates the training network for learning and test network(s) for evaluation. Secondly, it calls network forward and backward to optimise parameters in each iteration. Moreover, it evaluates the test networks periodically. Finally, it snapshots the model and solver state throughout the optimization. The Caffe solvers are Stochastic Gradient Descent, Adaptive Gradient, and Nesterov's Accelerated Gradient [4].

2.5.2 MatConvNet

MatConvNet [60] is a simple designed MATLAB toolbox for implementing of CNNs. It includes simple functions to compute CNN building blocks, which are easy to use MATLAB functions and integrate into a complete CNN. CNN computational blocks, CNN wrappers, example applications and pre-trained models are the elements in MatConvNet. Furthermore, it is flexible to modify, extend, or combine with new networks in MATLAB code. The implementation is efficient to run the latest models such as Krizhevsky et al. [5], and supports both CPU and GPU computation. The only requirements is to work with Matlab and C/C++ compiler. MatConvNet borrows its convolution algorithms from Caffe, while is somewhat slower than Caffe.

2.5.3 OverFeat

OverFeat [61] is an integrated framework for utilising CNNs for classification, localisation and detection. It is a feature extractor, which provides powerful features for computer vision research. The C++ source codes are provided to execute the OverFeat network for recognising images and extracting features. The main idea is to use a multiscale and sliding window approach to efficiently implement within a CNN. That is meant a CNN at multiple locations and scales to predict one bounding box per class. The integrated framework is the winner of the localisation task of ILSVRC 2013 and obtained very competitive results for the detection and classification tasks.

2.6 Summary

In summary, we demonstrated the general task of image classification, datasets, methods and toolboxes related to this thesis. First, the task of computer vision, image classification, is explained. In addition, CNNs are the main method used in the thesis. Finally, Caffe is utilised to implement and run the experiments.

3 INCREMENTAL TRAINING OF DEEP CONVOLUTIONAL NEURAL NETWORKS

This chapter describes the preliminaries required to understand for the next methodology chapter. The chapter starts from explaining the concept of incremental learning in detail. Secondly, the pre-processing of the datasets with low-resolution images and the structures of DNNs are investigated. Finally, the dataset and its corresponding DNN implementation in Caffe to train the model is explained.

3.1 Our Concept of Incremental Deep Learning

With the increasing number of databases in the real world, there is a requirement to exploit better learning algorithms to handle more training data. Inspired by human's brain,

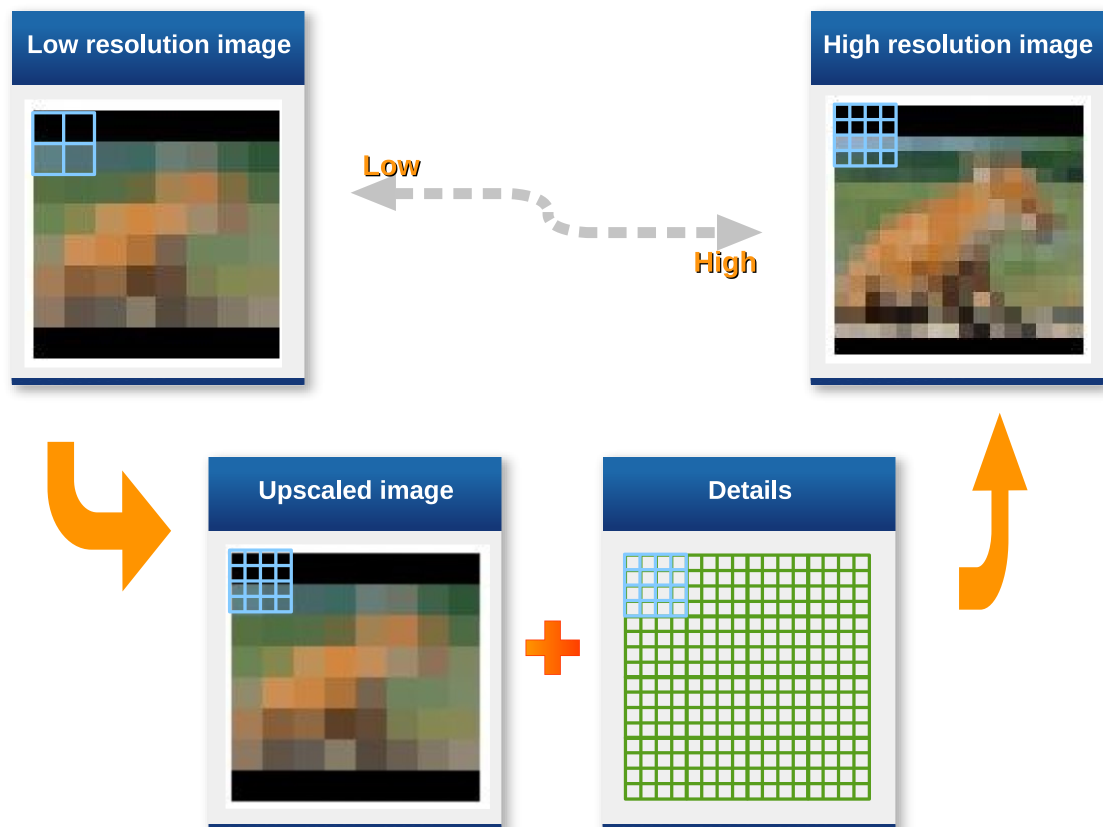

Figure 3.1. The relation between low- and high-resolution images.

incremental learning is the fastest and the most comprehensive way of learning available to people [62]. Therefore, it is also one possible solution to adopt for machine learning. Incremental learning is a machine learning paradigm where the learning process takes place whenever new example(s) emerge and adjusts what has been learned according to the previous example(s) [63].

Generally, the concept of our idea is in an coarse-to-fine manner. We utilise the trained "coarse" models from low-resolution images as initial parameters of the network for images with high resolution to improve efficiency of the entire training procedure. Compared with low-resolution images, high-resolution images can be divided into two parts, shown in Figure 3.1. The first part has the same information to the low-resolution images and learned by a tiny DNN, the second part has new detail information needed to be incrementally learned by the fine DNN from larger images. If the fine DNN needs to learn them from the start, it wastes time and memory to train the part available during low scale. Therefore, the pre-trained model for low-resolution images can be used to transform the information to the DNN trained with high-resolution images.

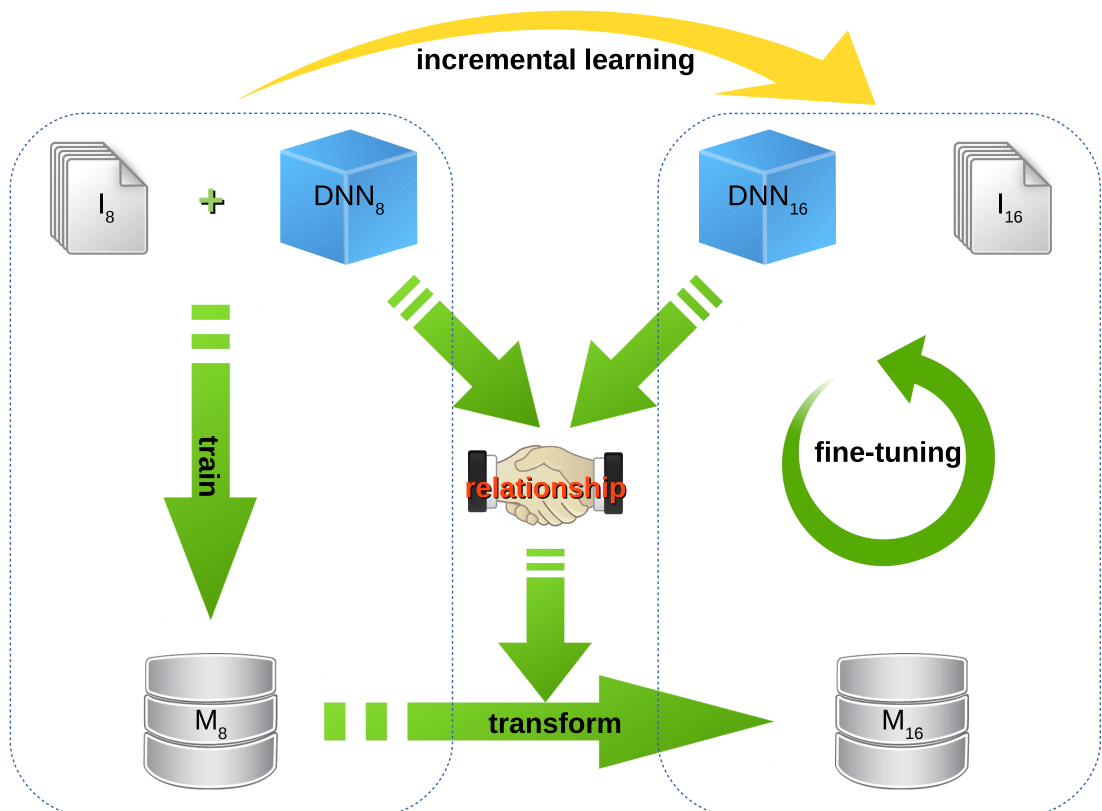

Figure 3.2. One procudure of incremental learning of CNNs.

The entire procedure of one incremental learning can be separated into several steps. Firstly, the datasets of tiny images and their corresponding tiny DNNs are constructed, e.g. 8×8 images dataset (denoted as I_8) with its DNN_8, and 16×16 images dataset (I_{16}) with its DNN_{16}. Secondly, the trained "coarse" model can be obtained by implementing the traditional method with the low-resolution images dataset and its DNN. That yields to the trained model M_8 for I_8 by using DNN_8. Similarly, the relationship between DNN_8 and DNN_{16} can be investigated, and transform the model form M_8 to M_{16t}. Eventually, a new trained model M_{16} can be further trained using the higher resolution images by fine-tuning DNN_{16} with I_{16}. The aforementioned procedure is our process of incremental learning from I_8 to I_{16}, shown in Figure 3.2. Similarly, incremental learning can be implement over and over until accomplish the final fine-tuning of the full size images, and obtain the final performance.

In the following section, the specific steps of incremental learning method are presented. The core part of our idea is described in Chapter 4. The ILSVRC image classification dataset is utilised in the experiments. For the Places205 dataset, the same networks and methods can be executed, and we show its results in the Chapter 5.

3.2 Generating ImageNet-Tiny Datasets

As described in Section 2.2.5, the ILSVRC image classification dataset is a large-scale dataset, which contains 1.2 million images for training, and $50,000$ images for testing. These images belong to $1,000$ object classes, and each image corresponds to one ground truth class label. For implementing the idea of incremental learning, "tiny" datasets of low-resolution images are created by downscaling the full size images, and fed to the DNN implemented in Caffe.

