University of Cape Town

Masters Thesis

Human Action Recognition with 3D Convolutional Neural Networks

Author:
Frans Cronje

Supervisor:
Mr. Allan Clark

A thesis submitted in fulfilment of the requirements for the degree of Masters of Science

in the

Department of Statistical Sciences
Faculty of Science

November 2014

Declaration of Authorship

I, Frans Cronje, declare that this thesis titled, 'Human Action Recognition with 3D Convolutional Neural Networks' and the work presented in it are my own. I confirm that:

- I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is my own.
- I have used the APA referencing guide for citation and referencing. Each contribution to, and quotation in this dissertation from the work(s) of other people has been contributed, and has been cited and referenced.
- I know the meaning of plagiarism and declare that all of the work in the dissertation, save for that which is properly acknowledged, is my own.

Signed: _____

Date: _____

UNIVERSITY OF CAPE TOWN

Abstract

Faculty of Science
Department of Statistical Sciences

Masters of Science

Human Action Recognition with 3D Convolutional Neural Networks

by Frans Cronje

Convolutional neural networks (CNNs) adapt the regular fully-connected neural network (NN) algorithm to facilitate image classification. Recently, CNNs have been demonstrated to provide superior performance across numerous image classification databases including large natural images (Krizhevsky et al., 2012). Furthermore, CNNs are more readily transferable between different image classification problems when compared to common alternatives.

The extension of CNNs to video classification is simple and the rationale behind the components of the model are still applicable due to the similarity between image and video data. Previous CNNs have demonstrated good performance upon video datasets, however have not employed methods that have been recently developed and attributed improvements in image classification networks.

The purpose of this research to build a CNN model that includes recently developed elements to present a human action recognition model which is up-to-date with current trends in CNNs and current hardware. Focus is applied to ensemble models and methods such as the Dropout technique, developed by Hinton et al. (2012) to reduce overfitting, and learning rate adaptation techniques.

The KTH human action dataset is used to assess the CNN model, which, as a widely used benchmark dataset, facilitates the comparison between previous work performed in the literature. Three CNNs are built and trained to provide insight into design choices as well as allow the construction of an ensemble model. The final ensemble model achieved comparative performance to previous CNNs trained upon the KTH data.

While the inclusion of new methods to the CNN model did not result in an improvement on previous models, the competitive result provides an alternative combination of architecture and components to other CNN models.

Acknowledgements

To my parents, I am grateful for your unending, unassuming and unwaivering support. Without the knowledge of which, I would not have embarked upon this experience.

To Allan Clark, thank you for always being readily available, your continuous support and willingness to hear and add structure to my otherwise unstructured thoughts.

To Richard Craib, thank you for encouraging participation in the Galaxy Zoo competition and assistance with Random Forests in that regard.

Computations were performed using facilities provided by the University of Cape Town's ICTS High Performance Computing team: http://hpc.uct.ac.za. Thank you for your readily available support with the implementation of the models upon the GPU nodes.

The financial assistance of the National Research Foundation (NRF) towards this research is hereby gratefully acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.

Contents

List of Figures

List of Tables

Abbreviations

BHHH	**B**erndt **H**all **H**all (and) **H**ausman
CNN	**C**onvolutional **N**eural **N**etwork
DBD	**D**elta-**B**ar-**D**elta
GBM	**G**radient **B**oosted **M**odel
GPU	**G**raphics **P**rocessing **U**nit
GPGPU	**G**eneral-**P**rocessing (computing on) **G**raphics **P**rocessing **U**nit
HOG	**H**istogram (of) **O**riented **G**radients
IDBD	**I**ncremental **D**elta-**B**ar-**D**elta
ILSVRC	**I**mageNet **L**arge **S**cale **V**isual **R**ecognition **C**hallenge
LSTM	**L**ong **S**hort **T**erm **M**emory
MRI	**M**agnetic **R**esonance **I**mage
NN	**N**eural **N**etwork
PCA	**P**rincipal **C**omponent **A**nalysis
RF	**R**andom **F**orest
RGB	**R**ed **G**reen **B**lue
RMSE	**R**oot **M**ean **S**quare **E**rror
RNN	**R**ecurrent **N**eural **N**etwork
SDLM	**S**tochastic **D**iagonal **L**evenberg **M**arquardt
SGD	**S**tochastic **G**radient **D**escent
SIFT	**S**cale **I**nvariant **F**eature **T**ransform
UML	**U**nified **M**odeling **L**anguage
vSGD	**v**ariance-based **S**tochastic **G**radient **D**escent

Chapter 1

Introduction

1.1 Context

Image classification is a widely studied problem that has a large array of different applications. Images, while typically thought of as photographs from handheld cameras, arise from a number of sources, including telescopes and magnetic resonance imaging (MRI) devices. As the number of devices that capture images increases along with the rate at which images are captured, assistance is sought from automated systems to reduce the burden of processing the images. In the case of astronomy, improvement in hardware has allowed telescopes to capture very high resolution images that contain many objects such as galaxies within them. Subsequently the amount of image data is too large for experts to classify, and as a result automated classification systems are being developed (Banerji et al., 2010). Further applications include the automated house numbering of the Google Street View data (Goodfellow et al., 2014) and identifying roads from aerial photography (Mnih and Hinton, 2010).

In addition to assisting with classifying images that would already have existed, visual sensors can be developed by combining image classification with an image capturing hardware. In this regard practical applications include face recognition for access control (Wagner et al., 2012), object recognition can be applied to assisted driving systems such as traffic sign recognition (Cireşan et al., 2012a) or pedestrian recognition (Geronimo et al., 2010), handwritten digit recognition to read cheques (Le Cun et al., 1998), mitosis detection in breast cancer histology images (Cireşan et al., 2013) or ore segmentation (Mukherjee et al., 2009).

Video classification is similar to image classification and frequently image classification is performed upon single frames from video. For example, in the afore mentioned application of access control through face recognition, the input can be a video stream, however

each frame is considered in isolation. Here, the scope of video classification is confined to classification upon input that consists of more than one frame from a video sequence. This is usually necessary when the classification subject is of a changing position, e.g. a person performing an action.

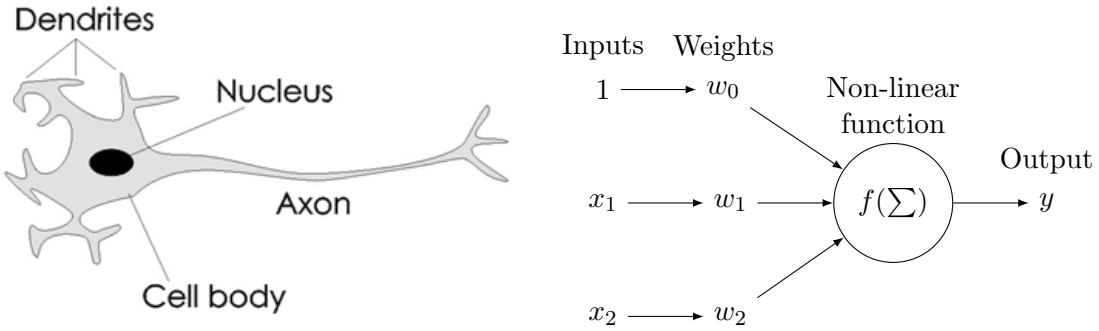
In practice, video classification is frequently applied to surveillance footage. Surveillance camera's are widely used to discourage or monitor areas that are usually at risk. The extent of their distribution and the mundanity of their output makes vigilant monitoring difficult. Subsequently, automated monitoring has been developed in order to classify violence (Nievas et al., 2011), human action in airports (Ji et al., 2013) or detection of humans falling (Rougier et al., 2011), to provide some examples.

Upon analysis, images are represented as one or more matrices of pixel values, each matrix encoding a separate colour channel. For example a 256×256 image composed of red green blue (RGB) pixels would have approximately 200,000 pixel values arranged in a $256 \times 256 \times 3$ matrix. A characteristic feature of image data is that objects within the images, such as a car or a person's face, influence a local region of values in the matrices. Videos are usually represented as a series of images. As a result the dimensionality of video data is usually very high and includes two spatial dimensions, a temporal dimension and can include a colour channel dimension. The very high dimensionality of the input presents one of the difficulties with image and video classification.

Artificial neural networks (NNs) were first conceived as attempts to algorithmically represent the biological brain (Bishop et al., 2006). Specifically the structure of the neurons, whereby the dendrites of a neuron connect via synapses, accumulating input from preceding neurons, the total of which is acted upon by the cell body (As illustrated in Figure 1.1a).

As a mathematical model, NN-like algorithms date back to the 1940s (Schmidhuber, 2014). The basic form of a neuron is a linear combination of inputs, each acted upon by an adjustable parameter, which is then acted upon by a non-linear function, as illustrated in Figure 1.1b. Neurons are usually arranged into successive layers, whereby neurons within a layer connect to neurons in the preceding and succeeding layers, but usually not amongst themselves. Layers between the input and output are referred to as hidden layers. A NN composed of a single hidden layer can be demonstrated to approximate any function in $\mathbb{R}^n$ with an arbitrary degree of accuracy (Hastie et al., 2009).

As an algorithm for classification and regression, NNs have demonstrated prowess in a wide range of tasks. Recent examples include audio and speech recognition (Lee et al., 2009; Graves et al., 2013) and natural language parsing (Socher et al., 2011). In the context of image classification, convolutional neural networks (CNN) employ a



(a) Biological neuron (Leverington, 2009) (b) Artificial neuron

Figure 1.1: In (a) dendrites connect to the axons of preceding neurons at synapses. When preceding neurons fire, the amount of charge passed on can be adjusted at the synapse. The cell body accumulates the charge from the dendrites and once sufficient charge is accumulated the neuron fires, passing on charge to succeeding neurons connected at it's axon. In the artificial neuron depicted in (b) inputs are combined with weights, multiplicatively. The result is accumulated and passed through a non-linear function. The output is then passed onto succeeding neurons.

network architecture adapted for images and have returned superior results across several different image datasets in recent years. This includes substantially improving upon the next best alternative for object recognition in regular images (Krizhevsky et al., 2012), attaining human-comparable performance upon handwritten digits (Wan et al., 2013) and outperforming humans ability upon traffic sign recognition (Cireşan et al., 2012a). Comparatively, less work has been performed upon video recognition. Yang et al. (2009) employed a CNN as part of a larger system upon a video surveillance dataset to recognise human action, which was then improved upon by Ji et al. (2013).

A large part of the recent success of NNs can be attributed to improvements in hardware, in particular the introduction of general purpose computing on the graphics processing unit (GPGPU), which has lead to the development of ensembles of very large NNs trained upon increasingly larger datasets (Schmidhuber, 2014). Prior to CNNs, image classification was typically performed using a specialised feature extraction method, such as scale-invariant feature transforms (SIFT) (Lowe, 1999) to reduce the raw input to lower dimensionality, before applying a classification technique such as support vector machines (SVM). Using GPGPU, CNNs can be enacted upon the raw input pixels of images, to develop a classifier that includes the feature extractor component. In addition, the CNN algorithm can readily be applied across different image classification tasks or datasets without substantial restructuring (Wan et al., 2013; Zeiler and Fergus, 2014).

Despite improvements in hardware, NNs are still constrained by the time taken to optimize their parameters (Cireşan et al., 2012b; Krizhevsky et al., 2012). Additionally, in many image recognition tasks the number of inputs is usually large when considering the number of cases that the network is optimized upon. As illustrated earlier, an image

can consist of 200,000 input values. As a result NNs are prone to overfitting due to the large number of free parameters necessary to receive the input and develop sufficient decision boundaries. The architecture constraints imposed by convolutions in CNNs assist in reducing overfitting, and are usually assisted by more traditional regularization techniques such as including a L1 or L2 weight penalty term. More recently, improvements have been achieved upon image recognition datasets by new regularization techniques (Hinton et al., 2012; Wan et al., 2013).

1.2 Purpose of Research

The purpose of the work presented here is to build a 3D CNN model that includes recently developed elements from successful 2D CNNs to present a human action recognition model which is more up-to-date with current trends. As a starting basis, the model presented by Ji et al. (2010) (and then later republished with further results in Ji et al. (2013)) was used to inform initial design choices, thereafter new components were informed by recent successes in image recognition with 2D CNNs.

CNNs, which have been demonstrated to provide superior results upon image classification, have not been widely applied to video classification. Furthermore, as many of the recent improvements are not constrained to image recognition and given the similar nature of video and image data, it is expected that the recent methods will garner a similar improvement in video classification using a CNN.

1.3 Overview

This dissertation is composed of 10 chapters, including the current chapter, which serves to introduce and provide the motivation behind the work. The following chapter details the necessary theory of the general, fully-connected, feedforward NN, including the gradient descent and backpropagation algorithm which informs how parameters are updated. Chapter 3 discusses the key problems with image classification as well as how these problems have been addressed by image classification algorithms in general. The chapter concludes with an explanation of the CNN theory and the rationale behind the design constraints that are imposed in order to improve their classification capability of images. The succeeding chapter provides a brief exposition as to CNNs' developmental history from the initial biological inspiration to the most recent trends in their development. The chapter introduces several popular image classification datasets and discusses different CNNs that have been developed and their respective results. Different methods

of informing the rate at which parameters are optimized are investigated in Chapter 5. Chapter 6 explains the improvement realised by ensemble methods as well as recent efficient ensemble methods that have been employed to produce superior performance in image classification. Chapter 7 presents a CNN built to identify morphological features in galaxies from images, as presented by the Galaxy Zoo competition (Willett et al., 2013). The succeeding chapter very briefly examines the extension of 2D CNNs to 3D CNNs and examines the work done by Ji et al. (2010) in this regard. Chapter 9 presents the methodology and results achieved by the 3D CNN built to achieve the purpose of this dissertation, it includes a discussion on the architectures that were used. Chapter 10 concludes the work.

Chapter 2

Basic Neural Network Theory

2.1 Introduction

An artificial NN consists of nodes (otherwise referred to as neurons) and directed edges between nodes which are typically depicted as per Figure 2.1. Neurons perform a linear or non-linear mapping of a weighted combination of inputs onto the real number domain. In turn, this output is then used as an input to further neurons. Neurons are arranged into layers with the preceding layers providing inputs to the current layer, which then provides input to the following layer. A NN will receive the initial data through an input layer. The information will be passed through the NN, each layer acting upon it, until the outputs are returned in the final layer. Typically, NNs are used to perform regression or classification tasks. The nature of the specific task will define the inputs, outputs and architecture of the network. For example, in an image classification task, inputs are commonly the grayscale intensity value for each pixel, taking on values between 0 and 255 and the output of the network is usually a 1-of-K classification type.

The structure of the NN and the mapping function of each neuron (which is referred to as the activation function) are usually defined at the outset of the construction of the network. The network is then 'taught' how to classify inputs by adjusting the weights of the connections (otherwise referred to as the free parameters) to improve an error value that is computed by a predefined error function. Weight updates are informed by the gradient of the error function and are assessed using a set of labelled 'training' cases.

In the following section, the neuron and the activation function are discussed, in addition to common activation functions for the hidden and output neurons of a NN. In Section 2.3 training of a NN is discussed including the definition of common error functions and gradient descent and backpropagation algorithm used to inform weight updates in a NN. Much of the work and notion presented here is informed by Bishop et al. (2006).

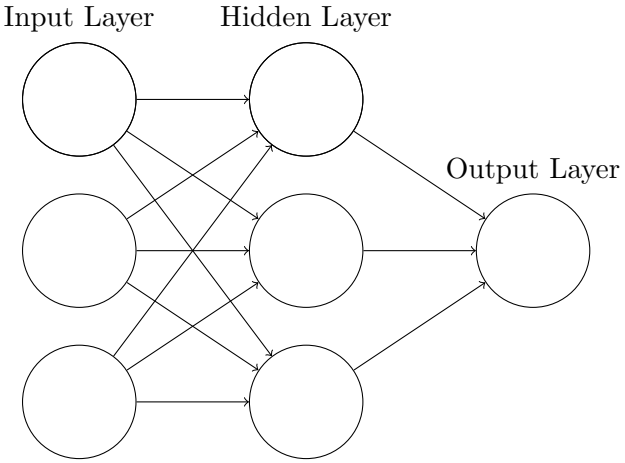


Figure 2.1: A simple two layered NN consisting of three inputs, a hidden layer with three neurons and an output layer with single neuron. When describing the number of layers, the number of layers of trainable weights are counted.

2.2 The Functional Form of Artificial Neural Networks

In this section the functional form of a NN is discussed, beginning with an introduction to neurons and the parameterization of their associated activation functions as well as notation that will be used throughout this research. The section concludes with the complete parameterization of a forward pass through a NN, referred to as forward propagation.

2.2.1 The Artificial Neuron

An artificial neuron computes an output value for a given set of input values. More specifically, a neuron consists of an activation function that acts upon a linear combination of the neuron's input values $\mathbf{x}^{(i)} = (x_1^{(i)}, x_2^{(i)}, \ldots, x_n^{(i)})$, weights associated with with the input values, $\mathbf{w}_1^{(i,j)} = (w_{1,1}^{(i,j)}, w_{2,1}^{(i,j)}, \ldots, w_{n,1}^{(i,j)})$ and an optional bias term, $w_{0,1}^{(i,j)}$. For example, the output of a neuron in the hidden layer in Figure 2.1 would be described by:

$$x_1^{(1)}(\mathbf{x}^{(0)}, \mathbf{w}^{(0,1)}) = f\left(\sum_{s=1}^{3} w_{s,1}^{(0,1)} x_s^{(0)} + w_{0,1}^{(0,1)}\right)$$

where:

- $w_{s,r}^{(i,j)}$ is the weight associated with the output from the s^{th} neuron in layer i that is received by the r^{th} neuron in layer $j = i + 1$. The matrix of weights from the i^{th} layer to the j^{th} layer is denoted $\mathbf{W}^{(i,j)}$.

- $w_{0,r}^{(i,j)}$ is the bias weight associated with the output from the neuron in layer i that is received by the r^{th} neuron in layer j. The bias unit can be included in the summation when defining $x_0^{(i)} = 1$.
- $x_s^{(i)}$ is the output from the s^{th} neuron in the i^{th} layer, $\mathbf{x}^{(i)}$ denotes the vector of outputs from the i^{th} layer.
- $f(\cdot)$ is the activation function of the neuron.

The weights $\mathbf{W} = (\mathbf{W}^{(0,1)}, \mathbf{W}^{(1,2)}, \ldots, \mathbf{W}^{(n-1,n)})$ are considered free parameters and are adjusted to correctly classify the inputs in the learning process. The linear combination of inputs to a neuron is commonly referred to as the activity of a neuron, e.g. $a_r^{(j)} = \sum_{s=0}^{S} w_{s,r}^{(i,j)} x_s^{(i)}$ is the activity of the neuron r in layer j. Figure 2.2 illustrates an artificial neuron.

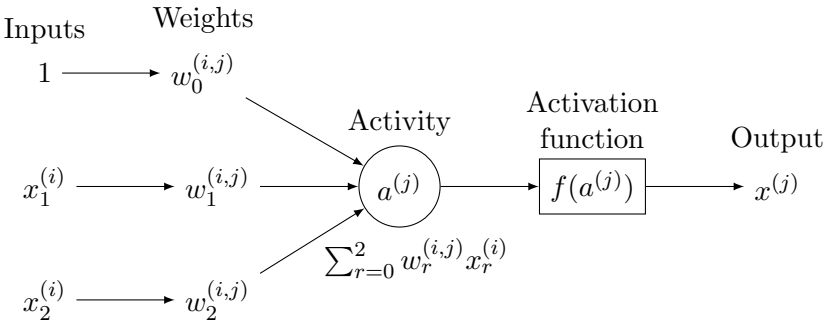


Figure 2.2: An illustration of the components of an artificial neuron.

2.2.2 Activation Functions

Described below are some typically used activation functions. The choice of activation function used is influenced by the type of network and the learning algorithm used.

Linear neuron:

$$f(a) = a$$

The linear neuron is commonly used as an output unit in regression tasks, but not as a hidden unit, as stacking linear combinations conveys no additional discriminative ability to the network. This is because a linear combination of a linear function can be defined as a linear function itself.

Sigmoidal neuron:

$$f(a) = \frac{1}{1 + e^{-a}} \tag{2.1}$$

$$f(a) = \tanh(a) \tag{2.2}$$

$$f(a) = \frac{a}{1 + |a|} \tag{2.3}$$

The logistic sigmoid neuron (Equation 2.1) was historically used to smooth mapping onto the $[0, 1]$ domain, facilitating the interpretation of the output as a probability which is helpful in a classification context. More recently and commonly the hyperbolic tangent (Equation 2.2, (Le Cun et al., 1998)) and softsign (Equation 2.3, (Glorot et al., 2011)) functions are used, resulting in a similarly shaped function that map onto the $[-1, 1]$ domain. The sigmoid unit has an asymptote at zero, i.e. as the input tends to zero the gradient of the sigmoid tends toward zero as well. This saturation can prevent learning occurring in preceding layers (Glorot and Bengio, 2010), subsequently the hyperbolic tangent and softsign function are preferred to the logistic sigmoid function. Glorot and Bengio (2010) found that the softsign function is more robust to initial starting condition than the hyperbolic tangent function and would train faster because its more gradual gradient in the tails. See Figure 2.3 (a), (b) and (c) for the respective illustrations of the logistic sigmoid, hyperbolic tangent and softsign activation functions and their gradients.

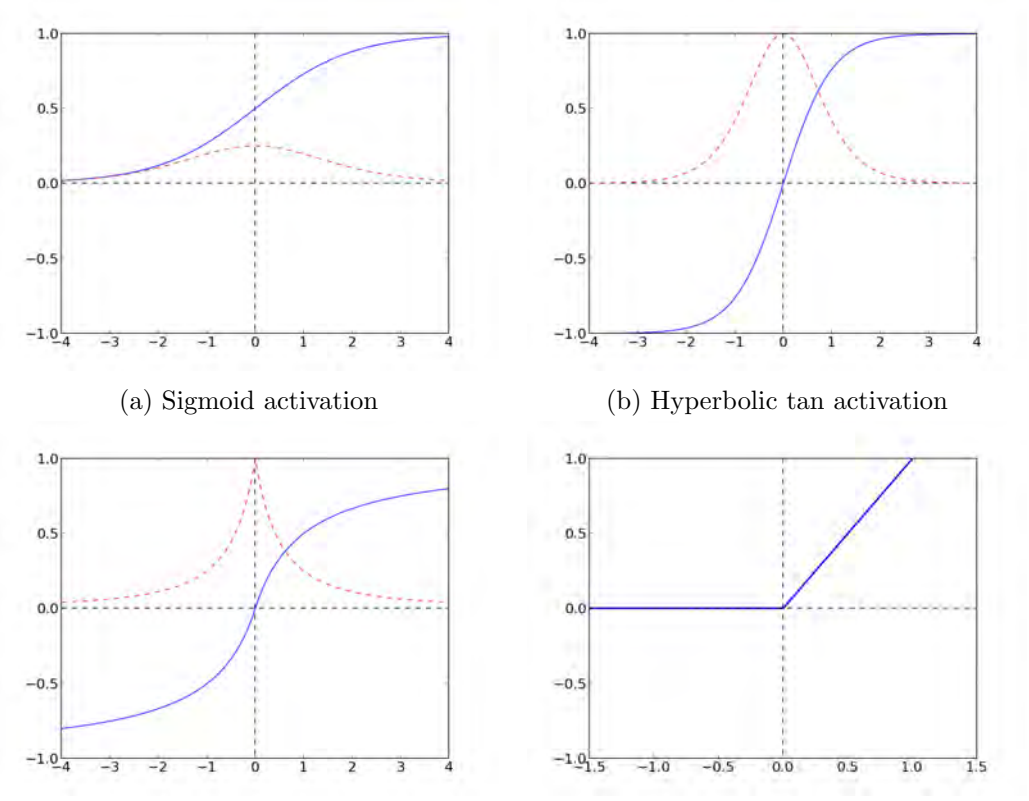
Rectified Linear neuron:

$$f(a) = \begin{cases} a & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

Recently, the rectified linear unit has achieved empirical success (Krizhevsky et al., 2012). Practically, the derivative is faster to compute and has been demonstrated to increase the speed of learning (Krizhevsky et al., 2012). Furthermore, the hard zero, which results in a large number of neurons being 'off' for any one case is more biologically plausible and has been experimentally demonstrated to improve the accuracy of NNs, contrary to intuition (Glorot et al., 2011). See Figure 2.3 (d) for an illustration of the rectified linear unit.

2.2.3 Output Units

Output units are defined by the classification or regression task that the NN has been built for. If the NN is being used for a regression task, commonly linear output neurons are used. In the case of a single or multi-class binary classification task, logistic sigmoid neurons are used such that the output of the neurons can be interpreted as a probability. In the case of a multi-class 1-of-K classification task the softmax output unit is used for which the output values fall in the range $[0, 1]$ and sum to one, allowing the output to be collectively interpreted as the probability of each class occurring. The softmax output



(a) Sigmoid activation

(b) Hyperbolic tan activation

(c) Softsign activation

(d) Rectified Linear Unit activation

Figure 2.3: Activation functions (Solid blue line) and their respective gradients (Dashed red line)

unit for class k is defined as:

$$g_k(\mathbf{x}) = \frac{\exp(x_k)}{\sum_{s=1}^{K} \exp(x_s)} \quad k = 1, 2, \ldots, K$$

Note that the output activation functions are denoted as $g(\cdot)$, to distinguish them from hidden layer activation functions.

2.2.4 Feed Forward Parameterization

Combining the above, a forward pass, referred to as forward propagation, through the NN calculates the output of the network for a given set of inputs, $\mathbf{x^{(0)}}$, and weights, $\mathbf{W}$, as depicted in Figure 2.1 and is defined as follows:

$$y(\mathbf{x}^{(0)}, \mathbf{W}) = g\left(\sum_{r=0}^{2} w_r^{(1,2)} f\left(\sum_{s=0}^{2} w_{s,r}^{(0,1)} x_s^{(0)}\right)\right)$$

2.3 Parameter Learning

In order to fit a NN to a task, a method of tuning the free parameters must be adopted. Commonly, gradient descent, adapted to the NN context, is employed. This necessitates a set of training examples and an error function.

Given labelled training data, the NN can be trained in a supervised manner by minimising the error of the network's outputs when compared to the labels. In contrast, NNs can be trained in an unsupervised manner with unlabelled data. While substantial research has been performed upon unsupervised training (Hinton and Salakhutdinov, 2006; Le et al., 2011), it is not included in the scope of this research. Further training regimes make use of labelled and unlabelled data in a semi-supervised manner or employ labelled data from similar datasets to pretrain the network under the assumption that there are commonalities to the defining features in the data (Zeiler and Fergus, 2014). These regimes are not included in the scope of the research either.

In the following subsection, different error functions are discussed. Thereafter gradient descent and backpropagation, the method to inform weight updates in hidden layers, are presented.

2.3.1 Error Functions

As mentioned above, to update the parameters in an informed manner an error function must be predefined. In this subsection, using a probabilistic interpretation of the output of NNs, the error function is presented in the context of maximising the likelihood function (Bishop et al., 2006). Typically in NN literature this is conveyed equivalently as minimising an error function. The output of the NN will inform the error function used. Below the negative log likelihood is presented for the linear, sigmoidal and softmax output neurons.

2.3.1.1 Linear Outputs

The linear output unit is used in the case of regression, where the goal is to predict a continuous output or target value t, given a set of inputs, $\mathbf{x}$. Assuming that the target value t is distributed about function $y(\mathbf{x}, \mathbf{W})$ according to a Gaussian distribution, i.e.:

$$t = y(\mathbf{x}, \mathbf{W}) + \epsilon$$

where $\epsilon \sim N(0, \sigma^2)$. Such that:

$$p(t|\mathbf{x}, \mathbf{W}) = N(t|y(\mathbf{x}, \mathbf{W}), \sigma^2)$$

Subsequently for a training set of N independent and identically distributed inputs $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N)$ and respective target values $\mathbf{t} = (t_1, t_2, \ldots, t_N)$, the likelihood and negative log likelihood can be constructed:

$$p(\mathbf{t}|\mathbf{X}, \mathbf{W}, \sigma) = \prod_{n=1}^{N} p(t_n|\mathbf{x}_n, \mathbf{W}, \sigma^2)$$

$$-\ln p(\mathbf{t}|\mathbf{X}, \mathbf{W}, \sigma) = \frac{1}{2\sigma^2} \sum_{n=1}^{N} (y(\mathbf{x}_n, \mathbf{W}) - t_n)^2 + \frac{N}{2} \ln \sigma^2 + \frac{N}{2} \ln(2\pi)$$

Maximising the likelihood function is equivalent to minimising the negative log likelihood function. As a result, to find the set of parameters, $\hat{\mathbf{W}}_{ML}$, that maximise the likelihood, the sum-of-squares error function is minimized:

$$E(\mathbf{W}) = \frac{1}{2} \sum_{n=1}^{N} (y(\mathbf{x}_n, \mathbf{W}) - t_n)^2$$

which is equivalent to the negative log likelihood function with the components not relevant to $\mathbf{W}$ discarded. Once the maximum likelihood $\hat{\mathbf{W}}_{ML}$ is found $\hat{\sigma}_{ML}$ can be estimated as follows:

$$\sigma^2_{ML} = \frac{1}{N} \sum_{n=1}^{N} \Big(y(\mathbf{x}_n, \hat{\mathbf{W}}_{ML}) - t_n \Big)^2$$

2.3.1.2 Sigmoidal Outputs

For the multiple binary classification problem a sigmoidal function is used, most commonly the logistic sigmoid function. Here, the likelihood for a single logistic sigmoid output is first considered, before generalising the likelihood to the multiple output case. The target t can take one of two classes C_1 or C_2, and the input $\mathbf{x}$ is mapped to the $[0, 1]$ domain, as such the probability associated with the classes can be defined as $p(t = C_1) = y(\mathbf{x}, \mathbf{W})$ and $p(t = C_2) = 1 - y(\mathbf{x}, \mathbf{W})$. t can then be represented by the Bernoulli distribution:

$$p(t|\mathbf{x}, \mathbf{W}) = y(\mathbf{x}, \mathbf{W})^t (1 - y(\mathbf{x}, \mathbf{W}))^{(1-t)}$$

The likelihood function and subsequently the negative log likelihood function for N independent and identically distributed cases can be constructed as follows:

$$p(\mathbf{t}|\mathbf{X},\mathbf{W}) = \prod_{n=1}^{N} y(\mathbf{x}_n,\mathbf{W})^{t_n}(1-y(\mathbf{x}_n,\mathbf{W}))^{(1-t_n)}$$

$$-\ln p(\mathbf{t}|\mathbf{X},\mathbf{W}) = -\sum_{n=1}^{N}\big(t_n \ln y(\mathbf{x}_n,\mathbf{W}) + (1-t_n)\ln(1-y(\mathbf{x}_n,\mathbf{W}))\big)$$

The negative log likelihood, interpreted as an error function, has the same form as the cross entropy error function (Bishop et al., 2006). The single sigmoid output can be extended to K logistic sigmoid outputs to perform multiple classification upon a pattern. Assuming class labels are independent given inputs values, the following likelihood function is arrived at:

$$p(\mathbf{t}|\mathbf{X},\mathbf{W}) = \prod_{n=1}^{N}\prod_{k=1}^{K} y_k(\mathbf{x}_n,\mathbf{W})^{t_{k,n}}(1-y_k(\mathbf{x}_n,\mathbf{W}))^{(1-t_{k,n})}$$

The subsequent error function, which is equivalent to the negative log likelihood function, is then defined as:

$$E(\mathbf{W}) = -\sum_{n=1}^{N}\sum_{k=1}^{K}\big(t_{k,n}\ln y_k(\mathbf{x}_n,\mathbf{W}) + (1-t_{k,n})\ln(1-y_k(\mathbf{x}_n,\mathbf{W}))\big)$$

2.3.1.3 Softmax Outputs

Softmax output unit is used upon the conventional classification problem where the target is a 1-of-K class, i.e. $\mathbf{t} = (t_1, t_2, \ldots, t_K)$ such that $t_i \in \{0,1\}\ \forall\ i = 1,\ldots,K$ and $\sum_{i=1}^{K} t_i = 1$. The output of the network can then be interpreted as $p(t_{k,n} = 1|\mathbf{W},\mathbf{x}_n) = y_k(\mathbf{x}_n,\mathbf{W})$, given the softmax's properties. The likelihood and error function, derived from the negative log likelihood, are then defined as follows:

$$p(\mathbf{t}|\mathbf{X},\mathbf{W}) = \prod_{n=1}^{N}\prod_{k=1}^{K} y_k(\mathbf{x}_n,\mathbf{W})$$

$$E(\mathbf{W}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{k,n}\ln y_k(\mathbf{x}_n,\mathbf{W})$$

2.3.2 Parameter Optimization

Weights need to be adjusted accordingly to minimize the error function of the network. This task is complicated by the non-convex error-weight surface resulting from

the non-linear dependence between weights and the error, as such multiple local and global minima can exist, many of which are induced by weight symmetries resulting in equivalent outcomes. Furthermore, in general it is not possible to determine whether the minima is local or global. Commonly, under these circumstances, iterative weight updates that make use of gradient information are used to gradually move the weight set towards a lower error value. In a NN context, simple gradient descent is the most widely employed method of doing so. Gradient descent involves moving gradually in a direction opposite the gradient of the error function in the error-weight space using small steps that are proportional to the current error gradient with respect to the weight. Weights in hidden layers are not directly connected to the error function and as a result to evaluate their error gradient, an implementation of the chain rule, referred to as backpropagation is used.

In this section gradient descent and backpropagation are presented in the context of NNs.

2.3.2.1 Gradient Descent

The gradient descent algorithm gradually updates the weight parameters by taking small steps in the negative direction of the gradient of the error function in the weight space. i.e.:

$$\mathbf{W}^{(\tau+1)} = \mathbf{W}^{(\tau)} + \Delta\mathbf{W}^{(\tau)}$$

$$\Delta\mathbf{W}^{(\tau)} = -\eta\nabla E(\mathbf{W}^{(\tau)})$$

where $\nabla E(\mathbf{W}^{(\tau)})$ is the gradient of the error function with respect to $\mathbf{W}$, η is termed the learning rate and τ is the weight update index. At a minimum $\nabla E(\mathbf{W}) = 0$ and learning will stop. As it is not possible to determine whether a minimum is global or local, multiple restarts with randomized initial weights are commonly employed to find the best weight combination, see Chapter 6 for further detail.

The learning parameter η is a pre-defined hyperparameter. Caution needs to be taken when defining the learning rate as it will influence the speed at which the network converges. If the learning parameter is defined to be very small, then the network will converge very slowly, and may make training until convergence impractical. Conversely, if the learning parameter is defined to be very large, successive weight updates can oscillate out of minimum bowls if they are approximately quadratic. The learning rate can alternatively be adaptive, rather than constant. For example, if a network has a single trainable parameter and $\eta = (\frac{\partial^2 E}{\partial w^2})^{-1}$, then gradient descent is optimal if the error surface is a quadratic bowl (Le Cun et al., 2012). Chapter 5 discusses choices of the

different learning rate adaptation techniques and their subsequent effect upon the speed of learning.

For a given set of training data, different approaches with regard to the number of cases to be evaluated prior to updating the weights can be taken. Either, full batch learning can be applied, whereby the average error for the entire dataset can be used to inform the next weight update, or online learning can be applied, whereby each weight update is informed by the estimated error derivative calculated from a single random training case. Online learning includes stochasticity as the gradient is informed by an error surface estimated from a single random case. Stochastic gradient descent (SGD) allows the model to escape local minimums whereas full batch learning will converge to the local minimum closest to the initial weights. However, due to the noisy updates, SGD will not completely converge, whereas full batch learning shall. In this regard, annealing schedules for the learning rate are recommended in order to reduce the fluctuations of stochastic gradient descent (I.e. $\eta(\tau) \propto \frac{1}{\tau}$). In practice it is common to apply mini-batch learning whereby weights are updated according to the average error gradient of a random subset of the training cases. This reduces the noise of online learning updates while still allowing the network to escape local minimums.

Furthermore, in the context of highly redundant data, SGD will accelerate learning as a subset of cases can be representative of the underlying components in the data and will be faster to compute between weight updates. For an extreme example of this, consider a training dataset that contains many duplicates, when the error gradient is averaged prior to weight updates, the duplicates will not add additional information over a representative subset of cases (Le Cun et al., 2012).

The number of cases included in a mini-batch is a predefined hyperparameter. Commonly, it is constrained by hardware rather than optimized upon the specific data through the use of a validation dataset.