Firstly, the images conversion provides by the ImageMagick tool[8] in Linux operating system. The *convert* command can be executed as

```
$ convert <INPUT>.JPEG -resize (weight)x(height) <OUTPUT>.bmp
```

which resizes the images to size weight×height. The parameters of weight and height are set to desired values to obtain the low-resolution images. Besides *resize* option, there are

[8] `http://www.imagemagick.org/script/index.php`

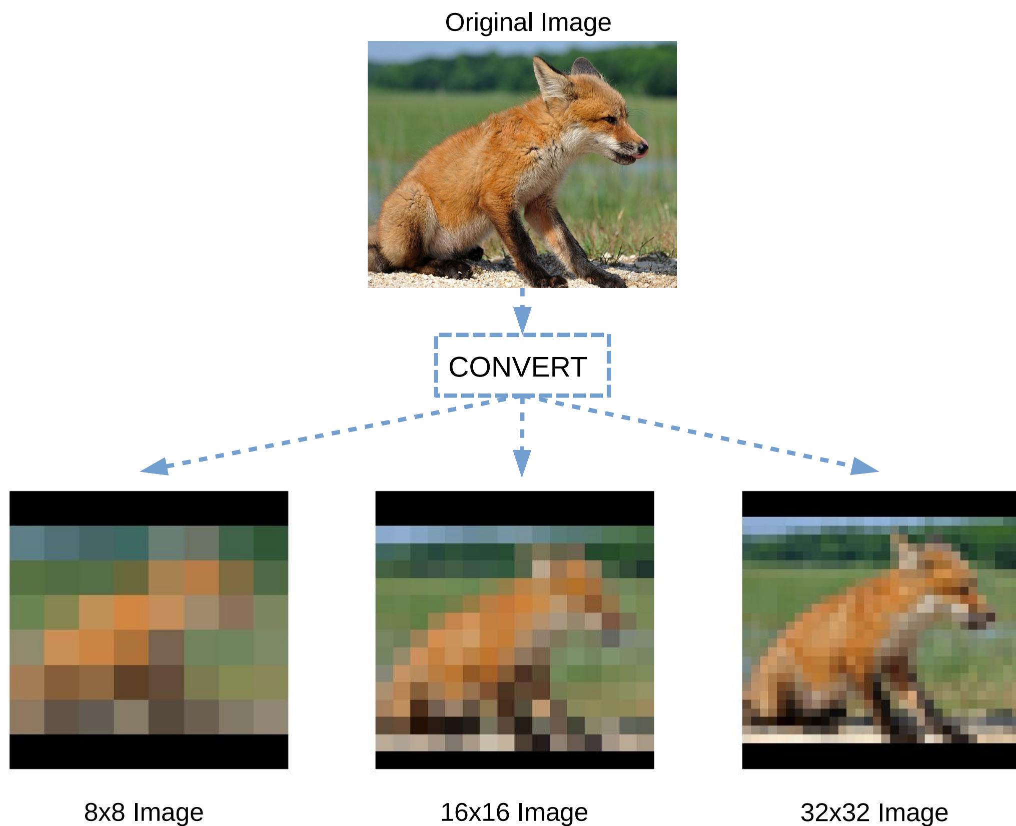

Figure 3.3. An example of full size image from class "kit fox" uses to generate 8×8, 16×16, 32×32 tiny images.

also other useful options, such as *extend*, *colorspace* and etc. The 8×8, 16×16, and 32×32 images are generated for the experiments, and the example is shown in Figure 3.3.

After that, the set of images are required to convert into a lmdb/leveldb database, which is an efficient way to enter Caffe networks. Caffe provides a program for the conversion, executed as [4]

```
$ convert_imageset [FLAGS] ROOTFOLDER/ LISTFILE DB_NAME
```

Ultimately, the Caffe compatible datasets of low-resolution images are obtained, denoted as I_8, I_{16}, and I_{32}, which are used in the following experiments.

3.3 Rescaling Deep Networks for Incremental Training

Our testbench is the Caffe deep learning architecture with a network configuration equivalent to that proposed by Krizhevsky et al [5]. The scaling procedure and the actual networks for each size of images are described in Table 3.1.

Table 3.1. The scaling procedure of DNNs for incremental learning.

	Network parameters				
	$L1(conv)$	$L2(conv)$	$L3(conv)$	$L6(full)$	$L8(full)$
orig [5]	$96(11)P3/2$	$256(5)P3/2$	$384(3)P$-/-	4096	1000
2^5-scaled (8×8)	$3(0.3)P0/0$	$8(1)P1/1$	$12(3)P$-/-	128	1000
2^4-scaled (16×16)	$6(0.7)P1/0$	$16(3)P2/1$	$24(3)P$-/-	256	1000
2^3-scaled (32×32)	$12(1.3)P2/1$	$32(4)P3/2$	$48(3)P$-/-	512	1000
net-1	$3(2)P1/1$	-	-	128	1000
net-2	$6(2)P1/1$	-	-	256	1000
net-3	$12(2)P1/1$	-	-	512	1000
net-4	$6(2)P2/1$	$16(3)P3/2$	-	256	1000

In the table, the network structure corresponds to Krizhevsky et al. [5] and $L\#$ $(L1$-$L8)$ refers to the corresponding layer. Layers not mentioned were switched off in our experiments since their spatial extent would be less than 1 pixel. Original and correspondingly scaled ($256/2^5 = 8$) numbers are given for comparison to the selected structures. The numbers in brackets for the convolutional layers corresponds to the size of the filters ($N \times N$). Filter sizes ≤ 1 were not tested since they are not filters. Pooling was disabled (NP: No Pooling) if its spatial size would be less than one pixel. The pooling setting $P1/1$ corresponds to no-pooling.

The results of training I_8 corresponded to the networks in Table 3.1 are shown in Figure 3.4. We choose the net-3 as the original network to implement our following experiments, because it has the highest accuracy. The architecture of net-3 contains one convolutional layer ($conv1$) and two fully connected layers ($fc6$ and $fc8$), shown in Figure 3.5. This network is with the low-resolution images, and provides the initial network to extend to the networks corresponding the high-resolution images.

With the increasing resolution of images, there are new details need to be learned by DNNs. In the light of this, DNNs for high-resolution images need a larger size filters or more filters than the network of low-resolution images. Thus, there are two kinds of

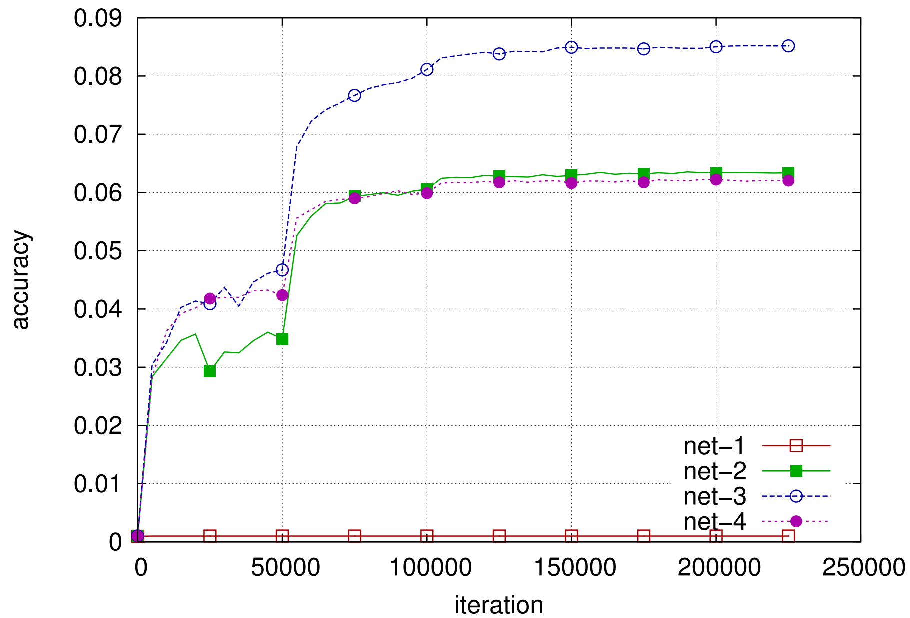

Figure 3.4. Results of training I_8 with net-1, net-2, net-3 and net-4 for the ImageNet dataset.

network rescaling steps designed for the experiments, the networks with scaled filters and the networks with added filters. The number of filters and the size of filters in the convolutional layer are shown in Table 3.2 which are readily utilised in DNN_8, DNN_{16}, and DNN_{32} .

Table 3.2. Summary of parameters in convolutional layer in DNN_8, DNN_{16}, and DNN_{32}.

	Filters number (size)		
	DNN_8	DNN_{16}	DNN_{32}
Scaled	$12(2)$	$12(4)$	$12(8)$
Added	$12(2)$	$16(4)$	$24(8)$

- The scaled filters: The size of convolutional filters is extended to higher resolution, while the number of filters is kept the same (12). During the experiments, the filter of $2{\times}2$ (denoted as K_2) is applied to I_8, extend to K_4 for I_{16}, and extend to K_8 for I_{32}.
- The added filters: Besides the size of filters, the number of filters is changed from 12 in I_8, to 16 in I_{16}, and to 24 in I_{32}.

For setting these parameters of DNNs, Caffe provides an easy configuration file to set

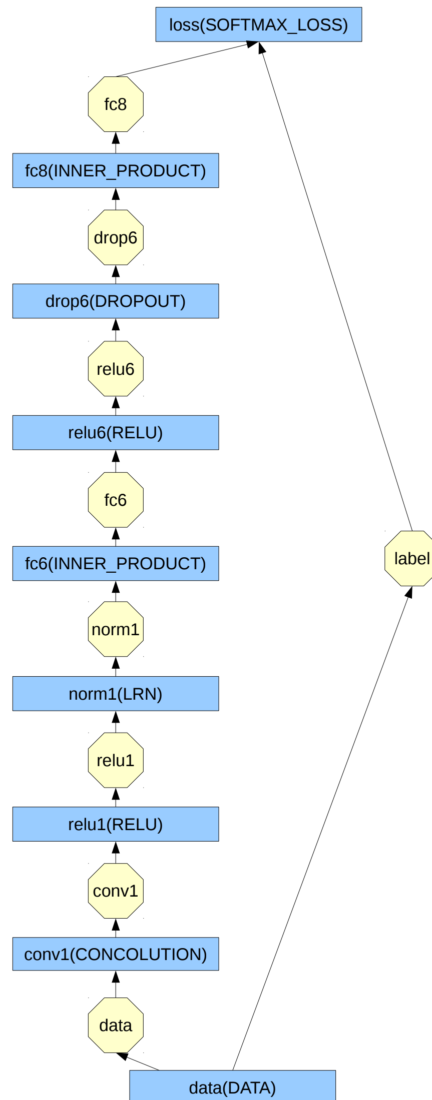

Figure 3.5. Net-3 architecture.

them, where networks are defined by plain text schemas [4]. An example of DNN_8 configuration files is given in Appendix A for the experiments.

3.4 Training Deep Networks

Based on generating ImageNet-tiny datasets and rescaling deep networks, the datasets and networks were prepared for the experiments. The last step is to train the "coarse" model of low-resolution images as the initial values for the next "fine" network. The experiments start from the dataset of lowest resolution images I_8 and its corresponded

networks DNN_8 to obtain the model M_8. Before the training phase begins, a solver is defined in Appendix B. In addition, the model requires us to subtract the image mean from each image [4, 5], which can be computed by the program provided in Caffe. After completed the pre-defined files, the network can be trained to obtain the model.

Caffe provides three interfaces for users [4], which are the command line, python and MATLAB. The command line interface was chosen for our experiments. For training a model, the *caffe train* command is used as

```
$ caffe train -solver path/to/solver.prototxt
```

When the code is executed, it provides a lot of details of briefly demonstrated in Figure 3.6, showing details such connections among layers and the shape of blobs. For example, "conv1 -> data" is the connection between data layer and conv1 layer, and "(256, 12, 7, 7)" is the shape of conv1 blob.

```
I0325 09:47:19.580461 31004 net.cpp:67] Creating Layer conv1
I0325 09:47:19.580466 31004 net.cpp:394] conv1 <- data
I0325 09:47:19.580478 31004 net.cpp:356] conv1 -> conv1
I0325 09:47:19.580488 31004 net.cpp:96] Setting up conv1
I0325 09:47:19.580858 31004 net.cpp:103] Top shape: 256 12 7 7 (150528)
```

Figure 3.6. Caffe log output during network initialisation.