2.3.2.2 Backpropagation

Gradient descent is directly applicable to weights in the final layer of a network, however, incoming weights to hidden neurons are not directly connected to the error term. In this case the chain rule is used to evaluate the partial derivative of the error with respect to the weight. In the context of NNs, this application of the chain rule is referred to as backpropagation and can be described in the following steps and is illustrated in Figure 2.4.

1. First, a forward pass is made determining the error value and the outputs at each layer are saved.

2. The error derivative with respect to each weight is calculated, using the error derivative with respect to each of the activities.

$$\frac{\partial E}{\partial w_{1,1}^{(i,j)}} = \frac{\partial E}{\partial a_1^{(j)}} \frac{\partial a_1^{(j)}}{\partial w_{1,1}^{(i,j)}} = \frac{\partial E}{\partial a_1^{(j)}} x_1^{(i)} \tag{2.5}$$

   If the j is the output layer, then $\frac{\partial E}{\partial a_1^{(j)}}$ can be directly evaluated. Otherwise, the error derivative with respect to $a_1^{(j)}$ is evaluated according to the following step.

3. The error derivative with respect to the activities for the hidden units is determined from the error derivative with respect to activities in the layer following it.

$$\frac{\partial E}{\partial a_1^{(j)}} = \frac{\partial x_1^{(j)}}{\partial a_1^{(j)}} \frac{\partial E}{\partial x_1^{(j)}} = f'(a_1^{(j)}) \sum_{q=0}^{2} w_{1,q}^{(j,k)} \frac{\partial E}{\partial a_q^{(k)}} \quad \text{where} \quad k = j + 1 \tag{2.6}$$

   It is from this step that the algorithm earns its name, the error derivative is propagated backward through the network to determine the portion of the error value attributable to each weight.

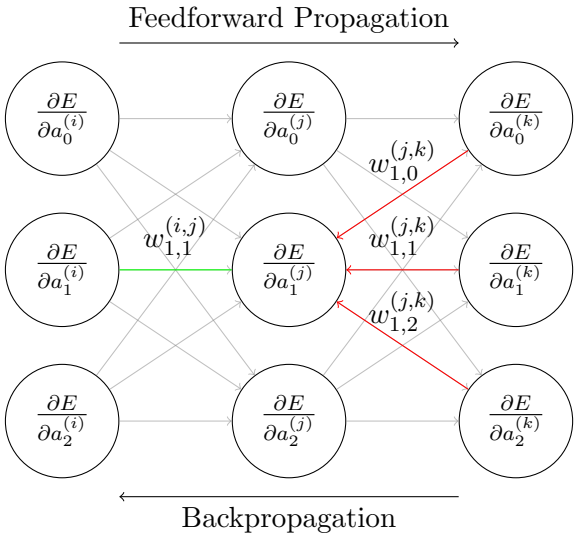


Figure 2.4: To find the partial error derivative with respect to the activity in layer j, the partial error derivatives with respect to the activations in the following layer, k, are backpropagated, as indicated by the red arrows. Then the partial error derivative with respect to $w_{1,1}^{(i,j)}$ can be evaluated (green line).

Chapter 3

2D Convolutional Neural Networks

3.1 Introduction

Image classification is complicated by the typically high dimension of input and substantial variation in appearance of the object of interest within an image. For example a 256×256 RGB image contains approximately 200,000 values or an object viewed from two different perspectives can have a substantially different numerical representation in the image. To classify images robustly, a classification algorithm must have the capability to receive high dimensional inputs and be robust to substantial variation inherent to image data. Usually, these two tasks are combined in a process called feature extraction which reduces the dimensionality while maintaining discriminative, invariant features in the image. Historically, feature extraction techniques have been hand developed such as the Scale Invariant Feature Transform(SIFT) (Lowe, 1999) or the use of a Histogram of Oriented Gradients (HOG) (Dalal and Triggs, 2005) features. Thereafter the extracted features are classified using a regular classification technique such as SVMs (Cortes and Vapnik, 1995).

Deep, fully-connected NNs are not well suited to image classification when using the raw pixels as input. The high dimensionality requires NNs that are burdensome upon current hardware. Furthermore, given the very large number of free parameters that would arise in a deep, fully-connected NN, without a very large training dataset, strict regularization must be applied to avoid overfitting.

Inspired by biological NNs, CNNs (Le Cun et al., 1998) take advantage of characteristics of image data to improve tractability and regularize the network. Specifically,

local feature detectors are employed to reduce the number free parameters and weight sharing is introduced to regularize the network as well as improve the CNN's robustness to the variance occurring in images. The hierarchical structure of a CNN further improves upon the network's robustness to variance and allows the algorithm to develop successively more complex representations of the image in a manner similar to biological NNs (Fukushima et al., 1983). CNNs are advantaged over typical image classification algorithms as their feature extraction is not hand developed but rather automatically trained according to the training data in conjunction with the classifier. As a result the CNN algorithm is more readily transferable between different image classification tasks.

Section 3.2 discusses the difficulty of image classification in further detail. In Section 3.3 different approaches to classifying images are discussed. Thereafter Section 3.4 presents the parameterization and intuition behind CNNs.

The image classification task referred to in this chapter involves classifying an image according to the main subject of the image. This task does not assume localization, nor is centring and size standardization of the region of interest across images assumed.

3.2 The Image Classification Problem

Very high dimensional data and high natural variance between items of the same category are the most significant difficulties associated with image classification. Each of these are discussed in this section.

3.2.1 High Dimensions

One of the challenges presented by image data is the very high dimensionality of image data. For example, the images in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) 2012 (See Figure 3.1) were commonly, approximately 500×400 RGB pixels large, giving a total of 600,000 values that describe the image completely when taking into account each of the red, green and blue intensity values describing every colour pixel.

Current available hardware is not sufficiently powerful enough to manipulate large amounts of high dimensional data. The most recent efficient implementations of NNs can receive a raw input of up to approximately 150,000 values, after which they quickly become impractical to implement and train (Krizhevsky et al., 2012).

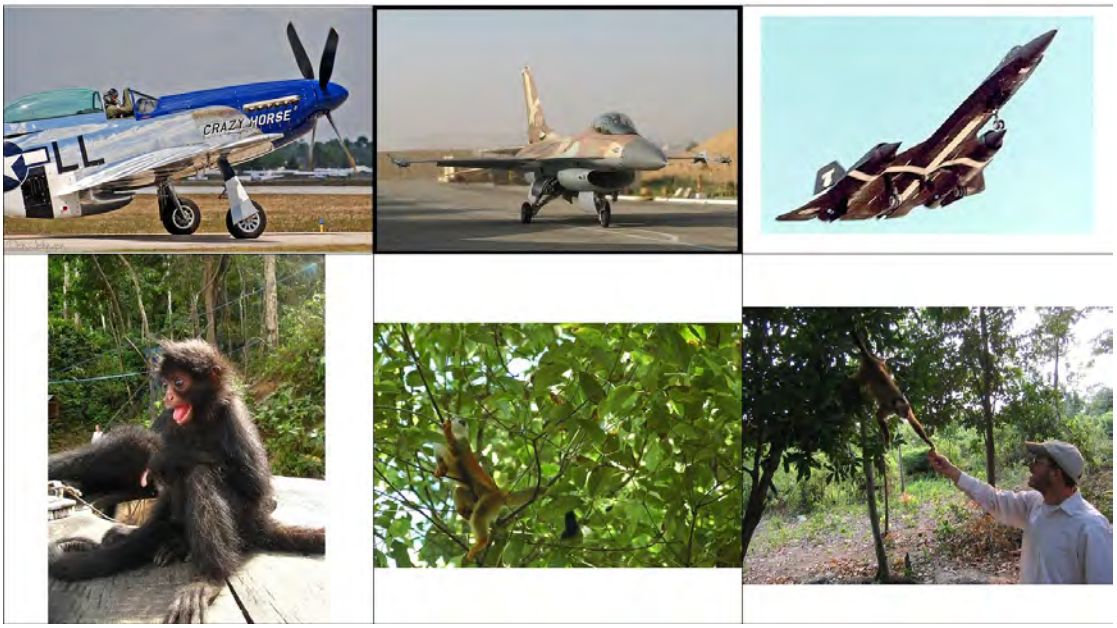


Figure 3.1: Examples of two categories of images from the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) 2012, the first row includes images labeled 'Interceptor', the second row includes images labeled 'Spider monkey' (Deng et al., 2009).

3.2.2 Variance in Natural Images

There is often substantial amount of variance between images of objects belonging to the same class. Consider examples of images in Figure 3.1 from two object categories portrayed in the ILSVRC 2012. This section examines several different sources of variation that arise in image data naturally.

Different approaches can be taken when categorising variance in images. Here, variation is considered according to the source from which it arises. Variation due to the image capture device, it's relative position to the object of interest and it's environment are considered together. Variation due to the object of interest itself are considered separately. More concretely, images of a uniform, non-deformable object would only consist of variance from the former category and could include the following:

- **Pose:** Different viewing angles of a 3D object will result in substantially different representations of the object in the pixel data.
- **Illumination:** The amount of ambient light will influence pixel values in the entire images. Furthermore, depending on the position of the lighting to the object shadows can create variance in local image features, such as a shadow cast by a nose obscuring facial features.
- **Shift:** The object of interest in an image is not necessarily centred upon.

- **Occlusion:** Objects can be partially obscured by items between the object and the camera. Alternatively, the object may be partially cropped out of the image.
- **Scale:** Commonly natural objects have different sizes, additionally distance between the camera and the object are not controlled. As a result the object of interest can be different size between images.
- **Clutter:** Natural scenes usually contain background objects that are not the focus of the image.

This list is by no means exhaustive and serves to demonstrate that there are many sources of variation due to the environment and position of capture.

The second category considers variance introduced by the object. Within this category there exists two types of variation. First, object classification can be performed upon generic object categories rather than unique objects, which is usually the case in image classification. As an example, the 'interceptors' category in Figure 3.1 includes aeroplanes that can have a propeller or a jet engine. This adds complexity to the image classification task, as classes can be abstract such as the 'interceptor' class which describes a small fighter aeroplane but not the exact features. Secondly, some objects can change form. For example recognising humans is a difficult task as, in different images, they are likely to have different stances such as sitting or walking. For example consider the images of a monkey in different poses in Figure 3.1.

3.3 Algorithms for Image Classification

There are two general approaches to handling high dimensional data. Dimensionality reduction can be performed upon the data prior to classification or the classification algorithm must be efficient enough to handle high dimensions, reducing the dimensionality implicitly. While either approach can be used with NNs, the focus of this research is upon using the full input rather than a reduced input. In this section both approaches are discussed.

There are numerous well established methods to reduce the dimensionality of the input prior to classification. These can be grouped into two categories, the first group entails using theoretically rigorous, dimensionality reduction techniques, such as principal component analysis (PCA), which was developed separately from image classification. The second group includes heuristic image specific reduction techniques which exploit features found to be useful in classifying images while remaining robust to variance in the image, such as the HOG feature extractor (Dalal and Triggs, 2005).

In the first category, PCA is the most commonly applied method. Combined with a simple classifier such as k-Nearest Neighbours, PCA performed successfully upon face detection and recognition tasks when in a controlled environment (Turk and Pentland, 1991; Kshirsagar et al., 2011). However when variance is included in the image, PCA does not assist the classification algorithm. Upon the NORB image dataset (See Figure 4.5b) which deliberately includes pose, lighting and centring variance, PCA provided no significant benefit to a SVM classifier when compared to the use of a lower resolution of the original image (Le Cun et al., 2004). This serves to demonstrate that PCA captures useful information when variation is primarily due to the object of interest, however it does not in an uncontrolled environment with natural variance occurring.

The alternative approach to reducing dimensionality prior to classification is to use methods that have been purposely built for dimensionality reduction of images. These methodologies reduce the image to a set of features that have been found to be discriminative in image classification and robust against variances occurring in the image. HOG and SIFT feature extractors are examples of such a methods which have proven to be helpful in image classification (Dalal and Triggs, 2005; Lowe, 1999). Up until recently, using these methods in a pre-processing step was common prior to classification (Le Cun et al., 2004). This approach moves away from the goal of enabling machine learning algorithms to interpret low-level data by explicitly imbuing the algorithm with expert knowledge (Bengio et al., 2013).

3.3.1 Deep Neural Networks for Image Classification

To build and train a deep, fully-connected NN upon large images to perform image classification would result in very large number of trainable parameters, especially in the first layer that connects with the raw pixel input due to the number of pixels commonly found in a large image. Training a fully-connected NN with many trainable parameters would require a very large training dataset due to the networks large capacity and would also be burdensome upon the memory of the hardware (Le Cun et al., 1998). The depth and breadth of recent CNNs fitted to large image datasets, which require less memory than a fully-connected NNs, are still constrained by the memory of the hardware components (Krizhevsky et al., 2012).

In order to improve the tractability of a network with high dimensional inputs, the number of connections can be reduced, however this must be done in a careful and deliberate manner as the architecture of the network will impact the networks ability to generalize (Le Cun et al., 1990b). Large networks with many parameters require very large training datasets otherwise they will overfit and subsequently generalize poorly.

Cireşan et al. (2010) successfully trained deep fully-connected networks by augmenting the training set with mild distortions to the input image to create a theoretically infinitely large dataset. The network's input of 32×32 grayscale images was sufficiently small for a fully-connected NN to be applied in this situation. While data augmentation has proven helpful at regularising networks (Krizhevsky et al., 2012) it does not address the tractability issue associated with high dimensions.

An alternative approach would be to remove parameters from the network, reducing its capacity for overfitting and improving tractability of the network. If done in a deliberate manner this approach will only slightly decrease the theoretically-best performance of the network (Krizhevsky et al., 2012). Thus the principle behind encoding sparse networks is to improve tractability and generalization by minimising the number of parameters without substantially reducing the ability of the network (Le Cun et al., 1990b).

Parameters can be removed in either a post-training informed manner or a pre-training manner. A simple approach to the former method is to remove the smallest parameters first, as they have the least impact upon the network if the input has been standardised. The network would then be retrained and the next smallest parameter would be removed until a suitable trade-off between network complexity and training error has been achieved (Le Cun et al., 1990b). A disadvantage of pruning methods is that the learning process speed is substantially slowed as the network needs to be retrained after removing parameters. Subsequently, pruning is not used upon high dimensional image classification NNs due to practical training time limitations, rather parameters are removed in a pre-training manner.

In order to remain aligned with the principle suggested by Le Cun et al. (1990a) prior knowledge about the data is required. In the case of image data, it is well accepted that images have inherent local structure, whereby neighbouring pixels are highly correlated (Le Cun et al., 1998). CNNs take advantage of this information to encode sparse networks with local dependencies as described in the next section.

3.4 Convolutional Neural Network Theory

As mentioned above, the principle behind CNNs is to reduce the number of parameters in a deliberate manner prior to training by taking advantage of knowledge about image data. In this regard CNNs take advantage of the local nature of features in images and pattern recognition is facilitated by identifying them and combining them (Le Cun et al., 1990a). Furthermore CNNs are encoded to accommodate for shift, scale and distortions of images. This is achieved through successive combinations of convolutional layers

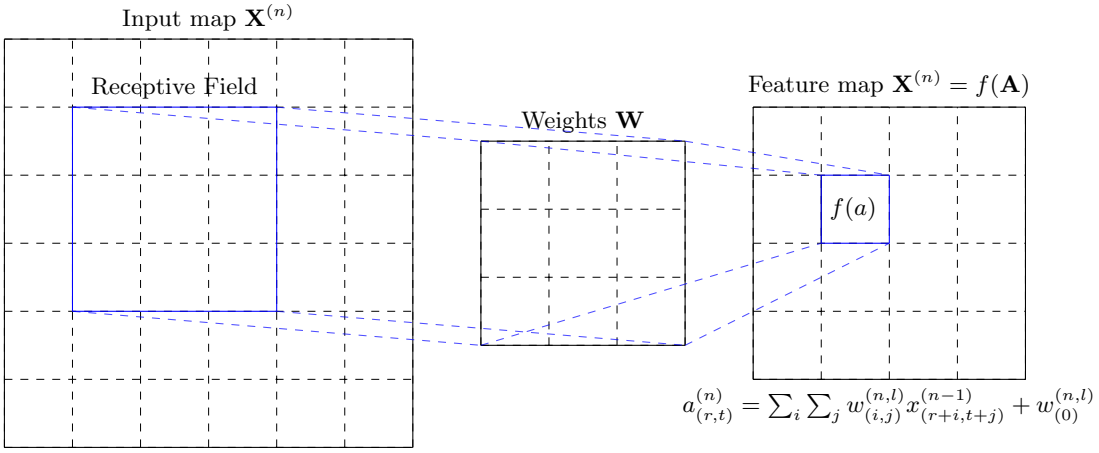


Figure 3.2: Representation of the convolution of weights $\mathbf{W}$ upon input map $\mathbf{X}^{(n)}$ to produce activities matrix $\mathbf{A}$. The output of the convolutional layer corresponding to the particular set of weights is feature map $\mathbf{X}^{(n)}$, which applies a non-linear activation function to each element of the activities matrix.

and downsampling layers. In this section the parameterization of convolutional and downsampling layers is introduced as well as the intuition behind them. The discussion of the inspiration and the development of CNNs is left to Chapter 4.

3.4.1 Feedforward Propagation in Convolutional Neural Networks

The convolutional layer exploits local dependencies in image data by constraining neuron inputs to a local area of the input. This area is referred to as the receptive field. This constraint forces the neuron to develop a local feature representation within the receptive field. For example, a neuron constrained to a small local area can only detect an edge or bar, rather than trying to identify the much larger object such as the person in the image. The receptive field is scanned through the input image in a contiguous and usually overlapping manner, a common set of parameters is applied to the inputs of each location. Neurons are arranged in the following layer such that contiguous neurons have contiguous receptive fields allowing the global structure of the image to be preserved. The map of neurons that uses the same set of parameters is referred to as a feature map. By scanning through an image with the same parameters, shift invariance is encoded into the model. Upon the left most grid in Figure 3.2 a receptive field is depicted as a blue square, to produce the following feature map, the inputs in contiguous receptive fields are combined with the weights and acted upon by the activation function. The method of scanning the receptive field across an image using the same set of parameters is similar to the mathematical convolution, hence the layer is referred to as a convolutional layer (Le Cun et al., 1998).

In each convolutional layer several feature maps are usually trained, each with their own set of parameters, allowing multiple features to be extracted from the same locations across the image. To encourage the receptive fields to learn different features, the weights are randomly initialized.

Parameterization of a convolutional layer in a NN is then as follows:

$$x_{(r,t)}^{(n,l)} = f(a_{(r,t)}^{(n,l)}) = f\left(\sum_{i,j,m} w_{(i,j)}^{(n,m,l)} x_{(r+i,t+j)}^{(n-1,m)} + w_{(0)}^{(n,l)}\right)$$

where

- $x_{(r,t)}^{(n,l)}$ denotes the output from the neuron at (r,t) in the l^{th} feature map in the n^{th} layer.
- Similarly, $a_{(r,t)}^{(n,l)}$ denotes the activity of the neuron at (r,t) in the l^{th} feature map of the n^{th} layer.
- $w_{(i,j)}^{(n,m,l)}$ is the weight acting upon the connection at (i,j) between the m^{th} feature map of the $(n-1)^{\text{th}}$ layer and the l^{th} feature map of the n^{th} layer.
- $w_{(0)}^{(n,l)}$ is the bias for the l^{th} feature map in the n^{th} layer.

The downsampling layer in a CNN introduces scale and shift invariance into a network and helps reduce the size of the dimensionality of the input. Downsampling reduces a local set of inputs, usually to a single output. For example, typically in a downsampling layer, non-overlapping $n \times n$ consecutive regions of the input are reduced to a single value using either an unweighted average or the maximum value of the region. See Figure 3.3 for an illustration of a downsampling layer. There are many different approaches to downsampling, including using over-lapping regions (Krizhevsky et al., 2012) or stochastic pooling (Zeiler and Fergus, 2014).

Downsampling reduces the dimensionality of the input while ensuring important local features persist through the layer. In the case of max-pooling this is achieved by passing the largest activity of a local region. As a result downsampling layers allow the network to place more emphasis on the relative position of features, rather than their absolute position in the scene. In a convolutional layer, if a feature is translated in the input image, the neuron that activates upon it will be similarly translated in the feature map. In this regard, the convolutional layer is susceptible to translation variance. The impact of translation, if it occurs within a local downsampling receptive field, it will be removed, improving the network's robustness against small translation variances. The network's robustness to scale invariance will similarly be improved by downsampling.

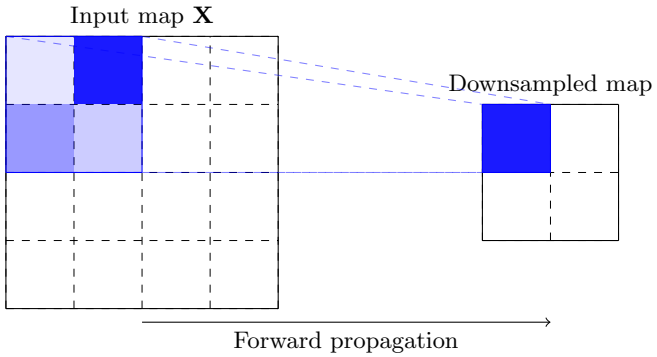


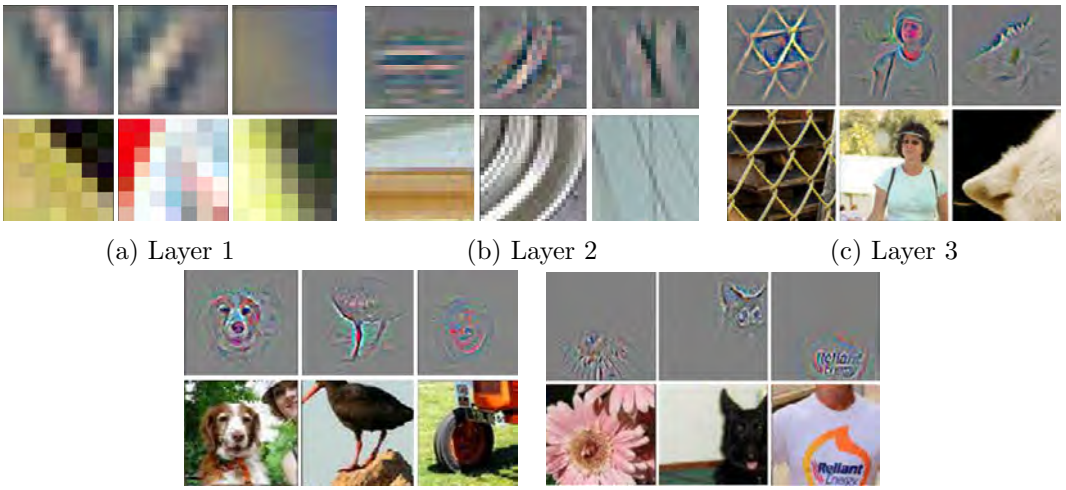
Figure 3.3: The darker blue indicates a larger activity. In max-pooling the largest activation value in the local region of the input map is maintained, and the remaining values are discarded.

A typical CNN consists of several alternating convolutional and sub-sampling layers with final layer aligning to the specific task of classification or regression. This hierarchical structure affords CNNs with ability to learn successively more complex features about the object and improves the networks' robustness against feature variance (Zeiler and Fergus, 2014). As an example of CNNs learning complex feature detectors, consider Figure 3.4 which depicts pixel reconstructions for different feature maps at different layers. In the early layers of a CNN, low level feature detectors, such as edge detectors, are typically learnt. In the higher layers, these low level features are successively combined to build more complex feature detectors. Feature detectors in hidden layers after the first layer, are difficult to interpret as they represent combinations of previous feature detectors. The additional complexity learnt in the higher layers is demonstrated when inspecting pixel reconstruction of activations, e.g in Layer 1 in Figure 3.4 edges are found, whereas in Layer 5, areas containing text in the image are identified. In addition, the hierarchy results in a gradual reduction of dimensionality of the input to the typically much lower dimensionality output.

3.4.2 Backpropagation in Convolutional Neural Networks

As with fully-connected NNs, CNNs commonly use gradient descent and backpropagation to update the free parameters. Backpropagation needs to be adapted to accommodate the weight sharing scheme, but is otherwise similar to backpropagation in fully-connected NNs (See Equations 2.5 and 2.6). This is achieved as follows:

$$\frac{\partial E}{\partial w_{(i,j)}^{(n,m,l)}} = \sum_{s,t} \left(\frac{\partial E}{\partial a_{(s,t)}^{(n,l)}} \frac{\partial a_{(s,t)}^{(n,l)}}{\partial w_{(i,j)}^{(n,m,l)}} \right) = \sum_{s,t} \left(\frac{\partial E}{\partial a_{(s,t)}^{(n,l)}} x_{(s+i,t+j)}^{(n-1,m)} \right)$$



(a) Layer 1 (b) Layer 2 (c) Layer 3

(d) Layer 4 (e) Layer 5

Figure 3.4: Examples of pixel reconstructions of the largest activations of feature maps at different layers in a CNN for given input images along with the corresponding image patch. This serves to illustrate which feature in the image results in the largest activation value. Features maps from 5 different layers are depicted, the grey squares illustrate the pixel reconstructions achieved through reversing a CNN and setting all other activation values to zero in the layer (Zeiler and Fergus, 2014). The corresponding image below the grey square illustrate the image patch that the reconstruction was inferred from. Notice that in the early layers, low level features are found such as edge detectors, where as in the high layers, more abstract features are found, such as faces. Furthermore the detector is robust against the translation of the object in the higher layers. See Zeiler and Fergus (2014) for further details.

such that weight sharing introduced by convolutions is accounted for when calculating the error derivative with respect to a weight.

$$\frac{\partial E}{\partial a_{(s,t)}^{(n,m)}} = f'(a_{(s,t)}^{(n,m)}) \sum_{l,r,t} \frac{\partial E}{\partial a_{(r,t)}^{(n+1,l)}} w_{(r,t)}^{(n+1,m,l)}$$

whereby the error derivative with respect to the activation value is backpropagated, accounting for different feature maps as well as the structure of the weights.

Similarly, backpropagation through downsampling layers needs to be accounted for. In the case where a max-pooling function is used, backpropagation uses the location of the maximum activation from the forward propagation step to inform which error derivative is backpropagated, as illustrated in Figure 3.5.

3.5 Conclusion

In this chapter, an adaptation to the NN algorithm, CNNs, was presented as an image classification algorithm. NNs' hierarchical structure allow successively more complex representations of the data, however in their default, fully-connected form, are

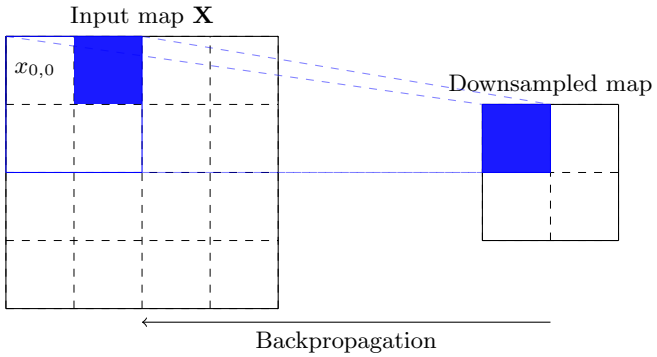


Figure 3.5: In the case of max pooling, the error is backpropagated to the unit in the receptive field with the largest activity upon the forward pass. I.e. in Figure 3.3, $x_{1,0}$ has the largest activity for a particular training case. Then upon backpropagation, the error derivative with respect to the activity of the previous layer is passed to the $(1, 0)$ location as illustrated above

intractable upon current hardware when applied to the raw input. While, historically, feature extraction techniques have been applied to reduce the dimensionality of the input while maintaining the discriminative information in the image, these approaches are not readily transferable between image sets as they encode expert knowledge. Rather architectural constraints are applied that allow the network to receive high dimensional input, learn features from raw inputs, improve robustness to variances that occur naturally in images and improve the tractability of the network.

As an example of the reduction in number of free parameters, Simard et al. (2003) achieved an error rate of 0.4% upon the MNIST dataset with approximately 128,000 parameters. Cireşan et al. (2010) achieve a comparatively similar result of 0.41% on the MNIST dataset using a deep fully-connected NN that had approximately 6,690,000 parameters. This serves to illustrate that for a much smaller number of parameters, a CNN can achieve a similar performance upon the MNIST dataset.

The intuition behind CNNs was also presented, whereby local features are successively combined to gradually develop from low level feature detectors to high feature level detectors. The succeeding chapter discusses the inspiration and history of the development of NNs as well as recent superior results that CNNs have achieved upon large image datasets.

In this section a number of subjective choices regarding the architecture of the network were introduced, however little insight was given as to the best choices. For example, how many convolutional and downsampling layers should be included in a network to achieve the best results. This is done deliberately since these discussions are more appropriately included in Chapter 9.

Chapter 4

History of Convolutional Neural Networks

4.1 Introduction

Many of the characteristic features of CNNs were inspired by observations of the visual cortex of a cat and first featured in the Neocognitron, a predecessor to the CNN (Fukushima, 1988). Using these features in the context of a well defined training regime provided by NNs, Le Cun et al. (1998) developed a system to accurately classify handwritten digits from raw input.

Subsequent research has focused upon improving the accuracy of CNNs (Cireşan et al., 2012b; Wan et al., 2013) and developing the capability of NNs to handle an input with higher dimensionality (Krizhevsky et al., 2012; Zeiler and Fergus, 2014). In order to achieve this, CNNs have increased in depth and consist of a larger number of trainable parameters and are trained for many more epochs, which have necessitated the development of improved regularization techniques (Hinton et al., 2012; Wan et al., 2013). These developments have been supported by the improvement in hardware and the use of graphical processing units (GPUs) to increase the speed of computation, allowing for previously intractable dimensionalities to be trained upon (Cireşan et al., 2012b; Krizhevsky et al., 2012; Zeiler and Fergus, 2014).

As a result, in recent years CNNs have demonstrated superior performance across a wide array of image classification datasets (Wan et al., 2013; Hinton et al., 2012; Zeiler and Fergus, 2014). In the case of the ImageNet Large-Scale Visual Recognition Challenge 2012 task of classifying images of a resolution in excess of $3 \times 256 \times 256$ into one of 1000

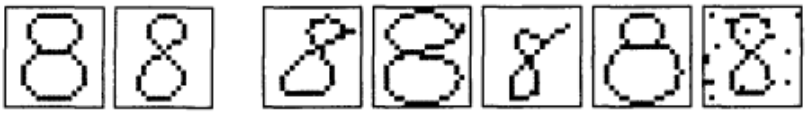
classes, a CNN provided superior performance, improving upon the next best entry by 37% (Krizhevsky et al., 2012).

In the next section important developments in CNN's history are introduced. The Neocognitron (Fukushima, 1988), and the architectural features it introduced are discussed as well as the seminal work performed by Le Cun et al. (1998). In Section 4.3 recent, successful results along with features which contributed to their improvement are discussed.

4.2 Early Convolutional Neural Networks

The Neocognitron's architecture and learning regime both draw inspiration from the biological structure of the brain (Fukushima, 1988). The former borrows the idea that through a hierarchy of feature detectors, simple, local features can be successively used to build complex feature detectors that are robust to distortions in images such as translation, deformations and scale. The second, uses the idea that biological NNs learn, starting from a very densely connected state initially and then removing the redundant connections.

The Neocognitron is presented as a universal pattern recogniser due to its trainable parameters and ability to handle the afore mentioned distortions. Once trained upon a small idealized training set of handwritten digits, the Neocognitron was demonstrated to recognise unseen cases even when they were distorted. See Figure 4.1 for examples of the training and test cases.



(a) (b)

Figure 4.1: The training pattern for the digit eight in layer S4 is depicted in (a). Weights were trained using hand built training patterns for each layer in an unsupervised manner. Figure (b) illustrates examples of test patterns that the Neocognitron correctly identifies (Fukushima, 1988).

The architecture of the Neocognitron is composed of alternating layers of S-cells and C-cells as displayed in Figure 4.2. S-cells aggregate a set of local inputs as well as an inhibitory input using a set of trainable parameters, the output of which is bounded below by 0 in a similar manner to the rectified linear unit. Through the Neocognitron's learning regime, sets of S-cells are trained to identify the same feature at different locations, which are then arranged together in 'cell-planes'. C-cells downsample S-cell layers by activating if at least one of a set of local S-cells to which a C-cell is connected is active.

The C-cell layer does not contain any trainable parameters. For complete details of the architecture and training algorithm of the Neocognitron, refer to Fukushima (1988).

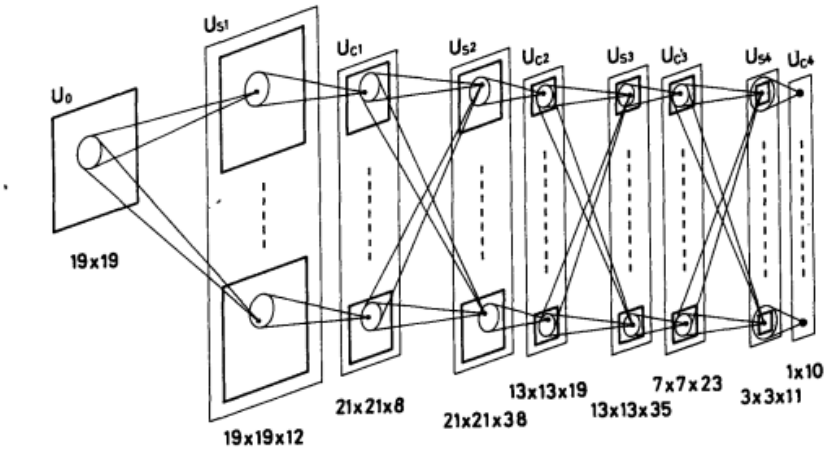


Figure 4.2: The hierarchical structure of the Neocognitron, an image is inputted on the left in U_0 and through alternating layers of S-cells and C-cells successively more complicated and distortion invariant features are recognized until the final classification in layer U_{C4} (Fukushima, 1988). Each rectangle depicts a layer, the squares within the rectangles represent cell-planes, the circles represents local connections for individual cells. Layer labels U_C or U_S detail whether the layer is composed of S-cells or C-cells. The output dimensions of each layers are included below the image.

Many of features of the Neocognitron are included in CNNs. Convolutional layers echo S-cell layers of the Neocognitron, although neurons in CNNs typically do not have an inhibitory component. The downsampling layers in CNNs are reminiscent of the C-cell layers.

The most substantial difference between CNNs and the Neocognitron is the training regime. The Neocognitron made use of an unsupervised method that required the construction of training examples on a layer by layer basis, making it difficult to generalize.

Le Cun et al. (1998) developed a system to identify and then classify hand written digits upon bank cheques, using a CNN to perform the classification. The system was implemented and used to read several million cheques a day. The authors demonstrated that automated feature learning upon raw pixels using a CNN could perform accurately.

To demonstrate the network's capability, Le Cun et al. (1998), modified the National Institute of Standards and Technology (NIST) dataset of handwritten digits, to produce the popular Mixed NIST (MNIST) dataset. The MNIST dataset consists of a training set of 60,000 28×28 grayscale images of hand written digits ranging from 0 to 9 and a test set of 10,000 similar cases. None of the writers in the training set feature in the test set and each image consists of only one class. See Figure 4.3 for a sample of the hand written digits.

Figure 4.3: Examples of MNIST digits. Source: Le Cun et al. (1998).

The CNN, named LeNet-5, had seven layers. The first four layers consisted of alternating convolutional and sub-sampling layers. The fifth layer, is described as convolutional although functions as a fully-connected layer, as the dimensions of the layer's input and the receptive field have the same size. The sixth layer is fully-connected and the output layer consists of radial basis functions that find the Euclidean distance of the output from the previous layer to a predefined pattern. The class with the smallest distance from it's associated vector is the predicted class.

To describe the network's architecture concisely the notation introduced by Cireşan et al. (2012b) is used. The parameterization of LeNet-5 is described by $32 \times 32 \rightarrow$ 6C5 $\rightarrow$ MP2 $\rightarrow$ 16C5 $\rightarrow$ MP2 $\rightarrow$ 120N $\rightarrow$ 84N $\rightarrow$ 10N. The input's dimensions are described by the first term. xCy indicate convolutional layers, where x is the number of feature maps trained each with a receptive field of $y \times y$. Non-overlapping max-pooling layers are described by MPz, where a $z \times z$ area is downsampled. Fully-connected layers with w neurons are described by wN. The final term is the output layer, which is described as a fully-connected layer. The resulting output dimensionality of each layer is portrayed in Figure 4.4.

The 28×28 pixel image was centred in the 32×32 input such that the strokes of the digits on the border of the image could appear in the centre of the receptive fields when they were convolved across the edges of the image.