After initialisation, the training phase begins from iteration 0 as displayed in Figure 3.7.

```
I0325 09:47:19.657320 31004 net.cpp:219] Network initialization done.
I0325 09:47:19.657325 31004 net.cpp:220] Memory required for data: 1299008
I0325 09:47:19.657382 31004 solver.cpp:41] Solver scaffolding done.
I0325 09:47:19.657399 31004 solver.cpp:160] Solving CaffeNet
I0325 09:47:19.657407 31004 solver.cpp:161] Learning Rate Policy: step
I0325 09:47:19.657426 31004 solver.cpp:264] Iteration 0, Testing net (#0)
I0325 09:47:22.079560 31004 solver.cpp:315]     Test net output #0: accuracy = 0.00112
I0325 09:47:22.079596 31004 solver.cpp:315]     Test net output #1: loss = 6.93571 (* 1 = 6.93571 loss)
I0325 09:47:22.098292 31004 solver.cpp:209] Iteration 0, loss = 6.96574
I0325 09:47:22.098330 31004 solver.cpp:224]     Train net output #0: loss = 6.96574 (* 1 = 6.96574 loss)
I0325 09:47:22.098342 31004 solver.cpp:445] Iteration 0, lr = 0.01
I0325 09:47:22.402977 31004 solver.cpp:209] Iteration 20, loss = 6.96712
I0325 09:47:22.403008 31004 solver.cpp:224]     Train net output #0: loss = 6.96712 (* 1 = 6.96712 loss)
I0325 09:47:22.403017 31004 solver.cpp:445] Iteration 20, lr = 0.01
```

Figure 3.7. Caffe log output during training.

For each training iteration, lr is the learning rate of that iteration, and loss is the training function defined in the solver. For the output of the testing phase, score 0 is the accuracy,

and score 1 is the testing loss function, shown in Figure 3.8.

```
I0325 09:47:37.036432 31004 solver.cpp:209] Iteration 980, loss = 6.33484
I0325 09:47:37.036463 31004 solver.cpp:224]     Train net output #0: loss = 6.33484 (* 1 = 6.33484 loss)
I0325 09:47:37.036469 31004 solver.cpp:445] Iteration 980, lr = 0.01
I0325 09:47:37.326050 31004 solver.cpp:264] Iteration 1000, Testing net (#0)
I0325 09:47:39.895401 31004 solver.cpp:315]     Test net output #0: accuracy = 0.0214201
I0325 09:47:39.895429 31004 solver.cpp:315]     Test net output #1: loss = 6.28455 (* 1 = 6.28455 loss)
I0325 09:47:39.904927 31004 solver.cpp:209] Iteration 1000, loss = 6.42154
I0325 09:47:39.904945 31004 solver.cpp:224]     Train net output #0: loss = 6.42154 (* 1 = 6.42154 loss)
I0325 09:47:39.904953 31004 solver.cpp:445] Iteration 1000, lr = 0.01
I0325 09:47:40.210124 31004 solver.cpp:209] Iteration 1020, loss = 6.35022
I0325 09:47:40.210155 31004 solver.cpp:224]     Train net output #0: loss = 6.35022 (* 1 = 6.35022 loss)
I0325 09:47:40.210181 31004 solver.cpp:445] Iteration 1020, lr = 0.01
```

Figure 3.8. Caffe log output during training and testing.

After training, the final result can be obtained as "Test net output #0: accuracy = 0.0835801", and the trained model M_8 will be saved, like "Snapshotting to ./TEMPWORK/caffe_ImageNet_tiny-8×8-sRGB-net-3_iter_120000.caffemodel", visualised in Figure 3.9.

```
I0325 10:23:03.249635 31004 solver.cpp:209] Iteration 119980, loss = 5.40909
I0325 10:23:03.249668 31004 solver.cpp:224]     Train net output #0: loss = 5.40909 (* 1 = 5.40909 loss)
I0325 10:23:03.249676 31004 solver.cpp:445] Iteration 119980, lr = 0.0001
I0325 10:23:03.548398 31004 solver.cpp:334] Snapshotting to ./TEMPWORK/caffe_ImageNet-tiny-8x8-sRGB-net-3_iter_120000.caffemodel
I0325 10:23:03.554522 31004 solver.cpp:342] Snapshotting solver state to ./TEMPWORK/caffe_ImageNet-tiny-8x8-sRGB-
net-3_iter_120000.solverstate
I0325 10:23:03.566525 31004 solver.cpp:246] Iteration 120000, loss = 5.34311
I0325 10:23:03.566553 31004 solver.cpp:264] Iteration 120000, Testing net (#0)
I0325 10:23:06.134392 31004 solver.cpp:315]     Test net output #0: accuracy = 0.0835801
I0325 10:23:06.134421 31004 solver.cpp:315]     Test net output #1: loss = 5.41878 (* 1 = 5.41878 loss)
I0325 10:23:06.134428 31004 solver.cpp:251] Optimization Done.
I0325 10:23:06.134433 31004 caffe.cpp:121] Optimization Done.
```

Figure 3.9. Caffe log output during training completion.

This procedure is repeated for all DNN_x networks, but the parameter transformation from the lower to higher scale is explained in the next section.

4 UPSCALING DEEP NETWORKS

To upscale DNN, we must transfer coarse DNN parameters to fine DNN before training. In this chapter, parameter transformation for incremental learning is proposed and analysed. The first two parts cover the transformation of parameters in different layers of upscaled networks described in Chapter 3. Furthermore, several methods to initialise parameters of new added filters in DNNs are presented. Finally, the function of fine-tuning provided by Caffe can be utilised to verify new models with transformed parameters.

4.1 The Networks with Scaled Filters

The networks with scaled filters need to extend the size of filters. According to Table 3.2, K_2 is applied to DNN_8, K_4 to DNN_{16}, and K_8 to DNN_{32} in the experiments. The upscaling does not only affect to the filters, but also increases the dimension of the output, which is used as the input to the next layer. The parameters (output weights) must be initialised in both filters and the next layer connections. In this section, the transformation of parameters in convolutional layer and its next layer, inner product layer, are discussed respectively.

4.1.1 Convolutional Layer

Conceptually, the output feature maps can be computed by a convolution from the input images with a linear filter, adding a bias, and through a nonlinear function. If we assume the kth output feature map as $\boldsymbol{Y^k}$, the convolutional filter is defined by the weights $\boldsymbol{W^k}$ and the bias b^k, which can be written as,

$$y_{ij}^k = \varphi((\boldsymbol{W^k} * x)_{ij} - b^k) \tag{4.1}$$

where φ is the nonlinear function, and y_{ij}^k is the (i, j) value of $\boldsymbol{Y^k}$. To simplify the problem, only the kth output feature map is discussed to exploit a solution. The label k is omitted, and a numeric label is added to distinguish the different size of images.

Firstly, in the convolutional layer of DNN_8, a 8×8 matrix $\boldsymbol{X^{(8)}}$ is the input image, a matrix $\boldsymbol{Y^{(8)}}$ is the output feature map, a matrix $\boldsymbol{W^{(8)}}$ of 2×2 and $b^{(8)}$ are the weights

and the bias of the filter. Based on the equation (4.1), the output can be written as

$$\begin{aligned} y_{11}^{(8)} &= \varphi((\boldsymbol{W^{(8)}} * x)_{11} - b^{(8)}) \\ y_{12}^{(8)} &= \varphi((\boldsymbol{W^{(8)}} * x)_{12} - b^{(8)}) \\ &\dots \end{aligned} \tag{4.2}$$

In addition, the same method can be applied to obtain the output of the convolutional layer of DNN_{16}, as

$$\begin{aligned} y_{11}^{(16)} &= \varphi((\boldsymbol{W^{(16)}} * x)_{11} - b^{(16)}) \\ y_{12}^{(16)} &= \varphi((\boldsymbol{W^{(16)}} * x)_{12} - b^{(16)}) \\ y_{13}^{(16)} &= \varphi((\boldsymbol{W^{(16)}} * x)_{13} - b^{(16)}) \\ &\dots \end{aligned} \tag{4.3}$$

In the pre-trained model, the parameters of $\boldsymbol{W^{(8)}}$ and $b^{(8)}$ can be extracted to tentatively utilise in $\boldsymbol{W^{(16)}}$ and $b^{(16)}$, and the output of DNN_{16} should be kept the same. Note that, in our experiments, the dimension of $\boldsymbol{W^{(16)}}$ is twice as $\boldsymbol{W^{(8)}}$, and the bias are the same.

For the bias, the same value can be used in DNN_{16}. The reason is that they have the same effect to the output, which is subtracted from the result of convolution. Based on the equations (4.2) and (4.3), the bias can be simply copied as

$$b^{(16)} = b^{(8)} \tag{4.4}$$

Nevertheless, for transforming the weights, $\boldsymbol{W^{(8)}}$ and $\boldsymbol{W^{(16)}}$ have different dimensions, how to extend the weights becomes the main concern. Here the sample images is introduced to analysis the problem. I_{16} sample images are converted from I_8 by using the method of pixel replication, which extends each pixel into four pixels with the same value, denoted as I_{16s}, shown in Figure 4.1. Thus, the relationship between I_8 and I_{16s} is

$$\begin{aligned} & x_{11}^{(16s)} = x_{12}^{(16s)} = x_{21}^{(16s)} = x_{22}^{(16s)} = x_{11}^{(8)} \\ & x_{13}^{(16s)} = x_{14}^{(16s)} = x_{23}^{(16s)} = x_{24}^{(16s)} = x_{12}^{(8)} \\ & \dots \\ & x_{31}^{(16s)} = x_{32}^{(16s)} = x_{41}^{(16s)} = x_{42}^{(16s)} = x_{21}^{(8)} \\ & \dots \end{aligned} \tag{4.5}$$

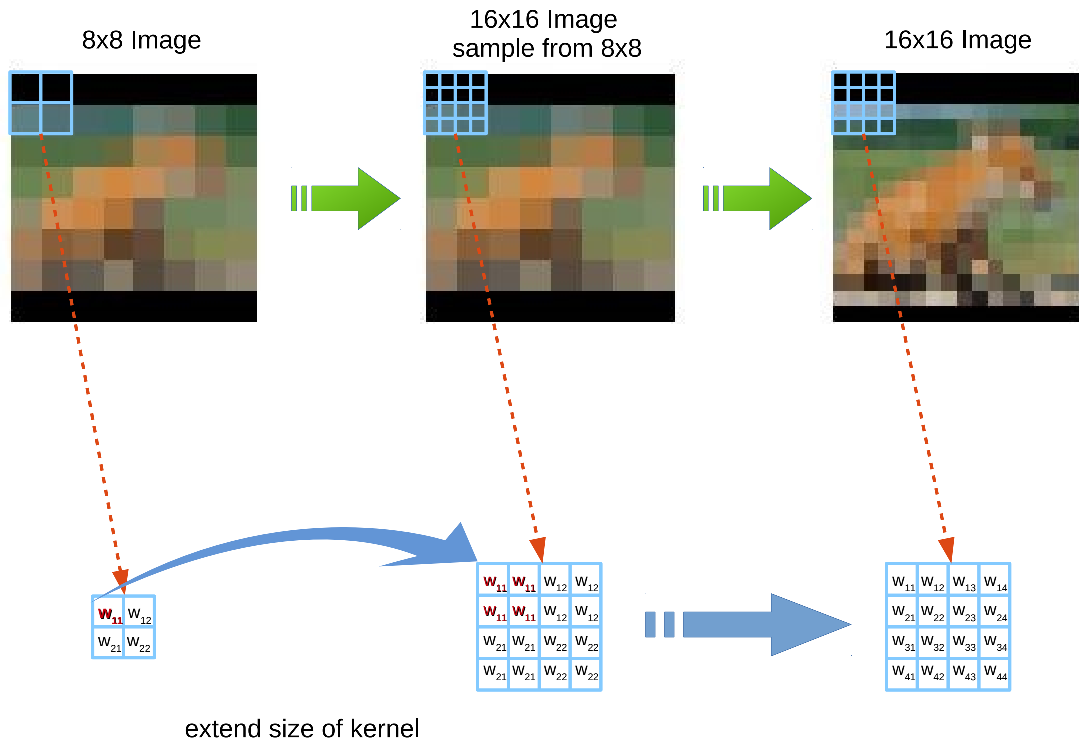

Figure 4.1. Parameters transformation in convolutional layer of DNNs.

Based on the equation (4.1), the output of I_{16s} can be similarly written with $\boldsymbol{Y}^{(16)}$ as

$$
\begin{aligned}
y_{11}^{(16s)} &= \varphi((\boldsymbol{W}^{(16s)} * x)_{11} - b^{(16s)}) \\
y_{12}^{(16s)} &= \varphi((\boldsymbol{W}^{(16s)} * x)_{12} - b^{(16s)}) \\
y_{13}^{(16s)} &= \varphi((\boldsymbol{W}^{(16s)} * x)_{13} - b^{(16s)}) \\
&\ldots
\end{aligned}
\tag{4.6}
$$

If the same result intend to be earned between $\boldsymbol{Y}^{(8)}$ and $\boldsymbol{Y}^{(16s)}$, the weights should also be copied and repeated as

$$
\begin{aligned}
& w_{11}^{(16s)} = w_{12}^{(16s)} = w_{21}^{(16s)} = w_{22}^{(16s)} = \frac{w_{11}^{(8)}}{4} \\
& w_{13}^{(16s)} = w_{14}^{(16s)} = w_{23}^{(16s)} = w_{24}^{(16s)} = \frac{w_{12}^{(8)}}{4} \\
& \ldots \\
& w_{31}^{(16s)} = w_{32}^{(16s)} = w_{41}^{(16s)} = w_{42}^{(16s)} = \frac{w_{21}^{(8)}}{4} \\
& \ldots
\end{aligned}
\tag{4.7}
$$

The reason of the values of the weights in $\boldsymbol{W}^{(16s)}$ to be $1/4$ in $\boldsymbol{W}^{(8)}$ is that one pixel in I_8 is repeated 4 times in I_{16s}. Therefore, the weights need to be divided by 4 to keep the same output between $\boldsymbol{Y}^{(8)}$ and $\boldsymbol{Y}^{(16s)}$. In addition, when the I_{16s} is utilised to convolve with the filter, the same results can be scored with $\boldsymbol{Y}^{(8)}$ in every other values, which means,

$$\begin{aligned}
&y_{11}^{(8)} = y_{11}^{(16s)}\\
&y_{12}^{(8)} = y_{13}^{(16s)}\\
&...\\
&y_{21}^{(8)} = y_{31}^{(16s)}\\
&...
\end{aligned} \tag{4.8}$$

Moreover, the different values between them can be ignored in the next layer.

Finally, the solution of transforming parameters in the convolutional layer is proposed. It is achieved by extending and copying from $\boldsymbol{W}^{(8)}$ to $\boldsymbol{W}^{(16)}$, and from $b^{(8)}$ to $b^{(16)}$. Although there is some difference between I_{16} and I_{16s}, the performance maybe reduced a little while not impact too much. The difference between them is that I_{16} has the new details needed to be learned by DNN_{16}. This keeps old information in I_8 and DNN_{16} continues to learn from I_{16}. The entire process of one transforming in convolutional layer is shown in Figure 4.1.

4.1.2 Inner Product Layer

Generally, the inner product layer is also referred as the fully connected layer. It treats the input as a simple vector and takes all neurons to produce output. In the experiments, the input for this layer is the output of the previous convolutional layer. Based on the relationship between Y_8 and Y_{16}, the solution to omit undesired output and keep similar output is required to be exploited in this layer.

Firstly, the inner product layers of DNN_8 produces the output as a vector $\boldsymbol{v}^{(8)}$, and computes matrix multiplication with the weights $\boldsymbol{W}^{(8)}$, as

$$\boldsymbol{y}^{(8)} = \boldsymbol{W}^{(8)} * \boldsymbol{v}^{(8)} - \boldsymbol{b}^{(8)} \tag{4.9}$$

where $\boldsymbol{W}^{(8)}$ and $\boldsymbol{b}^{(8)}$ denote the weight and the bias in the inner product layer, respectively. The same inner product in DNN_{16} is

$$\boldsymbol{y}^{(16)} = \boldsymbol{W}^{(16)} * \boldsymbol{v}^{(16)} - \boldsymbol{b}^{(16)} \tag{4.10}$$

The parameters of $\boldsymbol{W}^{(8)}$ and $\boldsymbol{b}^{(8)}$ of the pre-trained model can be extracted and transformed into $\boldsymbol{W}^{(16)}$ and $\boldsymbol{b}^{(16)}$. For the bias, it is similar with that in the convolutional layer.

Nevertheless, $\boldsymbol{W}^{(8)}$ and $\boldsymbol{W}^{(16)}$ do not have the same dimensions, which causes the problem on the transformation between them. As a result, the equation (4.8) is recalled, the relationship between $\boldsymbol{Y}^{(8)}$ and $\boldsymbol{Y}^{(16)}$ of convolutional layer is shown in Figure 4.2. In

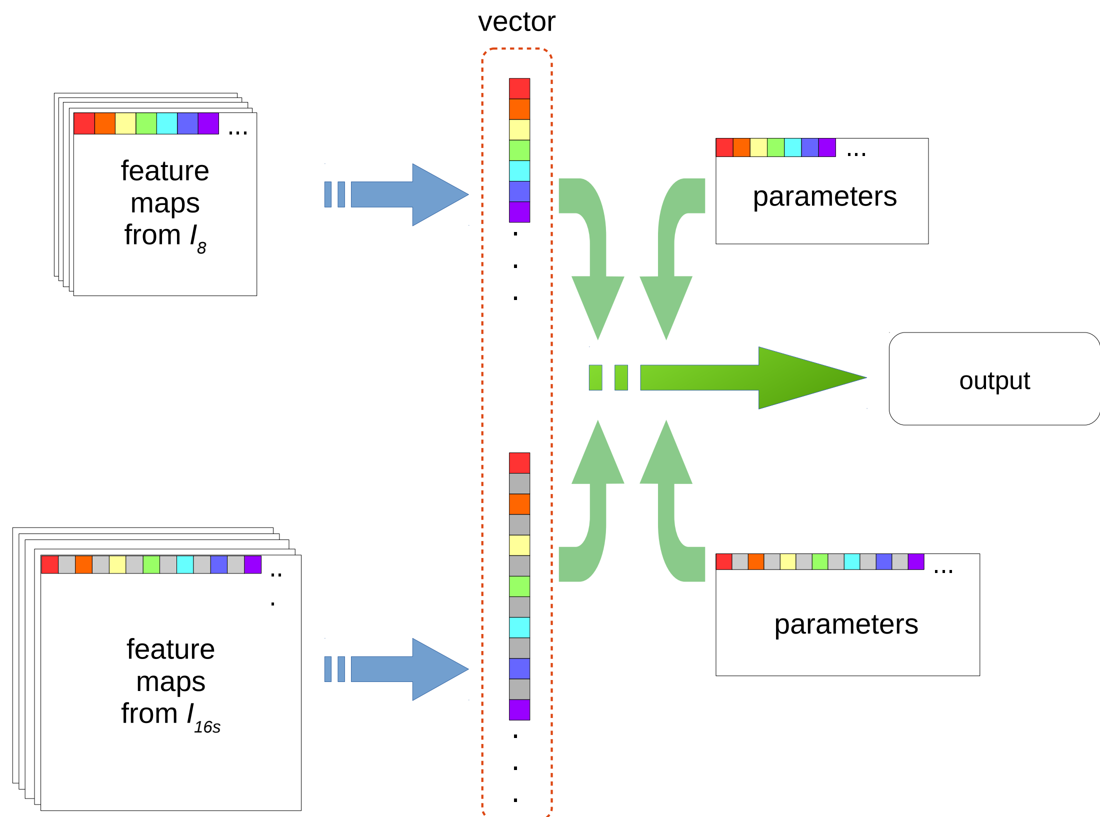

Figure 4.2. Parameters transformation in inner product layers of DNNs.

the figure, there are two sets of feature maps from the convolutional layers of DNN_8 and DNN_{16} in the left. The coloured blocks are represented by the corresponding pixels needed to be kept, while the values of grey blocks are necessary to be omitted in this layer. Next, they are reshaped as a vector for the next step of multiplication. In the right side of the figure, there are two matrices of parameters in the inner product layer, which also have corresponded colour coding. For example, the red blocks in the both feature maps and vectors are the pixels with the same values, which need to be multiplied with the corresponding red blocks in the matrices of parameters. Since the coloured blocks in feature maps or vectors of DNN_{16} need to be retained, the blocks in the parameters of

DNN_8 should be put to the corresponding position of DNN_{16}. The grey blocks in the feature maps and vectors of DNN_{16} are omitted, and the grey blocks in the parameters of DNN_{16} are set to zero. After multiplying by parameters matrices, the values of grey blocks are changed to zeros, which do not have an effect to the result. In other words, we can add zeros between two original values of $\boldsymbol{W}^{(8)}$ in both horizontal and vertical direction to generate $\boldsymbol{W}^{(16)}$.

Eventually, the solution of transforming parameters in inner product layers has been presented. For the weights, add zero values between two original values in $\boldsymbol{W}^{(8)}$ to earn the transformed parameters of $\boldsymbol{W}^{(16)}$. The zero values can assist to omit the difference from the output of convolutional layer and keep the output as similar as DNN_8. Meanwhile, for the bias, a direct solution is to copy $\boldsymbol{b}^{(8)}$ to $\boldsymbol{b}^{(16)}$. For following inner product layers with the same input (e.g. $fc8$ layer), the parameters can be simply copied.

In general, this subsection provides a solution to deal with parameters of upscaled filters for both convolutional layer and inner product layer. Algorithm 1 summarises the procedure of parameters transformation from M_8 to M_{16t} by using our incremental learning idea. In the experiments, the same way can be employed to transform parameters from M_{16} to M_{32t}. In brief, the weights in the convolutional layer just need to be extended and copied, while in the inner product layer zeros need to be added. The bias vectors are copied in the both layers.

Algorithm 1 Transform parameters from M_8 to M_{16t}

1: Load the tiny images network DNN_8 and model M_8
2: Load the large images network DNN_{16}
3: Create the model file M_{16t} and initialise as 0
4: **for** i in all layers **do**
5: copy the bias vector from M_8 to M_{16t} ▷ transform the bias
6: **if** i is '$conv1$' **then** ▷ transform the weights in convolutional layer
7: copy and extend the weights by $1/4$
8: **else if** i is '$fc6$' **then** ▷ transform the weights in the first inner product layer
9: add zeros and copy the weights
10: **else**
11: copy the weights ▷ transform the weights in the other inner product layer
12: **end if**
13: **end for**

4.2 Adding Convolutional Filters

The networks with additional filters not only extend the size of filters, which are the same with the networks with scaled filters, but also add the number of filters to the convolutional layer. This requires the initialisation of new added filters. In the experiments, 12 filters (denoted as F_{12}) is applied to DNN_8, F_{16} to DNN_{16}, and F_{24} to DNN_{32}. Next, we introduce three methods to initialise parameters for the new added filters.

4.2.1 Initialisation of Zero

The straightforward way is setting all parameters of the new added filters to zero. Using zero to ignore the extra output of new filters in each layer can reduce the influence of output, which has the similar idea used in transforming parameters in inner product layer with scaled filters. In short, the first method is to set all the weights and the bias in the new added filters as zero.

4.2.2 Initialisation of Random Gaussian Values

In addition, the original approach implemented in Caffe can be utilised. After checking the net configuration file, the random Gaussian values are used in the weights, and constants are set to the bias. The settings of initialising parameters in the three layers of DNNs in the experiments are shown in Table 4.1. The Gaussian distribution [64] has zero

Table 4.1. Summary of initialising parameters.

	weight	bias
$conv1$	Gaussian (0.01)	0
$fc6$	Gaussian (0.005)	1
$fc8$	Gaussian (0.01)	0

mean and standard deviation shown in parentheses. Based on that, the same mean value and standard deviation are used to generate the random Gaussian values for the weights and the same constants can be set to the bias. In addition, there are three related extensions for the experiments, which are random Gaussian values divided by 10, random Gaussian values divided by 100, and random Gaussian values divided by 1000. They attempt to

scale the initialised values for reducing the influence to the output. Thus, there are four methods in the experiments to set the parameters of new added filters based on random Gaussian Values.