The decrease in the size of the spatial dimensions of the feature maps is as a result of the convolutions and subsampling and improves the models invariance towards distortions in images as detailed in Chapter 3. Increasing the number of feature maps allows the

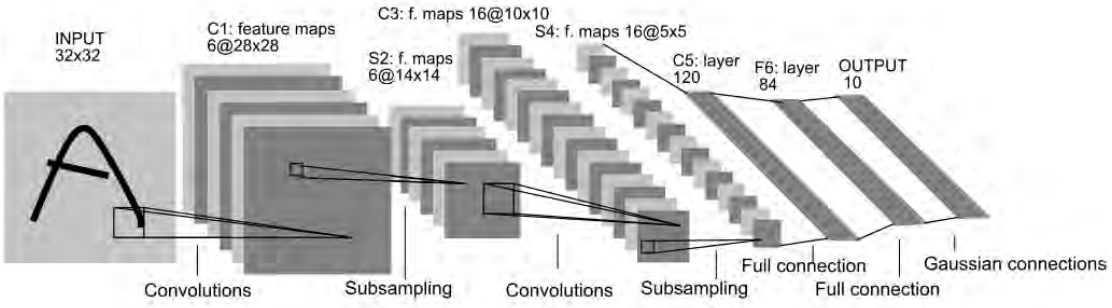


Figure 4.4: Depiction of the dimensionality of the output of LeNet-5, Cx, Sy and Fz describe convolutional, subsampling and fully-connected layers, respectively. Each square in the layer represents a separated feature map. Note that the fifth layer is described as convolutional, however functions as a fully-connected layer as the receptive field is the same size as the input (Le Cun et al., 1998).

model to improve its representation of the classes by increasing the models capacity with additional free parameters.

Le Cun et al. (1998) also introduced the Stochastic Diagonal Levenberg Marquardt (SDLM) learning rate adaptation algorithm, which is detailed in Section 5.5.1.

Data augmentation is commonly used in the context of image classification. Additional training data can be generated by adding noise to existing data in a manner that is consistent with natural variances occurring within the data. For example, images of natural scenes are orientated with the ground located at the bottom of the image, and can be flipped across their vertical axis without changing to nature of the object in the image as in Krizhevsky et al. (2012). Care needs to be taken when augmenting data, for example, the afore mentioned augmentation could not be applied upon digit recognition tasks as it would alter the discriminative features of the digit. Le Cun et al. (1998), however, used a combination small affine transformations such as horizontal or vertical translations. For further examples of augmentations see Sections 7.3.1 and 9.2.3.

Using augmentations Le Cun et al. (1998)'s network achieved an error rate of 0.8% with augmented data, a similar network, LeNet-4 used boosting and augmented data to achieve an error rate of 0.7%. The networks were demonstrated to surpass the performance of regular classification methods with the exception of SVMs, which performed comparably, however SVMs were relatively more computationally expensive when applied to the pixel data (Le Cun et al., 1998).

4.3 Recent Developments in Convolutional Neural Networks

In-line with improving hardware, CNNs have increased in scale and have demonstrated superior performance upon large image datasets compared to other algorithms as well as near or better than human performance upon smaller image datasets. These achievements have been in part attributed to improvements in hardware and the use of GPUs to increase the speed of computation (Cireşan et al., 2012b; Krizhevsky et al., 2012). This has facilitated CNNs with a larger number of trainable parameters trained upon larger datasets for more epochs, to improve accuracy. As a result of a larger number of trainable parameters, overfitting has become more prevalent and subsequently new regularization techniques have been developed.

Here, improvement of accuracy upon the MNIST and other small image datasets by CNNs is discussed, including factors contributing to their success. Thereafter CNNs that have been built for a larger dimensionality are presented, with focus on the work done by Krizhevsky et al. (2012) upon the ILSVRC-2012 dataset.

4.3.1 Small Image Datasets

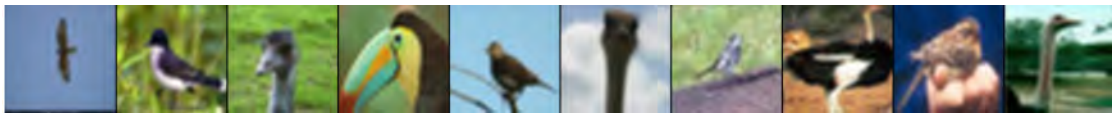
To assess classification algorithms numerous image datasets have been built. To provide context to the results, a couple of the popular small image datasets are introduced here.

The CIFAR-10 and CIFAR-100 datasets consist of 32×32 RGB pixel images, where each image belongs to one of 10 or 100 different classes, respectively (Krizhevsky, 2009). Images are constrained to only having one dominant, easily identifiable class present, but otherwise no attempt to constrain scale or viewpoint is made. The CIFAR-10 datasets consists of 50,000 32×32 RGB training images and 10,000 similar test images. See Figure 4.5 (a) for examples of the CIFAR-10 dataset.

The NORB dataset consists of 108×108 grayscale stereo images of posed toys upon cluttered backgrounds (Le Cun et al., 2004). The dataset was designed for 3D object recognition, and each pair of stereo images belongs to one of 5 classes of toys. Each toy is contained in the image and occasionally toys from other classes are added along the boundaries. See Figure 4.5 (b) for examples of the NORB dataset.

The Street View House Numbers (SVHN) dataset consists of 32×32 RGB images of digits from real world house numbers. Each image consists of one of ten different classes, one per digit (Netzer et al., 2011). Images are centred upon the digit and sometimes can include partially cropped out digits from neighbouring digits. The training dataset

consists of 73,257 images and the test dataset consists of 26,032 images. See Figure 4.5 (c) for examples of the SVHN dataset.



(a) CIFAR-10

(b) NORB

(c) SVHN

Figure 4.5: Examples of the (a) CIFAR-10 (Krizhevsky, 2009), (b) NORB (Le Cun et al., 2004) and (c) SVHN (Netzer et al., 2011)

4.3.2 Small Image Classification Results

Cireşan et al. (2012b) built an ensemble[1] of simple but deep CNNs that provided human comparable performance upon the MNIST dataset using networks with the following architecture: $29 \times 29 \rightarrow$ 20C4 $\rightarrow$ MP2 $\rightarrow$ 40C5 $\rightarrow$ MP3 $\rightarrow$ 150N $\rightarrow$ 10N. Compared to the CNN employed by Le Cun et al. (1998), this architecture employed more feature maps, but one fewer fully-connected layer, resulting in a comparable amount of free parameters.

Furthermore, each network used a simple exponential decay rate upon the learning rate rather than SDLM method of adaptation, and was trained for a much longer period upon augmented data. Using an efficient implementation upon a GPU, each network was trained for 800 epochs upon the dataset in 14 hours, where as in the work done by Le Cun et al. (1998), training took two to three days for 10 - 20 epochs.

The average of the errors by each of the 35 networks was $0.44 \pm 0.06\%$, and the error of the averaged response of all the networks was 0.23%, which is comparable to human performance of 0.2% (LeCun et al., 1995) (See Chapter 6 for further details on averaging the response of multiple networks).

Cireşan et al. (2012b) applied a similar methodology to other small image datasets, including the CIFAR-10 and NORB datasets, but with a larger CNN. Upon the CIFAR-10, eight networks with the following architecture were trained: $3 \times 32 \times 32 \rightarrow$ 300C3 $\rightarrow$

[1] An ensemble model combines the output of multiple NNs, usually by a simple average, to provide a single output. For further detail see Chapter 6.

MP2 $\rightarrow$ 300C2 $\rightarrow$ MP2 $\rightarrow$ 300C3 $\rightarrow$ MP2 $\rightarrow$ 300N $\rightarrow$ 100N $\rightarrow$ 10N. Upon the NORB dataset, downsampled to $3 \times 48 \times 48$, an ensemble of four networks, each using the afore mentioned architecture but with 50 feature maps per convolutional layer, rather than the 300. These ensemble networks improved the previous best results by 39% and 46% to arrive at an error rate of 11.21% and 2.7% upon CIFAR-10 and NORB, respectively.

More recently Wan et al. (2013) built an ensemble of 12 CNNs upon the CIFAR-10 dataset and achieving the current state-of-art of 9.32%. Each CNN used an architecture of $3\times24\times24 \rightarrow$ 64C5 $\rightarrow$ MP$3, 2 \rightarrow$ 64C5 $\rightarrow$ MP$3, 2 \rightarrow$ 64LC3 $\rightarrow$ 32LC3 $\rightarrow$ 128N $\rightarrow$ 10N. xLCy indicates a locally connected layer which are similar to convolutional layers, however weights common to a feature map are not shared. MP$z, y$ indicate over-lapping max-pooling layers, where $z \times z$ regions are downsampled and the stride between successive downsampled regions is y. Wan et al. (2013) implemented a new regularization technique, DropConnect, to improve the networks, see Chapter 5 for further details.

Using a similar architecture as described above in a five network ensemble, Wan et al. (2013), also improved upon the SVHN and downsampled NORB dataset, achieving an error rate of 1.94% and 3.03%, respectively.

4.3.3 ImageNet Large-Scale Visual Recognition Challenge

The ImageNet dataset consists of approximately 14 million high-resolution images arranged into approximately 22,000 categories (Deng et al., 2009). The ImageNet Large Scale Visual Recognition Challenge (ILSVRC), published on an annual basis, presents an image classification problem using a subset of the ImageNet data. In 2012, the ILSVRC the training set consisted of 1.2 million images arranged into 1000 distinct categories and the test data consisted of 150,000 images drawn from the same categories. The images in the dataset are not controlled for background clutter, angle or scale. See Figure 4.6 for examples of images.

The size of the dimensions between different images in the dataset was not controlled, in order to input the raw pixel values into CNN built by Krizhevsky et al. (2012) each image was downsampled to 256×256 RGB pixels. Each network in the ensemble of five had a total of 11 layers structured in an architecture as follows: $3 \times 224 \times 224 \rightarrow$ 96C11 $\rightarrow$ MP$3, 2 \rightarrow$ 256C5 $\rightarrow$ MP$3, 2 \rightarrow$ 384C3 $\rightarrow$ 384C3 $\rightarrow$ 256C3 $\rightarrow$ MP$3, 2 \rightarrow$ 4096N $\rightarrow$ 4096N $\rightarrow$ 1000N. Overlapping max-pooling was found to improve the results of the network, as such within each pooling layer a receptive field of 3×3 was used with a stride of 2.

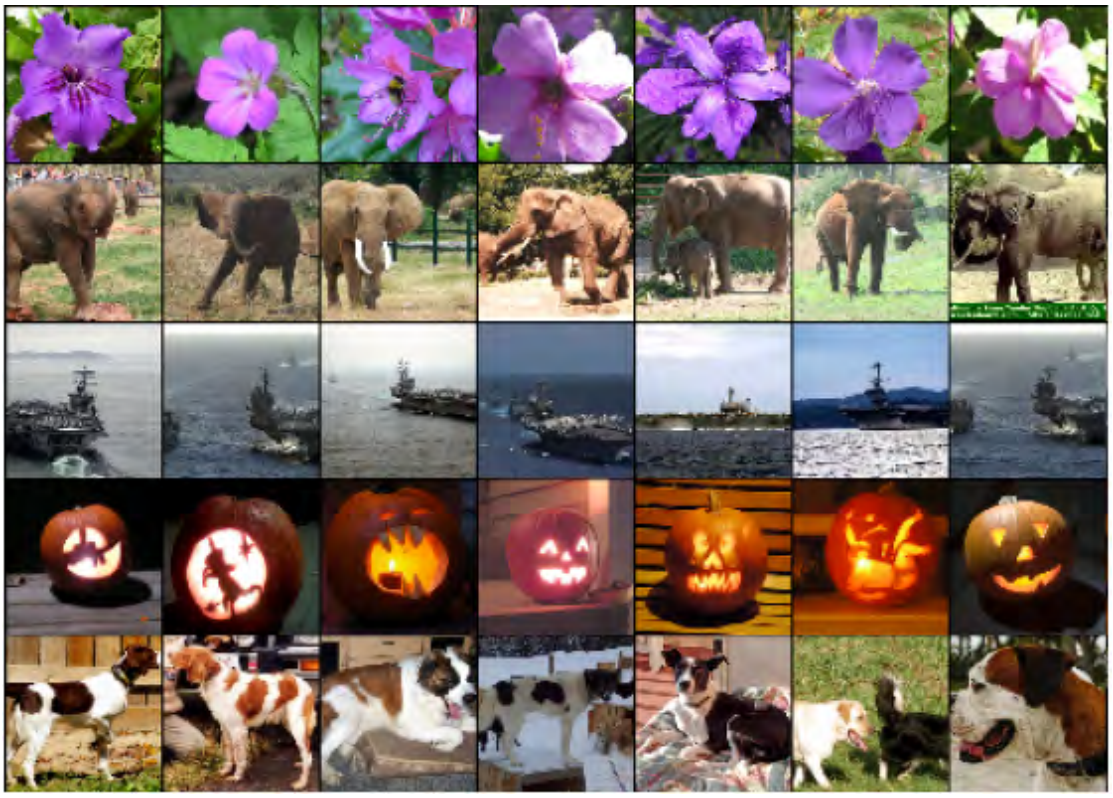


Figure 4.6: The first column shows five examples from the ImageNet which have been cropped and downsampled to 256×256. The following six columns are images with the closest Euclidean distance to the output vector of test image according to the network built by Krizhevsky et al. (2012)

The memory burden of the network required it to be implemented upon two graphics cards, which imposed constraints upon the convolutions in the NN. Specifically, feature maps were divided equally between the two GPUs and, without specific instruction, each GPU could not access the other's memory. For full details of the network's implementation see Krizhevsky et al. (2012).

Another significant challenge was avoiding overfitting. In this regard Krizhevsky et al. (2012) implemented several different methods to regularize the network. These included the use of Dropout (described in Section 6.3) and augmenting the data. The model also employed the rectified linear unit (See Section 2.2.2) which was demonstrated to speed learning.

Upon the ILSVRC-2012 dataset the network achieved an error rate of 16.4% upon the Top-5 task (correct label occurs in the top 5 predictions of system), improving on the next best submission by 37% which employed a mixture of hand built feature detectors. This result was improved by Zeiler and Fergus (2014) to 14.8%, through careful dissection of the layers to inform an improved architecture and use of an ensemble of networks. Similarly large CNNs have been employed to win the ILSVRC-2013 challenge.

4.4 Conclusion

The hierarchical combination of convolutional and pooling layers has repeatably been demonstrated to perform image classification well and the NN framework provides a reliable method of updating the parameters. While early work upon CNNs was limited, recent successes have drawn attention to them and CNNs have been demonstrated to provide superior performance upon several image classification datasets.

Where, previously, CNNs were difficult to implement upon existing hardware upon reasonably size images due to the large number of computations required to optimize the parameters in a CNN, recently, CNNs have been developed to classify larger image sizes due to improved hardware and the use of the GPU. The large CNN developed by Krizhevsky et al. (2012) had approximately 147× the number of inputs when compared to the early CNN developed by Le Cun et al. (1998). Whereas the number of layers has not substantially increased from the initial CNNs, recent CNN's architectures have placed emphasis on learning more feature maps, increasing the number of trainable parameters in a network.

The recent, larger CNNs, supported by improved hardware, are more prone to overfitting due to the increased number of free parameters. To mitigate this effect, larger datasets have been employed and new methods of regularization have been developed such as Dropout or DropConnect. Furthermore, more computing devices available, ensembles of CNNs have been developed and have been demonstrated to improve accuracy of CNNs Cireşan et al. (2012b) (See Chapter 6 for further detail).

Chapter 5

Learning Rate Adaptation Techniques

5.1 Introduction

An important component of the backpropagation algorithm is the learning rate η. It can influence the speed at which the network converges, the final classification accuracy of the network, and cause divergence if improperly defined.

If η is set too large then gradient descent could diverge from the minimum or if η is too small then gradient descent could take an impractical amount of epochs to converge upon a minimum (Le Cun et al., 2012). Consider Figure 5.1 where the effects of different values of η upon a simple example consisting of a linear single weight network with a quadratic error function are illustrated. Let η_{opt} be the optimal learning rate such that the illustrated network reaches a minimum in a single step as in 5.1(b). In Figure 5.1(a) multiple steps are required to reach the minimum when $\eta < \eta_{opt}$, increasing the time taken to train the network. Likewise in 5.1(c) where $\eta_{opt} < \eta < 2\eta_{opt}$, however in this case the weight oscillates around the minimum value. Finally, if $\eta > 2\eta_{opt}$ then the network will diverge from the minimum as illustrated in 5.1(c).

Previously when discussing the backpropagation algorithm, the learning rate had been assumed to be constant and identical for all weights. However, in an artificial NN with more than one weight using an individual or local learning rate for each weight rather than a global learning rate can be beneficial in reducing the learning time. Consider Figure 5.2 which illustrates a linear NN comprising of two weights and a quadratic error function where $|\frac{\partial E}{\partial w_1}| > |\frac{\partial E}{\partial w_2}|$ at point a. Let η_1^{opt} and η_2^{opt} be the optimal learning rates for the weights w_1 and w_2 respectively, where $\eta_1^{opt} < \eta_2^{opt}$. If a global learning rate η is

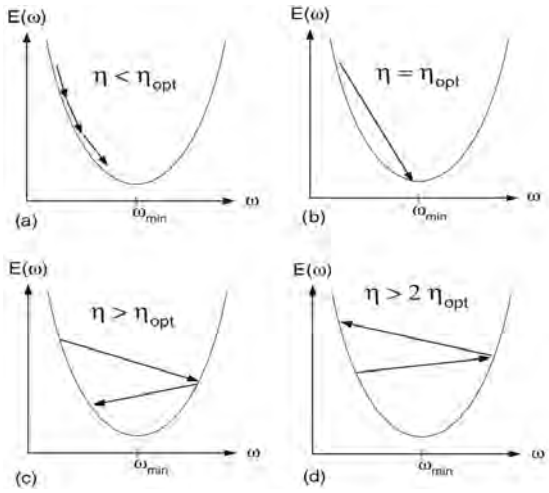


Figure 5.1: Quadratic error function for a linear, single weight network. Source: Le Cun et al. (2012)

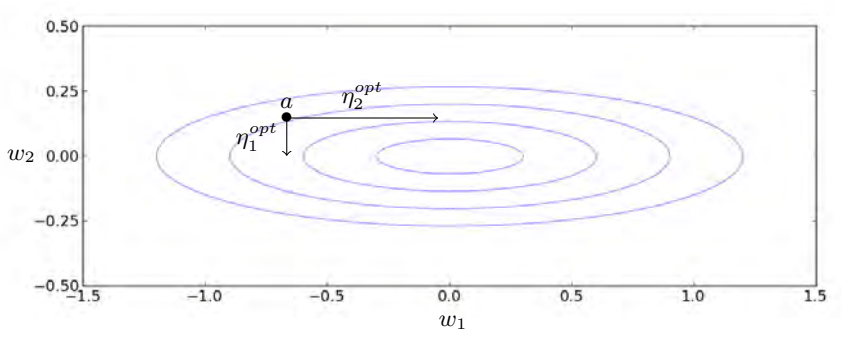


Figure 5.2: An elliptical error bowl in the weight space of w_1 and w_2. The contours map the error value associated with the weights with the minimum error located at the centre of the bowl. Due to the shape of the error surface the optimal step size for each parameter is different.

used where $2\eta_1^{opt} < \eta < 2\eta_2^{opt}$, the network will diverge upon the w_1 axis. As such, if a global learning rate is used, η must be less than twice the optimal learning rate of the weight with the steepest gradient η_1^{opt}. However, learning upon the w_2 axis will then be slowed. By using local learning rates this problem can be circumvented (Le Cun et al., 2012).