4.2.3 Initialisation of Flipping the Existing Filters

After training a DNN model, the convolutional filters can be learned and visualised. In the paper of Krizhevsky et al [5], there are 96 convolutional filters of size $11 \times 11 \times 3$ learned by the first convolutional layer (conv1) that show a variety of frequency- and orientation-selective filters, as well as various coloured blobs (Figure 4.3). Even though the number of filters implemented in our experiments is less, they still can learn parts of them. Thus, we attempt to find a way to initialise the new added filters as supplementary filters. Theoretically, if the supplementary filters are necessary in the network, they can boost trains.

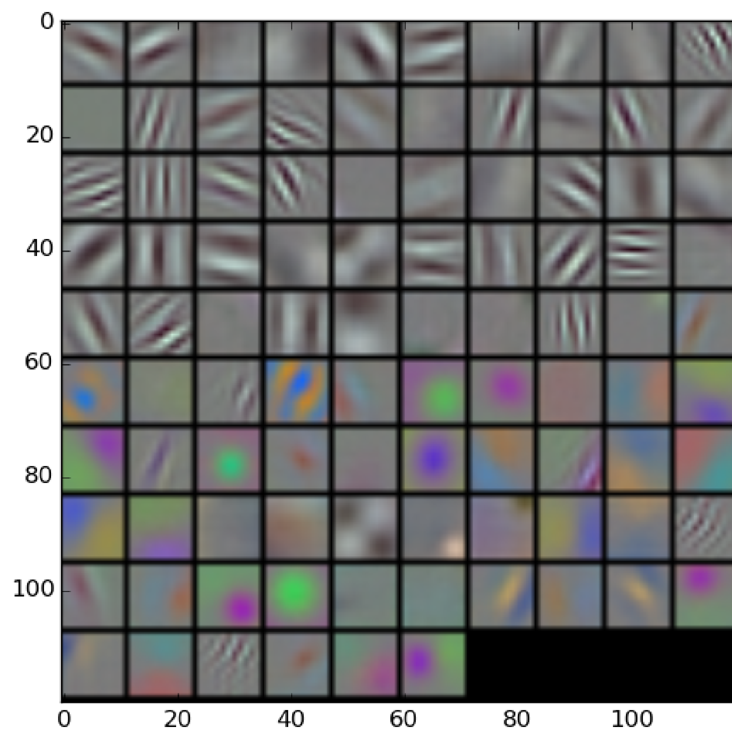

Figure 4.3. The first convolutional layers filters by Krizhevsky et al [5].

From Figure 4.3, it can be seen that there are several orientation-selective filters. If the operations of copy and flipping can be exploited, more directions of filters will be acquired to extract features. For example, if the network has learned a orientation-selective filter in one orientation/angle, more orientations in different angles can be obtained by flipping. Therefore, the flipping idea is proposed to supplement filters, which includes flipping horizontally, vertically, diagonally, and centrally, illustrated in Figure 4.4.

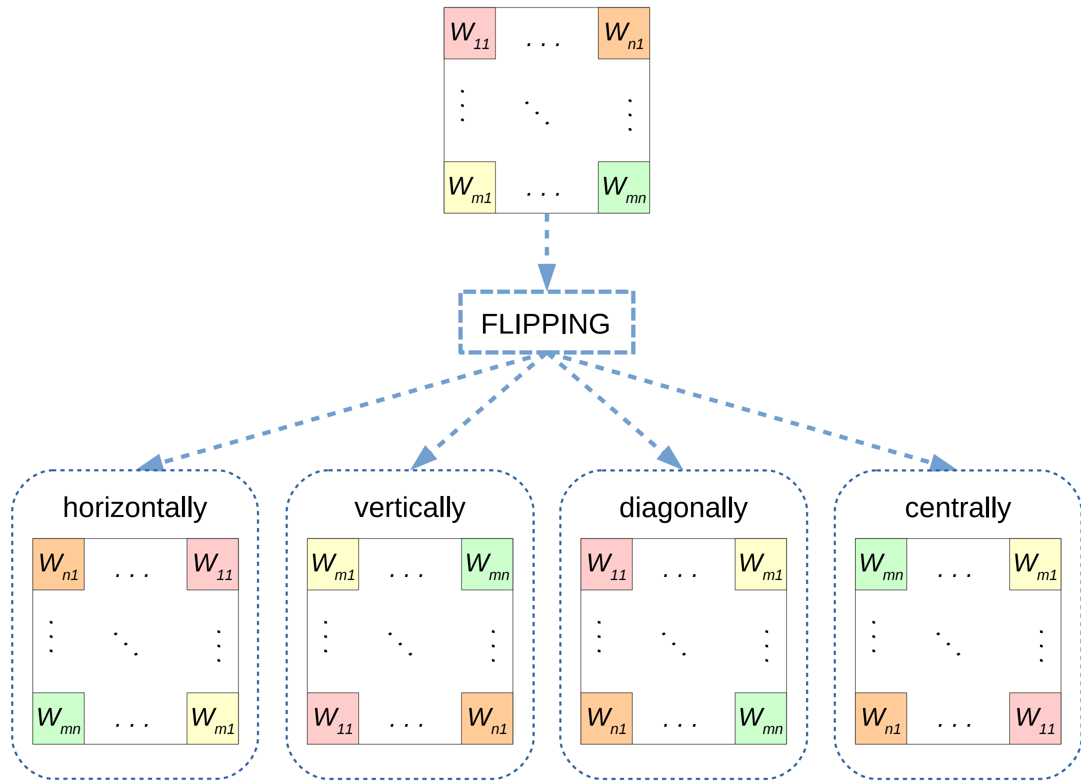

Figure 4.4. Flipping filters.

To summarise, a solution for adding convolutional filters is proposed. In general, the transformation of additional parameters can be divided into two parts. One is the same part with scaled filters, which can be executed by the same method. The other is the new added filters, different initialised ways should be implemented. The algorithm is similar to Algorithm 1, which needs to initialise new added filters. In brief, three main ideas to implement initialisation are provided, including zero, setting random Gaussian values, and flipping the existing filters. In the experiments, these are tested by upscaling from DNN_8 to DNN_{16}, and DNN_{16} to DNN_{32}.

4.3 Fine-Tuning

After dealing with upscaled parameters transformation in the previous section, the idea of incremental learning is realised through training DNN of the "fine" model. Caffe provides fine-tuning function, which takes an already learned model, adopts the architecture, and resumes training from the already learned model weights [4]. The corresponding command is

```
$ caffe train -solver path/to/solver.prototxt
              -weights path/to/weights.caffemodel
```

In the experiments, the trained model M_8 was scaled and filters added to form the model M_{16t}, which is used to I_{16} in DNN_{16} for fine-tuning. After fine-tuning, the trained model M_{16} can be obtained, and the same method is applied to generate the model M_{32t}. Finally, repeat the fine-tuning step to I_{32} in DNN_{32} with M_{32t}. As a consequent, three step incremental learning is completed, using DNN_8, DNN_{16}, and DNN_{32}. Figure 4.5 is a sample log of fine-tuning where iteration 0 accuracy starts from that achieved with DNN_8.

```
I0325 10:48:52.009484 19520 net.cpp:219] Network initialization done.
I0325 10:48:52.009487 19520 net.cpp:220] Memory required for data: 2278208
I0325 10:48:52.009534 19520 solver.cpp:41] Solver scaffolding done.
I0325 10:48:52.009541 19520 caffe.cpp:115] Finetuning from ./TEMPWORK/trans_caffe_ImageNet-tiny-16x16-sRGB-net-3.caffemodel
I0325 10:48:52.018720 19520 solver.cpp:160] Solving CaffeNet
I0325 10:48:52.018745 19520 solver.cpp:161] Learning Rate Policy: step
I0325 10:48:52.018774 19520 solver.cpp:264] Iteration 0, Testing net (#0)
I0325 10:48:55.321952 19520 solver.cpp:315]     Test net output #0: accuracy = 0.0815802
I0325 10:48:55.321985 19520 solver.cpp:315]     Test net output #1: loss = 5.43995 (* 1 = 5.43995 loss)
I0325 10:48:55.343711 19520 solver.cpp:209] Iteration 0, loss = 5.58631
I0325 10:48:55.343741 19520 solver.cpp:224]     Train net output #0: loss = 5.58631 (* 1 = 5.58631 loss)
I0325 10:48:55.343754 19520 solver.cpp:445] Iteration 0, lr = 0.001
I0325 10:48:55.780313 19520 solver.cpp:209] Iteration 20, loss = 5.46389
I0325 10:48:55.780344 19520 solver.cpp:224]     Train net output #0: loss = 5.46389 (* 1 = 5.46389 loss)
I0325 10:48:55.780352 19520 solver.cpp:445] Iteration 20, lr = 0.001
```

Figure 4.5. Caffe log output during fine-tuning.

It is noted in the instructions of Caffe about tricks for fine-tuning, which consist of learning the last layer first, and reduce the learning rate [4]. Based on our own situation, only the overall learning rate $base_lr$ in the solver is decreased. It makes the model more slowly to learn the new data, and preserves the initialisation from pre-training. In the experiments, the learning rate is dropped down by 10, when using the transformed model M_{16t} for fine-tuning with I_{16} and DNN_{16}, and also for M_{32t}.

4.4 Summary

In conclusion, the method of implementing incremental learning in DNNs is elaborated, which consists of three parts. First is to construct the networks with scaled filters. Second, the networks are formed with additional filters. Finally, the method of fine-tuning provided by Caffe is described to complete the incremental learning. Based on the methods presented above, we will illustrate experimental results in the next chapter.

5 EXPERIMENTS AND RESULTS

In this chapter, the experiments are summarised to present the results on the ImageNet dataset and the Places205 dataset. Firstly, the results by using the incremental networks with scaled filters are shown, and compared to the traditional training method. In addition, there are results of initialising new added filters based on different initialisation methods. Finally, the results of incremental networks with added filters are illustrated, which involve two comparative evaluations. One is the traditional training method and incremental learning with added filters, the other is the networks with scaled filters and added filters. From the results, our method achieves superior efficiency and higher performance.

5.1 Evaluation of Incremental Learning with Scaled Filters

Here is the result of I_{16s} with K_4 in DNN_{16} transformed from I_8 with K_2 in DNN_8, illustrated in Figure 5.1. The red curve with squares depicts the training curve by the

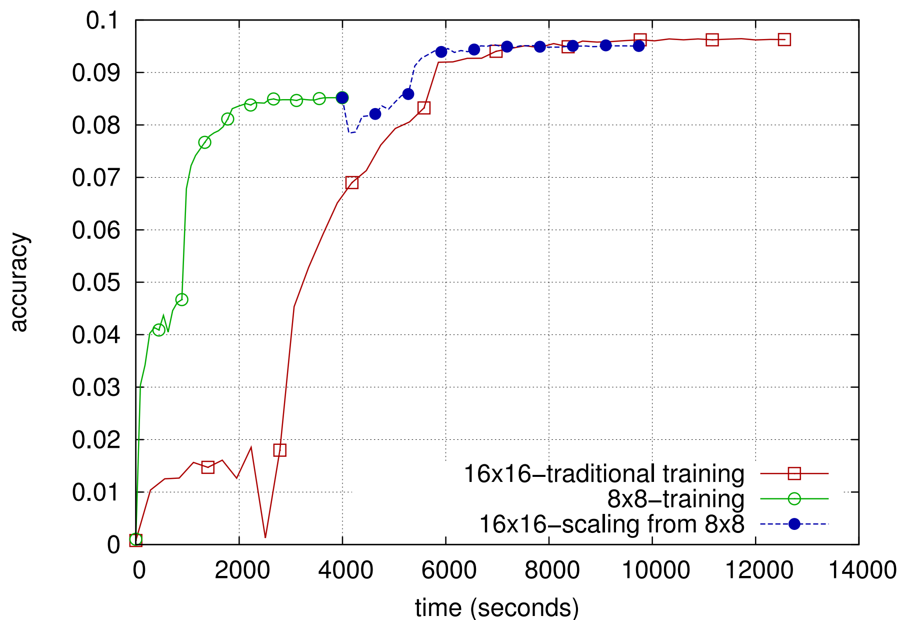

Figure 5.1. Comparison of training curves of the traditional method and incremental learning with scaled filters on the ImageNet dataset.

traditional method for training I_{16s} from the beginning. The curves with circles depict the

incremental learning method, while green and blue describe the training curves of I_8 and I_{16s}, respectively. From the curves, there is no performance gap when transforming models from M_8 to M_{16t} but similar final results are obtained. This means that the transformed model M_{16t} successfully keeps the original information from the pre-trained coarse model M_8 and continues to learn the new information in DNN_{16}. Therefore, the idea of incremental learning by transforming DNN parameters from coarse images network to fine images network seems to work.

In the Figure 5.2, for the networks with scaled filters, they are extended twice by the same method. The first is from DNN_8 with K_2 to DNN_{16} with K_4, the other is continuously from DNN_{16} with K_4 to DNN_{32} with K_8. Similarly, the red curve with triangles depicts

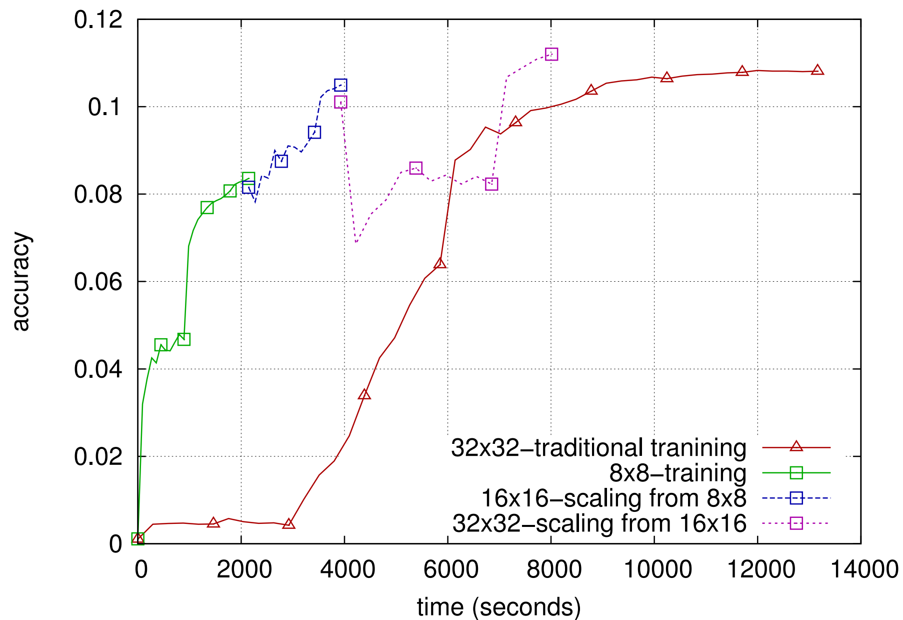

Figure 5.2. Comparison of training curves of the traditional method and incremental learning with scaled filters on the ImageNet dataset.

the training curve by traditional method for training I_{32} from beginning. The curves with squares show the incremental learning method, while green, blue, magenta respectively describe the training curves of I_8, I_{16}, and I_{32}. There are two gaps when the size of filters are scaled. The reason of that is new details added to the input and needed to be learned by DNNs. For example, there is a gap of transformation from DNN_8 to DNN_{16}. It is meant that I_{16} brings new visual features, the I_8 does not have, which are necessary to be learned by DNN_{16}. It is known that DNN_{16} learns the new details as the curve quickly improves. The transformation from DNN_{16} to DNN_{32} has the same situation.

As a result, the accuracy can be improved from 10.8% to 11.2% and only costs 61% time, shown in Table 5.1.

Table 5.1. Final accuracies of the traditional method and incremental learning with scaled filters on the ImageNet dataset.

	Accuracy	Time
Traditional Method	10.8%	$13160s$
Scaled Filters(Sec. 4.1)	11.2%	$8011s$

In addition, the Places205 dataset is also applied to verify the idea of scaled filters, which utilises the same structure of networks with the ImageNet dataset. The results are shown in Figure 5.3 and Table 5.2, where the incremental method saves 38% of the training time and increases the accuracy from 14.1% to 15.9%.

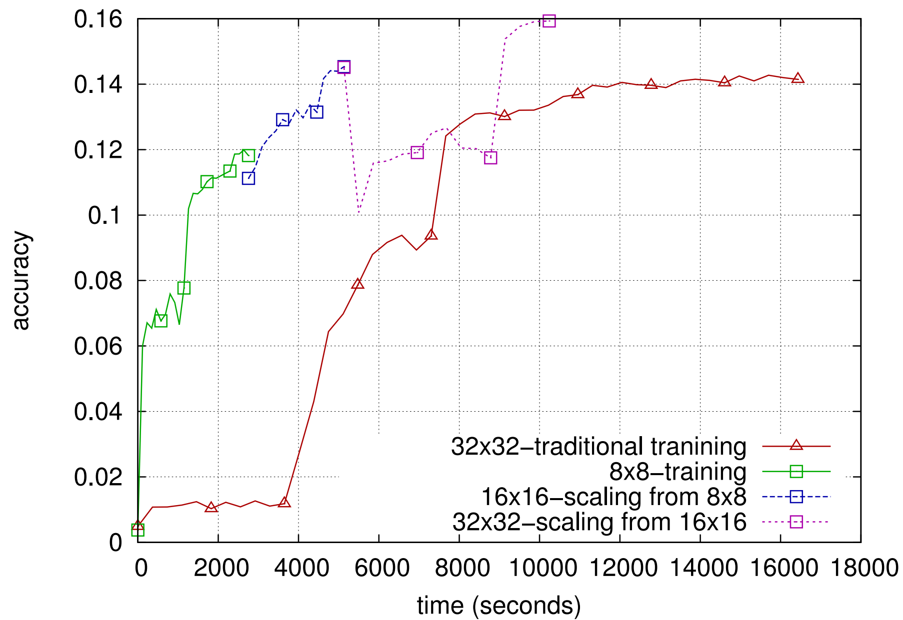

Figure 5.3. Comparison of training curves of the traditional method and incremental learning with scaled filters on the Places205 dataset.

Table 5.2. Final accuracies of the traditional method and incremental learning with scaled filters on the Places205 dataset.

	Accuracy	Time
Traditional Method	14.1%	$16431s$
Scaled Filters	15.9%	$10239s$

5.2 Experiments on Adding and Initialising New Convolutional Filters

Before showing the final results of incremental training with added filters, we show the results of initialising new filters by different methods. In this part, the parameters of convolutional layer are F_{12} with K_4 in I_{16}, and F_{24} with K_8 in I_{32}. The method of scaled filters is the same with previous experiment, and the number of filters is doubled from 12 to 24. As a result, the extra 12 filters are necessary to initialise with zero (Sec. 4.2.1), random Gaussian values (Sec. 4.2.2), or flipping the existing filters (Sec. 4.2.3).

Firstly, zero values are set to new filters and fine-tuning to generate the training curve, illustrated in Figure 5.4 by the red solid lines with squares. In addition, the approach of

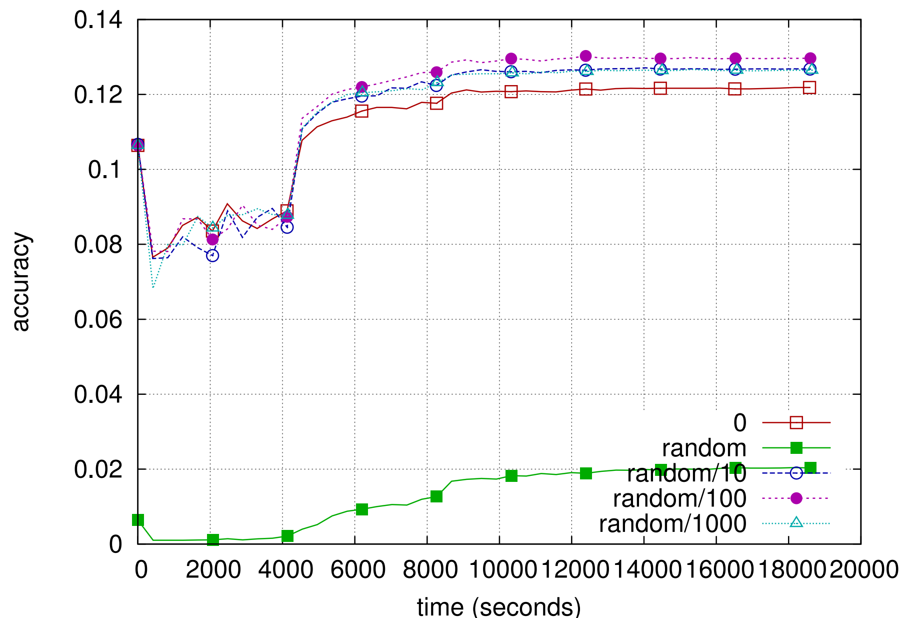

Figure 5.4. Results on initialising new added filters as 0, random Guassian values, random Guassian values divided by 10, 100 and 1000 on the ImageNet dataset.

random Gaussian values is implemented. The mean value and standard deviation have been defined in Table 3.2. In the light of that, the random Gaussian values are extended as divided by 10, 100, and 1000. The different colours and labels are utilised to distinguish the curves of these four methods, shown in Figure 5.4. It is known that they have very similar results, except the random Gaussian values. The reason is that the random Gaussian values are too large and affect to the output. Meanwhile the final result of fine-tuning is undesirable, because the large values make the DNNs "forgot" the pre-trained model and cannot adjust quickly. Thus, the division method to reduce the effect of the initial values works better.

Next, the flipping method is implemented, which can generate supplementary filters to the DNN. Here, four directions of flipping, horizontal, vertical, diagonal, and central, are utilised in the experiments, whose training curves are illustrated in Figure 5.5. The flipping method can obtain similar results at the end, while the starting points are quite different. Because some filters are new to the network, they may influence the output as giving different values with the pre-trained model.

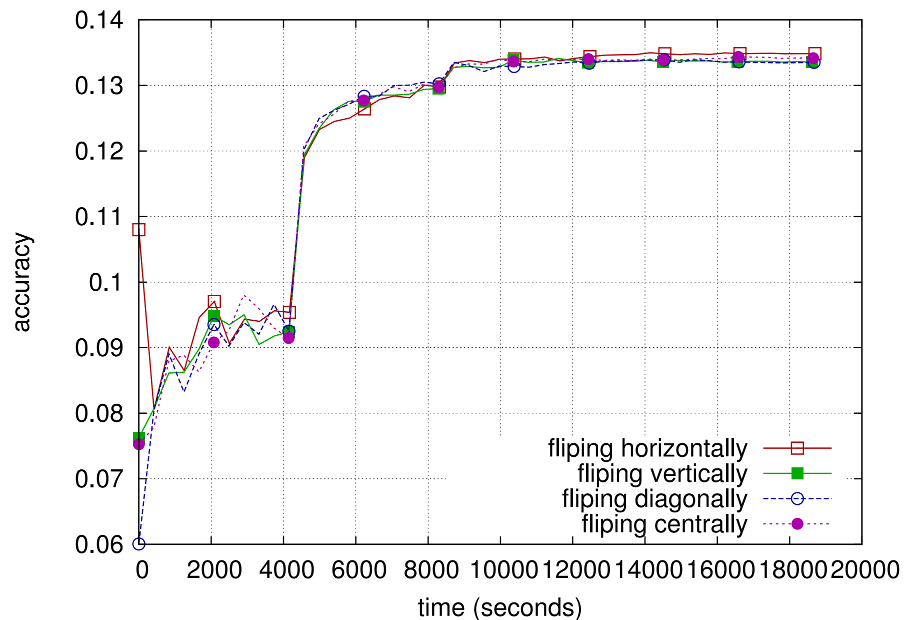

Figure 5.5. Results on initialising new added filters as flipping horizontally, vertically, diagonally and centrally on the ImageNet dataset.