A large number of local and global learning rate adaptation methods have been suggested in the literature. These can be categorised loosely according to the information they employ. Simple schedules generally only employ epoch index information and use a global learning rate. While not theoretically well justified, they are still widely and successfully employed due to their simplicity and ease of implementation (Krizhevsky et al., 2012; Cireşan et al., 2011, 2012b). The next category of learning rate adaptation algorithms employs the readily available first order error derivative, which would have been calculated prior to adaptation for backpropagation. Since the Delta-Bar-Delta method

(Jacobs, 1988), numerous adaptations and similar algorithms have been proposed, Section 5.4 reviews a method proposed by Almeida et al. (1998) alongside the Delta-Bar-Delta method. Finally, the third category uses Newtonian and Quasi-Newtonian type methods to better approximate the optimal learning rate for each weight and hence accelerate convergence. Due to the complexity of calculating the full Hessian matrix of the error function with respect to the weights, it is usually approximated (Le Cun et al., 1998; Duchi et al., 2011). The SDLM method and an extension of it are discussed in the Section 5.5. In addition to adapting the learning rate, momentum has very successfully been used with gradient descent to improve learning. Section 5.3 discusses momentum in full.

5.2 Static and Exponential Learning Schedules

Static and exponential schemes take advantage of starting with a relatively large learning rate and then reducing it to allow fast initial learning followed by a refinement of the weight values. Simple to implement, static and exponential learning rates have featured in a couple of recent successes (Krizhevsky et al., 2012; Cireşan et al., 2011, 2012b) demonstrating that they are still relevant.

Static scheduling usually involves an initial global learning rate that is adjusted by a predefined amount either according to a predefined schedule or depending upon the current performance of the NN. The latter case is more commonly referred to as performance scheduling where performance of the network is assessed upon a validation dataset to avoid overfitting upon the training dataset. Static scheduling requires an initial learning rate and an adjustment parameter to be defined as well as either a schedule defining when to reduce the learning rate or a set of conditions that define when to reduce the learning rate. Commonly, the initial learning rate is defined to be 0.01 and a multiplicative adjustment parameter is defined to be either 0.1 or 0.3 (Krizhevsky et al., 2012).

Despite it's simplicity and heuristic nature, a scheduled global static learning rate is still widely used. The recent CNN built by Krizhevsky et al. (2012), which provided superior performance upon the ImageNet dataset, employed performance scheduling along with momentum. In this case, the learning rate was initialized at 0.01 and reduced by a factor of ten if the validation error rate did not improve, this resulted in the learning rate being adjusted three times during the course of the network's training.

An exponential learning rate gradually decays the learning rate by a small multiplicative constant between each epoch:

$$\eta^{(t)} = \eta^{(0)} r^t$$

where r is the multiplicative constant, such that $0 < r < 1$ and t is the current epoch's index. Commonly r is defined after considering how many epochs the network is expected to run and what the desired final learning rate would be, given the initial learning rate. For example, Cireşan et al. (2011) defined $r = 0.993$ such that, with an initial learning rate of 10^{-3}, after approximately 1000 epochs the learning rate would be 10^{-6}. An exponential learning rate has the same advantage as static scheduling whereby it starts at a large initial learning rate that gradually reduces, facilitating rapid initial learning and then refined learning at latter epochs to allow parameters to converge upon locally optimal weights. Exponential scheduling requires two hyperparameters, r and $\eta^{(0)}$, to be defined, resulting in a simpler implementation when compared to the definition of the learning rate schedule required by a static learning rate schedule which updates the learning rate at specific epochs.

Recently, Cireşan et al. (2012b) used an exponential global learning rate schedule without momentum to achieve state-of-the-art performance upon the MNIST dataset using the same adjustment scheme as described above.

A similar method, Power scheduling, is used by Xu (2011); Bottou (2010) updates the learning rate according to:

$$\eta^{(t)} = \eta^{(0)}(1 + t/r)^{-1}$$

This rule results in an initial faster decay rate than an exponential learning rate allowing the network to spend more time refining the parameters with a smaller learning rate as t becomes large.

Senior et al. (2013) compared different learning rate schemes to identify speech using a fully-connected NN with four hidden layers and 1.6 million parameters. Their analysis included an adaptive global learning rate, AdaGrad (Duchi et al., 2011), which adapts the local learning rate according to the inverse of the root of the sum of squares of past error gradients such that the learning rate continuously decays, and decays faster during periods when the error gradient is large. In addition, Senior et al. (2013) introduced AdaDec, an extension of AdaGrad which had more memory, increasing the rate of decay's sensitivity to the recent error gradients. The authors found that despite additional complexity of AdaGrad and AdaDec, a global exponential learning rate decay provided the better or equivalent accuracy given sufficient training time although the adaptive static schedule and Power schedule achieved similar error rates to the exponential schedule with fewer training cases.

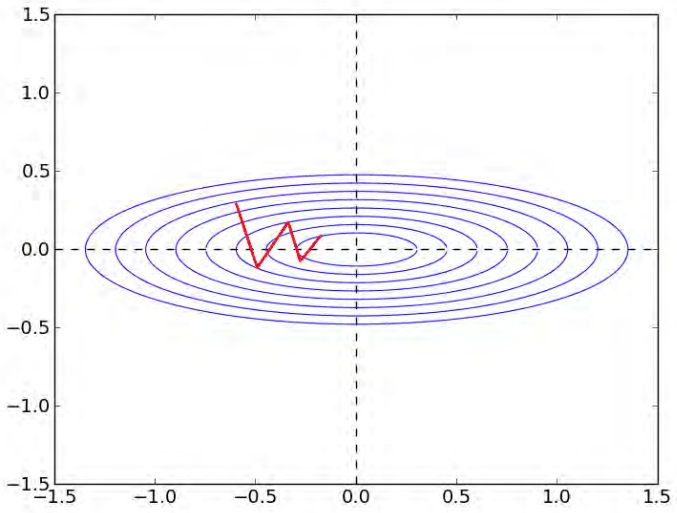


Figure 5.3: Gradient descent without momentum through an elliptical bowl. A pair of weights are portrayed by the axes, and the contour maps the error value associated with the weights. The minimum error is located at the centre of the bowl. The red line indicates a possible resulting series of weight updates without momentum which zigzags due to the local gradient at each update.

5.3 Momentum

The slope of the error surface can slow learning if the direction of the steepest gradient does not point towards the minimum. For example, in an elliptical bowl with regular gradient descent, the error value can oscillate, taking a long time to reach the minimum value as illustrated in Figure 5.3.

An effective method of dealing with this is to add a momentum term to the weight updates. i.e.:

$$\Delta \mathbf{w}^{(t)} = -\eta \nabla_{\mathbf{w}^{(t)}} + \mu \Delta \mathbf{w}^{(t-1)}$$

where $\nabla_{\mathbf{w}^{(t)}} = \frac{\partial E^{(t)}}{\partial \mathbf{w}^{(t)}}$. The momentum term has two beneficial effects, firstly, it dampens oscillations as opposing gradients eliminate each other. Secondly, it accelerates learning when the gradient is consistent. This can be demonstrated by considering an unchanging gradient and will result in the following (Hinton, 2012):

$$\Delta \mathbf{w} = -\eta \nabla_{\mathbf{w}} (1 + \mu + \mu^2 + \dots)$$

$$\Delta \mathbf{w} = -\frac{\eta}{1-\mu} \nabla_{\mathbf{w}} \quad \text{if } 0 < \mu < 1$$

Including the momentum term requires setting an additional parameter μ, where $0 \leq \mu < 1$.

5.4 First Order Learning Rate Adaptation Methods

To speed up learning, numerous local learning rates adaptation schemes have been proposed that make use of consecutive first order error derivatives' signs to determine whether to increase or decrease a weight's learning rate. They share the premise that if the sign of the first error derivative remains the same between the current and previous weight updates then learning should be accelerated. However if the sign changes, the learning rate should then decrease to prevent the weights from oscillating out of a minimum bowl and to allow the network to settle to a lower minimum.

This method was first popularized by the Delta-Bar-Delta (DBD) method presented by Jacobs (1988). DBD was a full batch update method that advocated a linear increase if the sign of the exponential average of previous error derivatives was the same as the current error derivative and an multiplicative decrease if they differed, i.e.:

$$\Delta\eta^{(t)} = \begin{cases} \kappa & \text{if } \bar{\nabla}_{w^{(t-1)}} \nabla_{w^{(t)}} > 0 \\ -\phi\eta^{(t)} & \text{if } \bar{\nabla}_{w^{(t-1)}} \nabla_{w^{(t)}} < 0 \\ 0 & \text{otherwise} \end{cases}$$

where $\bar{\nabla}_{w^{(t)}} = (1-\theta)\nabla_{w^{(t)}} + \theta\bar{\nabla}_{w^{(t-1)}}$ and $\eta^{(0)}$ is initialised as per a regular, uninformed learning rate.

The linear increase was implemented to prevent the learning rate becoming very large, whereas the exponential rather than linear decrease allows the learning rate to decrease quickly and prevents the learning rate from becoming negative which would result in gradient descent climbing uphill.

In a comparison between steepest descent and DBD across 4 low-dimensional classification tasks, Jacobs (1988) found that DBD converged substantially faster than regular steepest descent across all tasks, however results were not as well defined when momentum was included with steepest descent.

There are numerous variations of this method including adaptations to account for SGD (Sutton, 1992; Almeida et al., 1997, 1998). For example the Incremental Delta-Bar-Delta (IDBD) method (Sutton, 1992) extended the above algorithm to the online case and SGD case by decaying the exponential moving average according to the presence of the current input. The method maintains the previous learning rate when the input associated with the weight is not present, and decreases the previous learning rate when the input is present, improving upon DBD which is susceptible to large learning rate changes when the input is not relevant.

Almeida et al. (1998) derived a similar learning adaptation rule to the above using a different set of assumptions. Instead of adapting the exponential moving average according to the presence of the input, noise was accounted for explicitly in the derivation of the learning rule. Defining the noisy error derivative to be $d^{(t)} = \nabla_{w^{(t)}} + s^{(t)}$, where $s^{(t)}$ is a random value with a mean of zero to account for the stochasticity. The normalized learning rate adaptation rule is then defined as:

$$\eta_i^{(t)} = \eta_i^{(t-1)} \left(1 + k \frac{d_i^{(t-1)} d_i^{(t)}}{v_i^{(t)}} \right) \tag{5.1}$$

where $v_i^{(n)}$ is included to make the rule less sensitive to the particular function being minimized allowing the meta-learning rate, k, to be defined more generally. $v_i^{(n)}$ is defined as:

$$v_i^{(t)} = \gamma v_i^{(t-1)} + (1 - \gamma) \left(d_i^{(t)} \right)^2$$

In the above, k and the decay value for the exponential moving average, γ, need to be defined, Almeida et al. (1998) recommend a meta-learning rate of $k = 0.01$ and a rapid decay of the exponential moving average, where $\gamma = 0.9$. Furthermore it was recommended that a lower bound is defined for the step sizes such that they don't become zero, a lower bound of 10^{-10} was suggested. The above method was demonstrated to be robust against a wide array of different initial step-sizes, $\eta_i^{(0)}$, advantaging it over fixed step sizes, which were not found to be similarly robust (Almeida et al., 1998).

Where as not as simple as the exponential or static learning rate schedules, first order methods provide a well reasoned local learning rate adaptation method that only require additional memory to store the exponential average error gradient. However, in a comparison of learning rate adaptation techniques, Schaul et al. (2013) demonstrated that upon a NN applied to the MNIST dataset, AdaGrad (Duchi et al., 2011) provided a well tuned static learning rate provided superior accuracy to the method proposed by Almeida et al. (1998) upon the MNIST dataset despite Almeida et al. (1998)'s additional complexity.

5.5 Second Order Learning Rate Adaptation Methods

Second order learning rate adaptation methods exploit the local curvature to estimate the optimal learning rate. Consider the Taylor expansion of the error derivative of a network consisting of a single weight, $\frac{dE}{dw}$ around $w = w_c$ (Le Cun et al., 2012):

$$\frac{dE(w)}{dw} = \frac{dE(w_c)}{dw} + \frac{d^2E(w_c)}{dw^2}(w - w_c) + \dots$$

If the error function is quadratic around w_c as in Figure 5.1, then the higher order terms are zero. Let $w = w_{min}$, where w_{min} is the optimal value for w such that $\frac{dE(w_{min})}{dw} = 0$, results in the Newton-Raphson method, i.e.:

$$w_{min} = w_c - \left(\frac{d^2E(w_c)}{dw^2}\right)^{-1} \frac{dE(w_c)}{dw}$$

where $n_{opt} = (\frac{d^2E(w_c)}{dw^2})^{-1}$. In the case of multiple weights the inverse of the second order derivative is replaced with the inverted Hessian matrix, H^{-1}, which defines learning rates individually according to their local curvature. In practice, while the shape of the error surface is not necessarily quadratic, the use of H^{-1} can speed convergence (Le Cun et al., 2012). Subsequently, assuming the error surface to be locally approximately quadratic, multiple iterations H^{-1} are made with a learning rate $0 < \gamma < 1$ to account for non-zero higher order terms, such that weights are updated as follows:

$$\Delta w^{(t)} = -\gamma H(w^{(t-1)})^{-1} \nabla_{w^{(t-1)}}$$

Other than only approximating η_{opt}, the Newton-Raphson method has two drawbacks in the deep NN case. The inverse Hessian requires $O(N^3)$ calculations per iteration, hence H^{-1} is intractable for large networks. Secondly, the non-linear nature of the error surface can result in the Hessian not being positive definite everywhere, which can cause the error to diverge.

A number of methods have been developed to optimize SGD using second order approximations. For example quasi-Newton methods estimate the Hessian matrix through iterative updates employing only gradient information and usually perform in $O(N^2)$ time (Le Cun et al., 2012). Although initially limited to full-batch learning quasi-Newton methods have been adapted to the stochastic learning case (Schraudolph et al., 2007), and have demonstrated state-of-the-art performance, including winning one of the PASCAL Large Scale Learning Challenges (Bordes et al., 2009). Alternatively, the BHHH (Berndt, Hall, Hall and Hausman) method uses the Fishers Information matrix equality to demonstrate that the outer product of the error gradient can be used to estimate the Hessian matrix (Berndt et al., 1974). More recently, Roux and Fitzgibbon (2010) combined quasi-Newton methods with the Information matrix of the gradient to provide a learning rate adaptation algorithm that combined second order information with covariance information arguing that the covariance matrix provides additional information to the Hessian matrix. In a similar result, the Gauss-Newton and Levenberg Marquardt method estimate the Hessian to be the squared Jacobian of the error by making the assumption the network is linear with respect to w.

Using the same linearity assumption Le Cun et al. (1998) introduced the SDLM method which approximates the diagonal of H through a technique similar to backpropagation. Le Cun et al. (1998) employed this method in the well recognised CNN, LeNet, to classify handwritten digits of the MNIST dataset (See Chapter 3). SDLM also saw success more recently in Ji et al. (2010) where it was used in a CNN to recognise human actions in video. Schaul et al. (2013) improved upon SDLM in the variance-based stochastic gradient descent method which employs the backpropagated estimate of the diagonal H and uses automatic methods to define the hyperparameters, removing the need for a manual search for optimal hyperparameters.

In Section 5.5.1 the SDLM method is examined briefly. The Section 5.5.2 discusses the improvement suggested by Schaul et al. (2013) and it's extension to mini-batch gradient descent.

5.5.1 Stochastic Diagonal Levenberg Marquardt

The SDLM (Le Cun et al., 1998) defines the local learning rate for weight k as:

$$\eta_k = \frac{\mu}{\epsilon + h_{kk}}$$

where μ and ϵ are dampening variables used to prevent the learning rate from growing large when h_{kk} is small, similar to those used in the Levenberg Marquardt method. The remaining term, h_{kk}, is an estimate of the second order derivative of the error function with respect to weight k, such that for the n^{th} case:

$$h_{kk}^{(n)} = \frac{\partial^2 E^{(n)}}{\partial w_k^2}$$

Due to weight sharing in a CNN (See Section 3.4.2), the error derivative with respect to each weight is:

$$\frac{\partial E^{(n)}}{\partial w_k} = \sum_{(i,j)\in V_k} \frac{\partial E^{(n)}}{\partial w_{(i,j)}}$$

where $w_{(i,j)}$, an abbreviation of the notation described in Section 3.4.1 and describes the weight connecting neuron i to neuron j in a CNN, ignoring layer, feature map and location indices for ease of interpretation. The set V_k describes the set of connections that share weight k. As a result:

$$h_{kk}^{(n)} = \sum_{(i,j)\in V_k} \sum_{(k,l)\in V_k} \frac{\partial^2 E^{(n)}}{\partial w_{(i,j)} \partial w_{(k,l)}}$$

The first approximation made by Le Cun et al. (1998) was to drop the off diagonal terms of $h_{kk}^{(n)}$:

$$h_{kk}^{(n)} = \sum_{(i,j)\in V_k} \frac{\partial^2 E^{(n)}}{\partial w_{(i,j)}^2}$$

Then in a similar manner to backpropagation, the second order error derivative with respect to weight $w_{(i,j)}$ is decomposed using the second order chain rule according to the activity containing neuron j:

$$\frac{\partial^2 E^{(n)}}{\partial w_{(i,j)}^2} = \frac{\partial^2 E^{(n)}}{\partial a_j^2} x_i^2$$

Then the second order error derivative with respect to the input to activity containing neuron j, can be backpropagated through the network, where:

$$\frac{\partial^2 E^{(n)}}{\partial a_j^2} = f'(a_j)^2 \sum_k w_{(j,k)}^2 \frac{\partial^2 E^{(n)}}{\partial a_k^2} + f''(a_j) \frac{\partial E^{(n)}}{\partial x_j}$$

An approximation is made by dropping the second term in the above expression, which reduces the complexity but, more importantly, ensures that $h_{kk}^{(n)}$ is positive everywhere. This avoids the previously mentioned problem that H^{-1} is not positive definite upon a non-linear surface.