To conclude, several methods are provided to initialise the new added filters. Most of them can get similar results, thus the original way, random Gaussian values divide by 100, is employed in the follow experiments.

5.3 Evaluation of Incremental Learning with Added Filters

For evaluating the effect of networks with added filters, the number of filters can be separated into two parts. One is the same number of filters with the "coarse" network, the other is the new extra filters in the "fine" network. For the same part, the method of scaling filters described in Section 4.1 is utilised to transform parameters. For the extra part, the initialisation method is applied. For example, the transformation from DNN_8 with F_{12} of K_2 to DNN_{16} with F_{16} of K_4 consists of two parts. The first 12 filters in DNN_{16} are transformed from the filters in DNN_8, while the new 4 filters are initialised by random Gaussian values divided by 100.

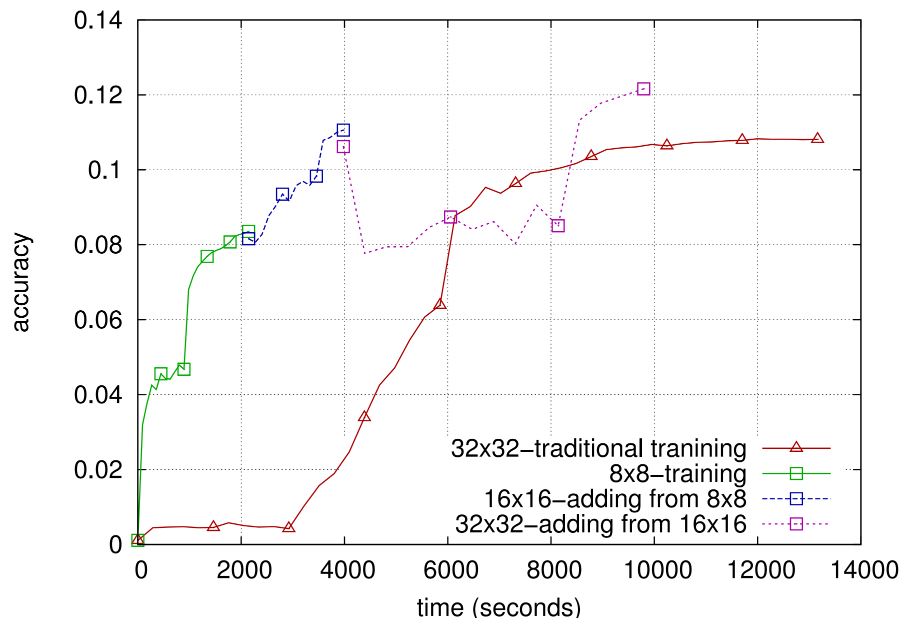

Figure 5.6. Results of the traditional method and incremental learning with added and scaled filters for the ImageNet dataset.

Table 5.3. Final accuracies of the traditional method and incremental learning with added filters on the ImageNet dataset.

	Accuracy	Time
Traditional Method	10.8%	$13160s$
Added Filters	12.1%	$9794s$

In the experiments, the number of filters are increased twice by the same method. The first is from DNN_8 with F_{12} of K_2 to DNN_{16} with F_{16} of K_4. Secondly, incrementally

add to DNN_{32} with F_{24} of K_8. The results of traditional method and incremental learning method are shown in Figure 5.6. The red curve with triangles depicts the training curve by traditional method for training I_{32} from the beginning. The curves with squares show the result of incremental learning method, and green, blue, magenta describe the training curves of I_8, I_{16}, I_{32}, accordingly. As a result, the accuracy can be improved to 12.1% and only use 74% time, shown in Table 5.3.

The results of networks with added filters by using the Places205 dataset are shown in Figure 5.7 and Table 5.4. In these experiments, the same structure of networks with the ImageNet dataset are implemented in adding filters, which raise the performance from 14.1% to 17.0% and only spend 66% time.

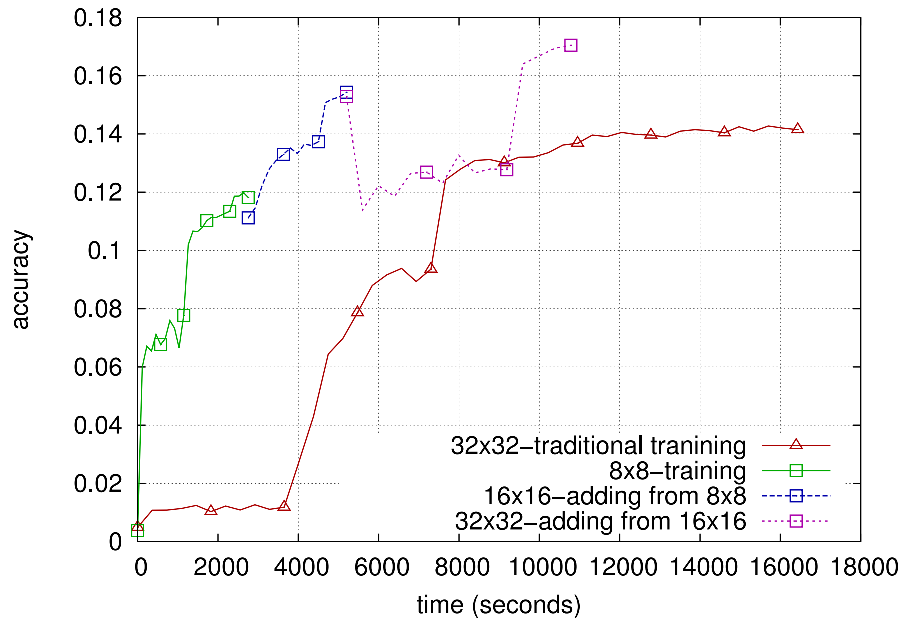

Figure 5.7. Results of the traditional method and incremental learning with added and scaled filters for the Places205 dataset.

Table 5.4. Final accuracies of the traditional method and incremental learning with added filters on the Places205 dataset.

	Accuracy	Time
Traditional Method	14.1%	$16431s$
Added Filters	17.0%	$10786s$

Finally, two results of incremental learning methods are compared, which include scaled filters and added filters for the ImageNet dataset and the Places205 dataset, illustrated in Figure 5.8. The red curve with triangles depicts the traditional training method for I_{32}. The cyan curve with diamonds shows the training procedure of I_8. The two blue curves with squares represent the method of scaled filters, while the two green curves with circles are the method of added filters. Both of the methods spend less time to achieve a higher performance than the traditional training method.

5.4 Summary

The two methods of incremental learning are successfully implemented. Both of them can improve efficiency and increase performance. Although the training time of networks with scaled filters is shorter, DNNs need more filters and larger size of filters to learn images with higher resolution or more details.

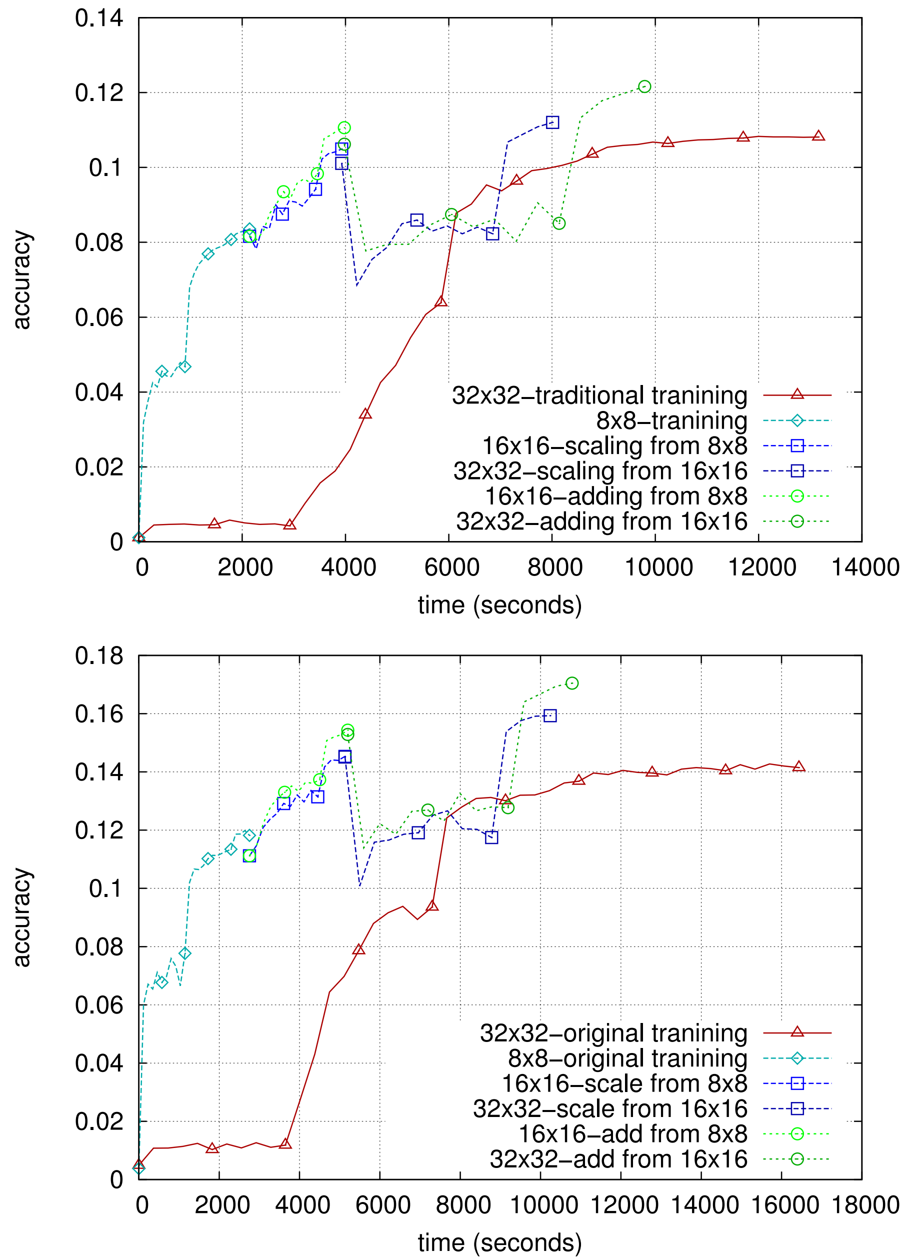

Figure 5.8. Results of the traditional method and incremental learning method for the ImageNet dataset (top) and the Places205 dataset (bottom).

6 CONCLUSIONS

This thesis proposes a promising method, incremental learning, to solve the task of image classification. It is shown to be an efficient tool for the task by using deep neural networks. One of the advantages is that a pre-trained model can be utilised as the initialisation of other networks to shorten the training time. The time-consuming training of large-scale datasets, e.g. ImageNet dataset, is a disturbing problem. The incremental learning method can achieve superior efficiency for training without sacrificing any accuracy. Another beneficial point is that the performance can slightly improve. Therefore, the main objective of the work is to improve the efficiency and the performance through incremental learning methods.

There are two kinds of incremental learning methods based on DNNs. One is the networks with scaled filters, which can improve the accuracy from 10.8% to 11.2% and from 14.1% to 15.9% for the ImageNet dataset and the Places205 dataset, respectively. Although they are not significantly improved, the training times were reduced 39% and 37% for these two datasets, which means it takes a shorter time to achieve a similar accuracy. The other method is the networks with added filters, which increase the accuracy to 12.1% for the ImageNet dataset, and to 17.0% for the Places205 dataset. The computing times were decreased by 25% and 34% for training, accordingly. These results imply that the performance can also be improved while learning is more efficient. Therefore, both efficiency and performance were improved by incremental learning.