The third approximation is to calculate h_{kk} from a subset of the cases rather than whole dataset, Le Cun et al. (1998) found that h_{kk} was more strongly influenced by the networks' structure than the subset of training cases used to estimate it, hence it is recommended that only a small sample of cases are used to reduce the cost of the calculation, i.e.:

$$h_{kk} = \frac{\partial^2 E}{\partial w_{(i,j)}^2} = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial^2 E^{(n)}}{\partial w_{(i,j)}^2} \approx \frac{1}{M} \sum_{m=1}^{M} \frac{\partial^2 E^{(m)}}{\partial w_{(i,j)}^2} \quad \text{where } M < N \tag{5.2}$$

Finally, Le Cun et al. (1998) suggested to only perform the update infrequently because the second order derivative's properties changes slowly.

5.5.2 Variance-based Stochastic Gradient Descent

More recently Schaul et al. (2013) proposed the variance-based stochastic gradient descent (vSGD) method that requires no predefined hyperparameters and was demonstrated to outperform several other popular learning rate adaptation methods upon the CIFAR-10 and MNIST dataset. In vSGD the learning rate is adapted according to an

approximation of the inverse of the diagonal Hessian term which is then adjusted according to the variance of the error gradient and the distance to the optimal value of the parameter. More concretely, using an idealized, separable, quadratic loss function, which can be considered as a local approximation for smooth non-quadratic functions, the following form for the learning rate was developed:

$$\eta_i = \frac{1}{h_i} \cdot \frac{\mathbb{E}[\nabla_{w_i}]^2}{\mathbb{E}[\nabla_{w_i}]^2 + \mathbb{V}\text{ar}[\nabla_{w_i}]} = \frac{1}{h_i} \cdot \frac{\mathbb{E}[\nabla_{w_i}]^2}{\mathbb{E}[\nabla^2_{w_i}]} \tag{5.3}$$

where $\mathbb{E}[\cdot]$ and $\mathbb{V}\text{ar}[\cdot]$ denote expectation and variance, respectively. Subsequently when the variance of the error gradient is low, the learning rate is approximately equal to the inverse of the diagonal Hessian term, and when the variance is larger, the learning rate is reduced, lowering the effect of noise from the error gradient. Furthermore, as the learning rate approaches the optimal value for the weight, $\mathbb{E}[\nabla_{w_i}]$ will become smaller, making the learning rate more sensitive to the variance of the error gradient which is expected to reduce the learning rate as the error gradient approaches a minimum. To implement the method practically for online learning approximations to the average and the variance of the gradient of w_i are made by an exponential moving average:

$$\mathbb{E}[\nabla_{w_i}] \approx \bar{g}_i \quad \text{where } \bar{g_i}^{(t+1)} = \left(1 - \left(\tau_i^{(t)}\right)^{-1}\right) \bar{g_i}^{(t)} + \left(\tau_i^{(t)}\right)^{-1} \nabla_{w_i(t)}$$

$$\mathbb{E}[\nabla^2_{w_i}] \approx \bar{v}_i \quad \text{where } \bar{v_i}^{(t+1)} = \left(1 - \left(\tau_i^{(t)}\right)^{-1}\right) \bar{v_i}^{(t)} + \left(\tau_i^{(t)}\right)^{-1} (\nabla_{w_i(t)})^2$$

These approximations allow for a online adaptation. In the above τ is designed to increase the memory of the exponential moving average when the steps taken are small, and decrease the memory when the learning rate is large:

$$\tau_i^{(t+1)} = \left(1 - \frac{(\bar{g_i}^{(t)})^2}{\bar{v_i}^{(t)}}\right) \tau_i^{(t)} + 1$$

The h_i is approximated in a similar exponential moving average, where $h_{ii}^{(t)}$ is estimated using the method proposed in Equation 5.2:

$$\bar{h_i}^{(t+1)} = \left(1 - \left(\tau_i^{(t)}\right)^{-1}\right) \bar{h_i}^{(t)} + \left(\tau_i^{(t)}\right)^{-1} h_{ii}^{(t+1)}$$

Rewriting Equation 5.3 in terms of the above approximations:

$$\eta_i = \frac{1}{\bar{h_i}} \cdot \frac{(\bar{g_i})^2}{\bar{v_i}}$$

When initialising the approximations, Schaul et al. (2013) advocated the use of the

arithmetic average of a small number of cases from the training set and slowing the initial learning by increasing $\bar{v}_i$ by a factor of $K/10$, where K is the total number of parameters. The model was demonstrated to be robust against these initialising parameters. Finally, Schaul and Le Cun (2013) extended vSGD to the mini-batch case, accounting for the reduced variance of the error gradient due to the averaging occurring in the mini-batch method:

$$\eta_i = \frac{1}{\bar{h}_i} \cdot \frac{n(\bar{g}_i)^2}{\bar{v}_i + (n-1)(\bar{g}_i)^2}$$

where n is the number of cases in the mini-batch.

Comparing vSGD to other learning rate adaptation methods, including a static learning rate and the method proposed by Almeida et al. (1998), upon the MNIST and CIFAR datasets, vSGD achieved equal or better error rates while requiring no predefined hyperparameters.

5.6 Conclusion

The learning rate used in gradient descent will affect the speed of convergence of a NN and can prevent a NN from converging. Information can be sought from the slope of the error surface to inform the learning rate to ensure convergence is reached and reduce the number of iterations required to reach convergence. In this regard there exists a trade-off between the time taken to better define learning rates and the reduction of iterations required to reach convergence. In addition, upon smooth surfaces, momentum can be employed to accelerate learning.

Learning rates that are uninformed by the slope of the error surface are often employed for their simplicity, reducing the computational burden of updating the parameters each epoch. Commonly, this method uses a decreasing, global learning rate to allow large adjustments to the parameters initially and then more refined updates towards the end of the training of the NN. Non-adaptive learning rates have featured in recent, successful, large CNNs (Krizhevsky et al., 2012; Cireşan et al., 2010).

Alternatively, the learning rate can be adapted using the first order derivate of the error function on the premise that if the direction of the change is consistent between updates then the size of the step should increase in that direction, otherwise the size of the step should decrease. As the first order error derivative is calculated in order to inform gradient descent, this approach does not increase the amount of computational burden of weight updates substantially. The original DBD method is not suitable for SGD as it is not robust to noisy error derivatives. Almeida et al. (1998) extended upon the DBD, making the provision for SGD. However, the additional complexity of the method

proposed by Almeida et al. (1998) has not been demonstrated to improved accuracy when compared to simpler and uninformed methods when applied to a NN (Schaul et al., 2013; Duchi et al., 2011).

Finally, second order learning rate adaptation schemes are optimal in quadratic bowls and can provide a good approximation to the optimal learning rate (Le Cun et al., 2012). The inverse Hessian matrix is intractable to compute for large NNs and can cause the parameters to diverge, hence approximations to it are sought. In the seminal work by Le Cun et al. (1998) the SDLM learning rate adaptation method is presented, which provided an approximation of the diagonal of the Hessian matrix which would not result in parameters diverging and is tractable to compute. Where the SDLM method employs predefined dampening parameters upon the diagonal Hessian approximation, vSGD informs dampening from the error gradient, requiring no predefined hyperparameters (Schaul et al., 2013). Furthermore vSGD was demonstrated to outperform first order adaptive and non-adaptive learning rate techniques.

Chapter 6

Ensemble Methods

6.1 Introduction

Deep NNs commonly overfit the data they are trained upon as the number of trainable parameters in a network is usually much larger than the number of training cases. Numerous different regularization techniques and combinations thereof are used to address overfitting, including applying a weight penalty, stopping training early or augmenting the data. More recently ensemble methods have been used to mitigate the effect of overfitting. The improvement realised from ensemble methods can be understood using the bias-variance decomposition of the error. Typically ensemble models, such as Random Forests (Breiman, 2001), can be shown to decrease the variance component of error through using an unweighted average of independent predictors' outputs.

In the case of NNs, throughout training, the bias error component gradually decreases, however, as the model increasingly overfits, the variance component increases as illustrated in Figure 6.1. As a result, a deep NN that has overfitted the training data will have low bias and high variance error component. In recent NN models it has been observed that the use of an ensemble does improve the error even though the models are typically not independent due to common influences from architecture for example. These results include state-of-the-art results upon several image datasets such as the ImageNet LSVRC, MNIST, SVHN, CIFAR-10 and NORB (Cireşan et al., 2012b; Wan et al., 2013; Krizhevsky et al., 2012).

The long training time of a single deep NN will practically curtail the number of models in an ensemble although parallelization of model training can be exploited. The Dropout and DropConnect techniques proposed by Hinton et al. (2012) and Wan et al. (2013), respectively, can be construed as efficient methods of training an ensemble of NNs. State-of-the-art results upon the small image datasets mentioned above are achieved with a

combination of regular ensemble of networks and Dropout or DropConnect techniques rather than a single network with Dropout or DropConnect.

In the following section the bias-variance error decomposition is presented for the mean square error and the theoretical bounds of the improvement gained from an ensemble of NNs is discussed. Thereafter, the Dropout and DropConnect methods as algorithms for efficient ensemble models are presented along with their performance upon benchmark datasets.

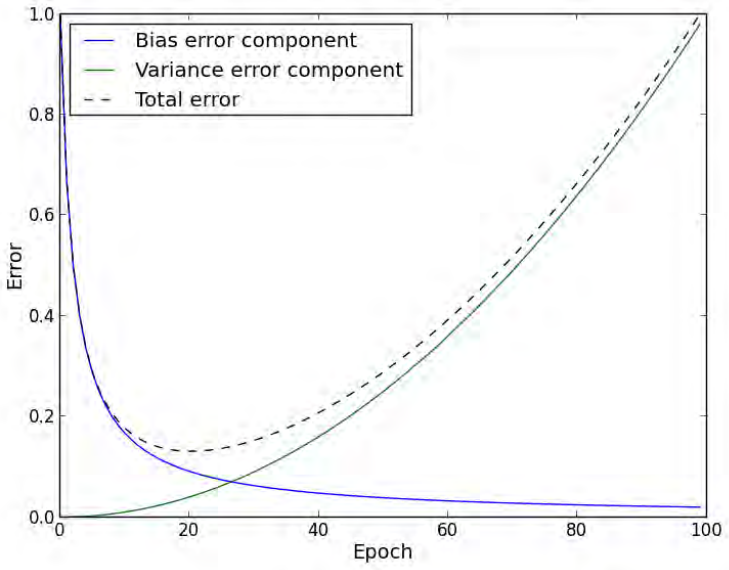


Figure 6.1: During training the bias error component continuously decreases, overfitting is observed when the increase in the variance component offsets the decrease in the bias error component resulting in the total error increasing.

6.2 Bias-Variance Error Decomposition

The bias-variance error decomposition can be used to illustrate the theoretical improvement realised from averaging a set of predictors' results. Typically this is achieved using bagging to produce multiple predictors. Upon testing, the ensemble returns a simple average of the outputs of each of the predictors (Breiman, 2001). In the case of NNs, an alternative approach, whereby randomly initialised weights can be used to train a series of models, which avoids reducing the training dataset for any one network. Through the bias-variance error decomposition, this approach can be demonstrated to reduce the variance error component arising from the initial weights when the models are independent (Horn et al., 2012; Naftaly et al., 1997).

Data upon which a NN is trained will influence it's ability to generalize to other cases. In this regard, a NN trained upon dataset $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ is denoted $f_D(\cdot)$. Then using the mean square error as a measure of the proficiency of the NN, f_D, to classify y given the observations $\mathbf{x}$ (Geman et al., 1992):

$$\mathbb{E}[(y - f_D(\mathbf{x}))^2|\mathbf{x}, D] = \mathbb{E}[(y - \mathbb{E}[y|\mathbf{x}])^2|\mathbf{x}, D] + (f_D(\mathbf{x}) - \mathbb{E}[y|\mathbf{x}])^2$$

The first term on the right hand side is the irreducible error component or noise term and as it does not depend upon $f_D(\mathbf{x})$ it can be disregarded for the purposes of assessing f_D. The first term on the right hand side is the irreducible error component or noise term and as it does not depend upon $f_D(\mathbf{x})$ it can be disregarded for the purposes of assessing f_D. The bias-variance decomposition can then be applied to the expectation of the second term over the different training datasets D to better understand the error (Geman et al., 1992):

$$\begin{aligned}\mathbb{E}_D[(f_D(\mathbf{x}) - \mathbb{E}[y|\mathbf{x}])^2] = &(\mathbb{E}_D[f_D(\mathbf{x})] - \mathbb{E}[y|\mathbf{x}])^2 + \\ &\mathbb{E}_D\left[(f_D(\mathbf{x}) - \mathbb{E}_D[f_D(\mathbf{x})])^2\right]\end{aligned} \tag{6.1}$$

The first term is referred to as the squared bias component which describes the difference between the noiseless outcome and the average model's estimate. The variance component is the second term which describes the model's susceptibility to different training sets.

In the case of an ensemble of NNs the predictor $\bar{f}_D(\mathbf{x}) = \frac{1}{Q}\sum_i^Q f_{i,D}(\mathbf{x})$ for Q randomly initialized NNs is used. Then, by substituting in $\bar{f}$, the bias becomes $\overline{\text{Bias}(f_D(\mathbf{x}))}$, however the variance component using $\bar{f}$ is as follows (Naftaly et al., 1997), where D and $\mathbf{x}$ are dropped for readability:

$$\begin{aligned}\text{Var}(\bar{f}) &= \mathbb{E}[(\bar{f} - \mathbb{E}[\bar{f}])^2] \\ &= \frac{1}{Q^2}\sum_i^Q(\mathbb{E}[f_i^2] - \mathbb{E}[f_i]^2) + \frac{2}{Q^2}\sum_{i<j}(\mathbb{E}[f_i f_j] - \mathbb{E}[f_i]\mathbb{E}[f_j])\end{aligned}$$

Thus:

$$\frac{1}{Q}\text{Var}(f_i) \leq \text{Var}(\bar{f}) \leq \frac{\text{Var}(f_i) + \max_{i,j,i\neq j}(\mathbb{E}[f_i f_j] - \mathbb{E}[f_i]\mathbb{E}[f_j])}{Q} \leq \max_i \text{Var}(f_i)$$

As such, the decrease in the variance error component when substituting in $\bar{f}$ depends on the level of independence of the different networks. Typically the networks are not independent due to similarities in architecture and hyperparameters (Naftaly et al., 1997).

	Error			
Dataset	Best Single	Mean	Ensemble	n models
MNIST	0.29%	$0.44 \pm 0.06\%$	0.23%	35
CIFAR-10	15.63%	$17.42 \pm 1.96\%$	11.21%	8
NORB (10-fold)	3.18%	$3.4 \pm 0.23\%$	2.7%	4
NORB (2-fold)	4.49%	$4.72 \pm 0.16\%$	3.57%	4

Table 6.1: Cireşan et al. (2012b) ensembles of CNNs results upon different small image datasets. The mean error column describes the average error and standard deviation of the individual models. In each case the ensemble model improves upon the average error and the single best error. Note that the number of networks trained for the CIFAR-10 and NORB datasets are too few to infer significance, but rather the standard deviation is included as an illustration of spread of the individual models.

As the advantage of building an ensemble model is realised in a reducing the variance component of the error, the individual models should be trained beyond the minimum error value. This will result in the individual model having a lower bias but much larger variance error component when compared to it's optimal error, which is reduced upon averaging the outputs of the networks (Horn et al., 2012).

A similar approach for different error functions can be taken to demonstrate the improvement of predictions through simple averaging as demonstrated by Tibshirani (1996).

Cireşan et al. (2012b) improved the then state-of-the-art results of numerous image datasets using an ensemble of CNNs, including the MNIST, CIFAR-10, NORB datasets and outperforming human results upon the Traffic signs dataset by a factor of two. Upon the MNIST dataset the ensemble of 35 NNs improved the error of the single best network of 0.28% and the average error of $0.44 \pm 0.06\%$ to 0.23%. Upon the CIFAR-10 dataset the ensemble improved the average error rate of $17.42 \pm 1.96\%$ to 11.21%. Similar improvements in the error rate were realised upon other image datasets, see Table 6.1 for further results.

6.3 Dropout and DropConnect

Hinton et al. (2012) proposed Dropout, a method that randomly drops the output of neurons in a fully-connected NN as a method of regularization that prevents the co-adaption of neurons. If all neurons are present upon each of a limited number of training cases, then they will be collaboratively adjusted through gradient descent to fit the training data. By randomly omitting neurons upon training, neurons in the same layer are no longer always simultaneously adjusted preventing their co-adaptation. Alternatively, Dropout can be interpreted as an efficient implementation of an ensemble method. Random omission of neurons will result in one of 2^M different architectures, for a network

consisting of M neurons in the Dropout layers. In most NNs, M is large and as such it is unlikely that the same architecture will occur twice upon training. Upon testing the 'mean' network is used, whereby no neurons are omitted, rather the outgoing weights are reduced according to the chance with which the preceding neurons were dropped. Thus upon inference, only one network is forward propagated through the network.

Implementation of Dropout is described in Algorithm 1. Upon training, neurons are recommended to be dropped with a 50% chance (Hinton et al., 2012). Then to account for each weight update occurring upon a different architecture, a much higher learning rate coupled with a weight constraint rather than penalty is also recommended. The suggested approach of constraining the weight updates, is to bound the squared L2 value of the incoming weight vector (i.e. $||W||^2$) by a constant k. A drawback of Dropout is that it is found to require a longer training period to converge (Krizhevsky et al., 2012). Dropout is not applied to convolutional layers as the regularization that occurs naturally through weight sharing substantially reduces the network's ability to overfit reducing the benefit of Dropout.

Algorithm 1 Training and Inference with Dropout (Hinton et al., 2012).