The two methods of incremental learning were implemented in Caffe. On one hand, the number of filters are retained and the size of filters are extended in the convolutional layer to form the networks with scaled filters. The parameters of filters are copied and extended with a small adjustment from "coarse" network to "fine" network for keeping the learned information from low-resolution images. On the other hand, both the number of filters and the size of filters are changed to construct the networks with added filters. In addition, a suitable approach of initialisation to the new added filters were tested and random Gaussian values divided by 100 found as the best.

The main idea of incremental learning is to retain and transform the learned information from pre-trained model of low-resolution images to new "fine" model, and continually learn among images of increasingly higher resolution. In an implementation of these methods, one needs to pay attention on the training parameters, not to forget the pre-trained parameters.

The methods of incremental learning can improve the efficiency and the performance, and there are some promising future directions. Firstly, the peculiar size of scaled filters may need to be considered, such as transforming the parameters of filters from K_2 to K_3. In the experiments, the size of filters is only doubled by each step of incremental learning. Secondly, an integrated training procedure from "coarse" images to full size images may promote the improvement of efficiency and performance observably. Finally, to train images with high resolution, the method of adding layers in CNNs is a challenging and promising problem needed to be investigated in the future.

REFERENCES

[1] Graham Taylor. Unsupervised learning - tutorial on deep learning for vision. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, 2014.

[2] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, 2nd edition, 1998.

[3] Michiel Hermans and Benjamin Schrauwen. Training and analysing deep recurrent neural networks. In *Neural Information Processing Systems (NIPS)*, pages 190–198, 2013.

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

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *Neural Information Processing Systems (NIPS)*, 2012.

[6] Simon J.D. Prince. *Computer Vision: Models Learning and Inference*. Cambridge University Press, 2012.

[7] Li Deng and Dong Yu. Deep learning: Methods and applications. Technical Report MSR-TR-2014-21, Microsoft Research, May 2014.

[8] Yoshua Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, January 2009.

[9] Bolei Zhou, Agata Lapedriza, Jianxiong Xiao, Antonio Torralba, and Aude Oliva. Learning deep features for scene recognition using places database. In *Neural Information Processing Systems (NIPS)*, pages 487–495, 2014.

[10] K. Ramesha, N. Srikanth, K.B. Raja, K.R. Venugopal, and L.M. Patnaik. Advanced biometric identification on face, gender and age recognition. In *Advances in Recent Technologies in Communication and Computing, 2009. ARTCom '09. International Conference on*, pages 23–27, 2009.

[11] Yuichi Hirano and Tomoharu Nagao. Feature transformation using filter array for automatic construction of image classification. In *Computational Intelligence and Applications (IWCIA), 2014 IEEE 7th International Workshop on*, pages 59–64, 2014.

[12] Wikipedia. Statistical classification, 2015. [The last modified on 13 June 2015].

[13] Ashley Walker Robert Fisher, Simon Perkins and Erik Wolfart. *Hypermedia Image Processing Reference*. Online Book, 2000.

[14] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., second edition edition, 2005.

[15] Wikipedia. Test set, 2015. [The last modified on 20 June 2015].

[16] Ke-Lin Du and M. N.S. Swamy. *Neural Networks and Statistical Learning*. Springer Publishing Company, Incorporated, 2013.

[17] Christoph H. Lampert, Hannes Nickisch, and Stefan Harmeling. Learning to detect unseen object classes by between class attribute transfer. In *Computer Vision and Pattern Recognition (CVPR), 2009 IEEE Conference on*, pages 951–958, 2009.

[18] Christoph H. Lampert, Hannes Nickisch, and Stefan Harmeling. Attribute-based classification for zero-shot visual object categorization. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36(3):453–465, 2014.

[19] Daniel N. Osherson, Joshua Stern, Ormond Wilkie, Michael Stob, and Edward E. Smith. Default probability. *Cognitive Science*, 15(2):251–269, 1991.

[20] Charles Kemp, Joshua B. Tenenbaum, Thomas L. Griffiths, Takeshi Yamada, and Naonori Ueda. Learning systems of concepts with an infinite relational model. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pages 381–388, 2006.

[21] Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao, and Li Fei-Fei. Novel dataset for fine-grained image categorization. In *First Workshop on Fine-Grained Visual Categorization, IEEE Conference on Computer Vision and Pattern Recognition*, 2011.

[22] Omkar M. Parkhi, Andrea Vedaldi, Andrew Zisserman, and C. V. Jawahar. Cats and dogs. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3498–3505, 2012.

[23] Catherine Wah, Steve Branson, Peter Welinder, Pietro Perona, and Serge Belongie. The caltech-ucsd birds-200-2011 dataset. Technical Report CNS-TR-2011-001, California Institute of Technology, 2011.

[24] Thomas Berg, Jiongxin Liu, Seung Woo Lee, Michelle L. Alexander, David W. Jacobs, and Peter N. Belhumeur. Birdsnap: Large-scale fine-grained visual categorization of birds. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 2019–2026, 2014.

[25] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition (CVPR), 2009 IEEE Conference on*, pages 248–255, 2009.

[26] Alex Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, University of Toronto, 2009.

[27] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision (IJCV)*, pages 1–42, April 2014.

[28] Igor Aleksander and Helen Morton. *An introduction to neural computing*. Chapman and Hall, 1990.

[29] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[30] Wee Kheng Leow and R. Miikkulainen. Representing visual schemas in neural networks for scene analysis. In *Neural Networks, 1993., IEEE International Conference on*, pages 1612–1617 vol.3, 1993.

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

[32] Hakan Arslan and Ahmet Kuzucu. A robot control application with neural networks. In *Industrial Electronics, 1997. ISIE '97., Proceedings of the IEEE International Symposium on*, pages 1238–1241 vol.3, 1997.

[33] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature 323*, pages 533–536, 1986.

[34] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

[35] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[36] Andrew R. Webb and Keith D. Copsey. *Statistical Pattern Recognition*. Wiley, 3 edition, 2011.

[37] Arthur Earl Bryson and Yu-Chi Ho. *Applied Optimal Control: Optimization, Estimation and Control*. Blaisdell Publishing, 1969.

[38] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[39] Paul J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Harvard University Press, 1974.

[40] Ethem Alpaydin. *Introduction to Machine Learning*. The MIT Press, 2nd edition, 2010.

[41] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern classification*. Wiley, 2001.

[42] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning, Data Mining, Interference, and Prediction*. Springer, 2009.

[43] Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. In James A. Anderson and Edward Rosenfeld, editors, *Neurocomputing: Foundations of Research*, pages 123–134. MIT Press, Cambridge, MA, USA, 1988.

[44] Alex Graves. *Supervised sequence labelling with recurrent neural networks*. Springer, 2012.

[45] Alex Graves, Marcus Liwicki, Santiago Fernandez, Roman Bertolami, Horst Bunke, and Jurgen Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(5):855–868, May 2009.

[46] Jurgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[47] Herbert Jaeger. A tutorial on training recurrent neural networks, covering bppt, rtrl, ekf and the "echo state network" approach. Technical report, Fraunhofer Institute for Autonomous Intelligent Systems (AIS), 2013.

[48] Razvan Pascanu, Çaglar Gülçehre, Kyunghyun Cho, and Yoshua Bengio. How to construct deep recurrent neural networks. *Computing Research Repository (CoRR)*, abs/1312.6026, 2013.

[49] Yoshua Bengio and Aaron Courville. Deep learning of representations. In Monica Bianchini, Marco Maggini, and Lakhmi C. Jain, editors, *Handbook on Neural Information Processing*, pages 1–28. Springer, 2013.

[50] Jurgen Schmidhuber. Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642–3649, 2012.

[51] Marc'Aurelio Ranzato. Supervised deep learning - tutorial on deep learning for vision. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, 2014.

[52] Andrew Ng, Jiquan Ngiam, Chuan Yu Foo, Yifan Mai, Caroline Suen, Adam Coates, Andrew Maas, Awni Hannun, Brody Huval, Tao Wang, and Sameep Tandon. Unsupervised feature learning and deep learning. Technical report, Stanford University, 2013.

[53] Wikipedia. Convolutional neural network, 2015. [The last modified on 15 June 2015].

[54] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

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

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

[57] Ilya Sutskever and Geoffrey E. Hinton. Learning multilevel distributed representations for high-dimensional sequences. In *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics, (AISTATS)*, pages 548–555, 2007.

[58] Geoffrey E. Hinton. Deep belief networks. *Scholarpedia*, 4(5):5947, 2009.

[59] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. *Computing Research Repository (CoRR)*, abs/1303.5778, 2013.

[60] Andrea Vedaldi and Karel Lenc. Matconvnet - convolutional neural networks for MATLAB. *Computing Research Repository (CoRR)*, abs/1412.4564, 2014.

[61] Pierre Sermanet, David Eigen, Xiang Zhang, Michael Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2014.

[62] MediaWiki. Incremental learning, 2015. [The last modified on 20 January 2015].

[63] Xin Geng and Kate Smith-Miles. Incremental learning. In StanZ. Li and Anil Jain, editors, *Encyclopedia of Biometrics*, pages 731–735. Springer US, 2009.

[64] Wikipedia. Normal distribution, 2015. [The last modified on 4 July 2015].

APPENDIX A. An Example of Network Configuration File for DNN_8

```
name: "CaffeNet"
layers {
  name: "data"
  type: DATA
  top: "data"
  top: "label"
  data_param {
    source: "path/to/training/set"
    backend: LMDB
    batch_size: 256
  }
  transform_param {
    mean_file: "path/to/meanfile"
    mirror: false
  }
  include: { phase: TRAIN }
}
layers {
  name: "data"
  type: DATA
  top: "data"
  top: "label"
  data_param {
    source: "path/to/validation/set"
    backend: LMDB
    batch_size: 50
  }
  transform_param {
    mean_file: "path/to/meanfile"
    mirror: false
  }
  include: { phase: TEST }
}
layers {
```

```
  name: "conv1"
  type: CONVOLUTION
  bottom: "data"
  top: "conv1"
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
  convolution_param {
    num_output: 12 # the number of filters
    kernel_size: 2 # the size of filters
    stride: 1
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
layers {
  name: "relu1"
  type: RELU
  bottom: "conv1"
  top: "conv1"
}
layers {
  name: "norm1"
  type: LRN
  bottom: "conv1"
  top: "norm1"
  lrn_param {
    local_size: 5
    alpha: 0.0001
    beta: 0.75
  }
```

```
}
layers {
  name: "fc6"
  type: INNER_PRODUCT
  bottom: "norm1"
  top: "fc6"
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
  inner_product_param {
    num_output: 512
    weight_filler {
      type: "gaussian"
      std: 0.005
    }
    bias_filler {
      type: "constant"
      value: 1
    }
  }
}
layers {
  name: "relu6"
  type: RELU
  bottom: "fc6"
  top: "fc6"
}
layers {
  name: "drop6"
  type: DROPOUT
  bottom: "fc6"
  top: "fc6"
  dropout_param {
    dropout_ratio: 0.5
  }
layers {
  name: "fc8"
```

```
  type: INNER_PRODUCT
  bottom: "fc6"
  top: "fc8"
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
  inner_product_param {
    num_output: 1000
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
layers {
  name: "accuracy"
  type: ACCURACY
  bottom: "fc8"
  bottom: "label"
  top: "accuracy"
  include: { phase: TEST }
}
layers {
  name: "loss"
  type: SOFTMAX_LOSS
  bottom: "fc8"
  bottom: "label"
  top: "loss"
}
```

APPENDIX B. An Example of Network Solver File

```
net: "path/to/network/configuration/file"
test_iter: 1000
test_interval: 1000
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 50000
display: 20
max_iter: 120000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix: "path/to/save/snapshot/file"
solver_mode: GPU
```