Input: Input into layer $\mathbf{x}^{(l-1)}_{(1\times n)}$, Trainable weights $W^{(l-1,l)}_{(n\times m)} = [\mathbf{w}^{(l-1,l)}_{(1)} \cdots \mathbf{w}^{(l-1,l)}_{(n)}]^T$, Activation function f, Dropout chance $1-p$, Learning rate η, Squared length weight constraint k

Feedforward

Generate Dropout mask: $M_{(1\times m)}$ where $M_{1,j} \sim \text{Bernoulli}(p)$

$\mathbf{a}^{(l)} = \mathbf{x}^{(l-1)} \cdot W^{(l-1,l)}$

$\mathbf{x}^{\star(l)} = M \times f(\mathbf{a}^{(l)})$

GradientDescent

$W^{(l-1,l)} := W^{(l-1,l)} - \eta \mathbf{x}^{T(l-1)} \cdot (M \times \frac{\partial E}{\partial \mathbf{a}^{(l)}})$

for i in $1, 2, \ldots, n$ **do**

if $||\mathbf{w}^{(l-1,l)}_{(i)}||^2 > l$ **then**

$\mathbf{w}^{(l-1,l)}_{(i)} := \mathbf{w}^{(l-1,l)}_{(i)} \times \sqrt{k/||\mathbf{w}^{(l-1,l)}_{(i)}||^2}$

end if

end for

Inference

$\mathbf{a}^{(l)} = \mathbf{x}^{(l-1)} \cdot W^{(l-1,l)}$

$\mathbf{x}^{(l)} = p \times f(\mathbf{a}^{(l)})$

Upon several different benchmark datasets including speech recognition, document classification and hand written digit classification, Hinton et al. (2012) demonstrated that Dropout improved the test error of fully-connected NNs. In some cases, these improvements led to new records upon the datasets. Upon the CIFAR-10 dataset, with a CNN, applying Dropout to the final fully-connected hidden layer decreased the test error from 16.6% to 15.6%. Upon the ImageNet LSVRC-2010 dataset in a comparison of a CNN that employed two fully-connected Dropout layers and CNN that did not, the former

achieved an error rate of 48.6% whereas the latter achieved an error rate of 42.4%, improving upon the current state-of-art error rate of 45.7%. The second CNN could not employ the additional fully-connected layers due to substantial overfitting.

Wan et al. (2013) proposed the DropConnect method, which generalizes upon the Dropout method by randomly omitting weights rather than neurons. DropConnect is motivated by an improved theoretical grounding compared to Dropout. While training using DropConnect is similar to Dropout (see Algorithm 2), inference using Dropout estimates the expected output of the model using the 'mean' model described earlier. This method makes the approximation that $\sum_M f(M \times (W \cdot \mathbf{x})) \approx f(\sum_M (M \times (W \cdot \mathbf{x}))$ which does not hold mathematically, considering f is non-linear (Wan et al., 2013). Whereas, in DropConnect, upon inference the expected output of the model is assessed by drawing a sample of the activities from a representative Gaussian distribution and then averaging $f(\mathbf{a})$ to provide a more mathematically robust estimate. For further explanation see Wan et al. (2013). As a result the DropConnect approach is more computationally expensive.

Algorithm 2 Training with DropConnect (Wan et al., 2013).

Input: Input into layer $\mathbf{x}_{(1\times n)}^{(l-1)}$, Trainable weights $W_{(n\times m)}^{(l-1,l)}$, Activation function f, DropConnect chance $1-p$, Learning rate η

Feedforward

Generate DropConnect mask, $M_{(n\times m)}$ where $M_{i,j} \sim \text{Bernoulli}(p)$

$a^{\star(l)} = \mathbf{x}^{(l-1)} \cdot (M \times W^{(l-1,l)})$

$x^{(l)} = f(a^{\star(l)})$

GradientDescent

$W^{(l-1,l)} := W^{(l-1,l)} - \eta M \times (\frac{\partial E}{\partial W^{(l-1,l)}})$

In an empirical assessment Wan et al. (2013) built three different ensembles for the MNIST, CIFAR, SVHN and NORB image datasets. Each ensemble included 5 randomly initialized networks with the same architecture where each of the three ensembles used either Dropout, DropConnect or neither in their fully-connected layer. In all cases the ensemble models improved upon the individual model's average results and, upon the SVHN and NORB dataset, all three ensembles improved upon the state-of-the-art results. Performance upon the different datasets by the Dropout, DropConnect and unmodified CNN ensembles was similar in most cases, see Table 6.2 for specific results. Assessing performance of Dropout and DropConnect applied to single networks, across the MNIST dataset with augmentations, CIFAR-10, SVHN, these methods did not improve upon the unmodified network. However, in the case of the unaugmented MNIST dataset, Dropout and DropConnect improve upon the unmodified network. This result may be due to the smaller number of training cases requiring stricter regularization techniques.

Dataset	Mean error	Ensemble error
	No dropping	
MNIST	$0.77 \pm 0.051\%$	0.67%
Augmented MNIST	$0.30 \pm 0.035\%$	**0.21**%
CIFAR-10	$11.18 \pm 0.13\%$	10.22%
NORB (2-fold)	$4.48 \pm 0.78\%$	3.36%
SVHN	$2.26 \pm 0.072\%$	**1.94**%
	Dropout	
MNIST	$0.59 \pm 0.039\%$	**0.52**%
Augmented MNIST	$0.28 \pm 0.016\%$	0.27%
CIFAR-10	$11.52 \pm 0.18\%$	9.83%
NORB (2-fold)	$3.96 \pm 0.16\%$	**3.03**%
SVHN	$2.25 \pm 0.034\%$	1.96%
	DropConnect	
MNIST	$0.63 \pm 0.035\%$	0.57%
Augmented MNIST	$0.28 \pm 0.032\%$	**0.21**%
CIFAR-10	$11.10 \pm 0.13\%$	**9.41**%
NORB (2-fold)	$4.14 \pm 0.06\%$	3.23%
SVHN	$2.23 \pm 0.039\%$	**1.94**%

Table 6.2: Results of ensemble CNNs by Wan et al. (2013) upon small image datasets, the best results for each dataset is made bold. The ensembles were composed of 5 CNNs each with two convolutional layers, a fully-connected layer and a softmax output layer. In every case the ensemble error improves upon the individual model's errors. Note that there are too few networks trained in each case to infer a significant difference, but rather the standard deviation is included as an illustration of spread of the individual models.

6.4 Conclusion

Ensemble models improve upon the accuracy of models by reducing the variance component in the bias-variance error decomposition. Increasing the independence of NNs in an ensemble will reduce the upper bound upon the variance component of the error.

The Dropout method can be viewed as an efficient ensemble model as it facilitates the aggregation of different architectures without lengthy retraining of multiple NNs. DropConnect improves upon the theoretical basis of Dropout, however is substantially more computationally expensive upon inference, requiring the sampling between layers to correctly ascertain the activities for neurons.

Empirically, upon the small image datasets, a regular ensemble model was demonstrated to improve the accuracy of NNs. The addition of Dropout or DropConnect to the ensemble did not substantially improve the accuracy. However, in the absence of an ensemble model, Dropout have been demonstrated to improve accuracy, upon small and large image datasets.

Chapter 7

Identification of Morphological Features of Galaxies

7.1 Introduction

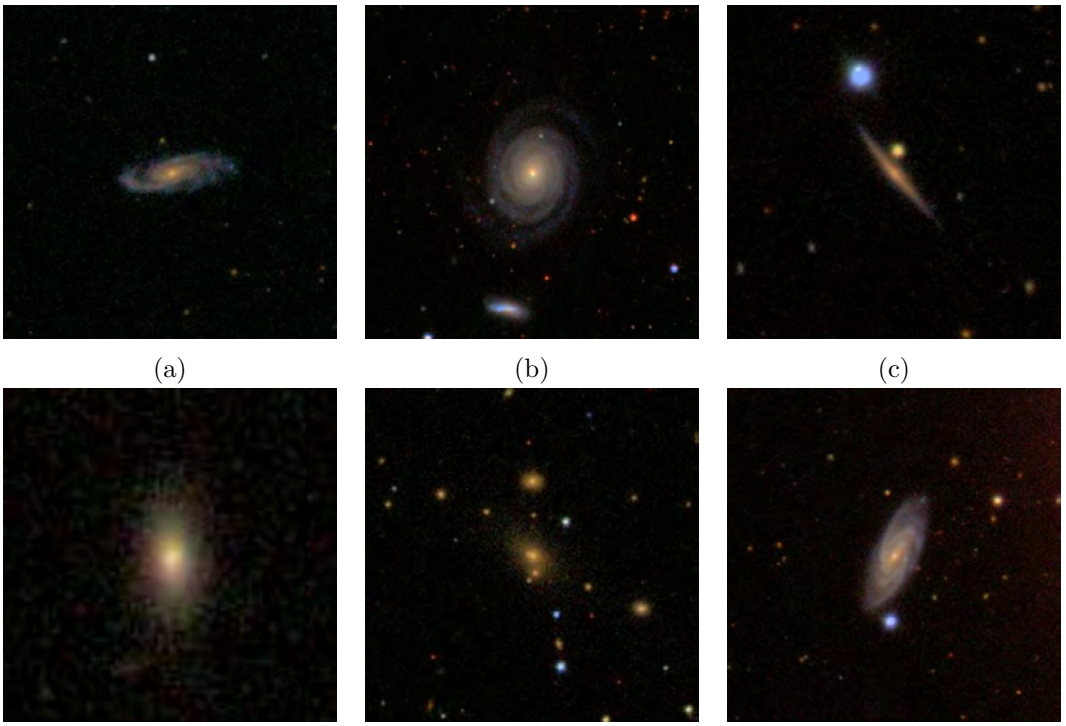
As illustrated in previous chapters, CNNs have been demonstrated to be proficient at image classification. Kaggle, a data classification crowdsourcing platform, in conjunction with Galaxy Zoo (Willett et al., 2013) and Winton Capital[1], presented the problem of identifying morphological features in images of galaxies to assist in classifying the type of galaxy. The problem appeared well suited to CNNs and this chapter describes an implementation of CNNs that achieved a competitive error and demonstrated superior performance when compared to a more traditional classifier.

The chapter is structured as follows, the sections 7.2 and 7.3 describe the Galaxy Zoo competition and the CNN applied to the problem, respectively. Thereafter Section 7.4 the training experience and results of the CNN are presented. Finally, Section 7.5 concludes with suggestions for further improvements as well as a brief discussion of the winning solution.

7.2 Galaxy Zoo Problem Description

An increasing quantity of astronomical data is being collected to assist in understanding the universe and its development. In the context of galaxies, it is helpful to have them classified according to their morphological features. However, as increasingly more galaxies are observed, it is no longer viable for each image of galaxy to be classified by

[1] Competition sponsors



(a) (b) (c)

(d) (e) (f)

Figure 7.1: Examples of images from the Galaxy Zoo 2 dataset. Source: Willett et al. (2013)

experts. Subsequently assistance from automated classification systems is being sought. The Galaxy Zoo competition sought to improve upon current automated classification systems via crowdsourcing.

For the purposes of the Kaggle competition a training set of 61,578 and a test set of 79,975 RGB images were drawn from the original Galaxy Zoo 2 dataset that consists of 304,122 images of galaxies (see Figure 7.1) drawn from the Sloan Digital Sky Survey and labelled via crowdsourcing (Willett et al., 2013). Labelling focused upon morphological features in the data, and averages multiple user classifications per an image to account for classification ambiguity. The objective of the Galaxy Zoo competition was to accurately reconstruct these continuous value labels for each image.

When building the dataset, galaxies were centred in each of the 424×424 pixel images and Willett et al. (2013) selected images to be sufficient size and brightness for the necessary morphological features to be identified, however substantial variation still exists. For example, consider the following:

- The orientation of the galaxy, Figures 7.1 (a), (b) and (c) could all be spiral galaxies, viewed from different perspectives.

- Clutter occurring naturally from other galaxies in the foreground and background, for an example consider Figure 7.1 (e).
- Artifacts arising from the image capturing device, such as edges arising from splicing images together. Consider Figure 7.1 (f).
- Between a certain threshold the scale of galaxies in the images was not controlled, consider Figures 7.1 (e) and (b).

The labels for each image were generated by crowdsourcing, whereby a volunteer would assign an image to different categories according to a series of questions, where each successive categorization question was determined by the answer to the former according to a hierarchical decision tree type structure that contained a total of eleven questions upon the presence of morphological features in an image. See Table 7.1 for the full list and structure of questions.

To capture the uncertainty associated with the assignment of ambiguous images to categories, multiple individuals would be shown the same image and the results were averaged. The expertise of the volunteers was not controlled, but Willett et al. (2013) found that the aggregated response agreed strongly with expert classification. The final modification upon the labels was to normalize the sum of each question to one and then multiply each child question by the value of its parent's response. E.g. for an image, the total of the response value of question 02 would be equal to the average number of responses that opted for 'Features or Disk' upon question 01. This modification placed more emphasis on correctly reconstructing the initial questions of the decision tree, which agrees intuitively with the structure of the decision tree. Note that for question 06, the values are not multiplied by the parent value, rather they are left normalised at one as this question was always asked. This resulted in each of the training images having label vector $\mathbf{Y}_i$, composed of 37 elements for each of the total possible answers to the eleven questions. Note each vector element was from the continuous range $[0, 1]$. i.e. $Y_i = (y_{(i,1)}, \ldots, y_{(i,37)})^{\mathrm{T}}$ where $y_{(i,j)} \in [0, 1] \forall i, j$.

The Root Mean Square Error (RMSE) function was used as the error measure of the difference between the predicted output and the actual output and was defined as follows:

$$\sqrt{\frac{1}{N \times M} \sum_{i=1}^{N} \sum_{j=1}^{M=37} (\hat{y}_{ij} - y_{ij})^2}$$

where $\hat{y}_{ij}$ and y_{ij} are the predicted and actual values for the jth element in the ith output vector and N and M are the total number of images and total number of elements in each output vector.

#	Question	Response	Next
01	Is the galaxy simply smooth and rounded, with no sign of a disk?	Smooth	07
		Features or Disk	02
		Star or Artifact	End
02	Could this be a disk viewed edge-on?	Yes	09
		No	03
03	Is there a sign of a bar feature through the centre of the galaxy?	Yes	04
		No	04
04	Is there any sign of a spiral arm pattern?	Yes	10
		No	05
05	How prominent is the central bulge, compared with the rest of the galaxy?	No bulge	06
		Just noticeable	06
		Obvious	06
		Dominant	06
06	Is there anything odd?	Yes	08
		No	End
07	How rounded is it?	Completely round	06
		In between	06
		Cigar-shaped	06
08	Is the odd feature a ring, or is the galaxy disturbed or irregular?	Ring	End
		Lens or arc	End
		Disturbed	End
		Irregular	End
		Other	End
		Merger	End
		Dust Lane	End
09	Does the galaxy have a bulge at its centre? If so, what shape?	Rounded	06
		Boxy	06
		No Bulge	06
10	How tightly wound do the spiral arms appear?	Tight	11
		Medium	11
		Loose	11
11	How many spiral arms are there?	1	05
		2	05
		3	05
		4	05
		More than four	05
		Can't tell	05

Table 7.1: Galaxy Zoo 2 question template. Note that the number of the question does not infer sequence, rather refer to the Next column to determine what question would follow for a given answer. Depending on the response, some questions would not be asked, with the exception of question 1 and 6, which were always asked. Source: Willett et al. (2013)

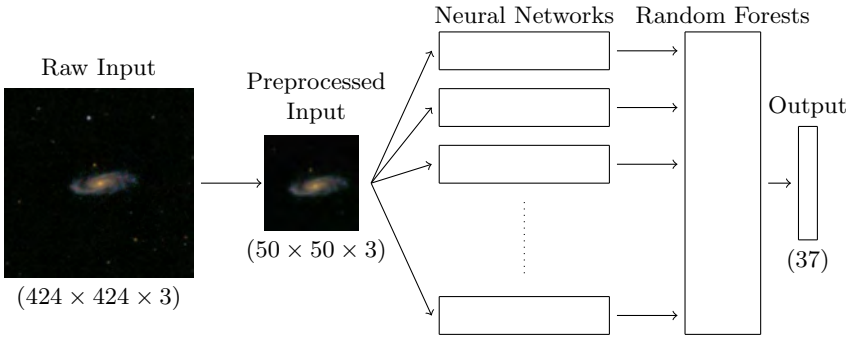


Figure 7.2: Architecture of complete classification system

7.3 Methodology

Several different architectures and adaptations were explored to improve the model's performance. The final model was composed of two parts, the first part was a collection of eleven NNs, one per question, each of which used a preprocessed, augmented image as input. The second part of the model consisted of a Random Forest which used as input, the outputs of all of the NNs and returned the final 37 long vector. Figure 7.2 illustrates the complete model. This section discusses some of the key components of the model.

The intention of this design was to allow for the NNs to have sufficient capacity to accurately classify galaxies while taking advantage of multiple graphics cards. The eleven questions in the decision tree, which are occasionally referred to as classes, provided a natural division of the output to train separate networks upon. The classes outputs are dependent by construction of the decision tree, hence the response in one class would be helpful to another class. For example, the output to question 04, "Is there any sign of a spiral arm pattern?" is helpful for question 11, "How many spiral arms are there?". Two components were added to take advantage of this. The first was to add a Random Forest onto the outputs of the NNs. The second was to adapt the softmax function upon which each CNN was trained, which is discussed further in Section 7.3.2.1.

One of the shortcomings for CNNs is that searching for the optimal architecture is time consuming due to the large number of different combinations and computation time taken to fit a CNN. Subsequently, while there was some preliminary exploration of different architectures, emphasis was placed upon building a large network while focusing upon regularising the network.

7.3.1 Data Preprocessing and Augmentation

To make the problem tractable the data was first preprocessed, then, to prevent overfitting and assist in building rotational invariance, the data was augmented through random rotations.

The raw image data of RGB images with a resolution of 424×424 pixels was too large to input into a CNN as is, furthermore by inspecting the data it was observed that images contained a large amount of empty space around each galaxy. In order to reduce the size of the input each image was reduced to a 84×84 pixel image using bi-cubic interpolation (Keys, 1981) and then cropped to the middle 50×50 pixels. The three RGB channels were maintained, subsequently the input to the CNN was a 3D matrix of size $50 \times 50 \times 3$.

In the images, no attempt was made to orientate the galaxies resulting in substantial rotational variance between similar galaxies, for example, consider Figure 7.1 (a) and (f). Furthermore, while CNNs have an inherent scale and translation invariance, they do not account explicitly for rotation invariance. To improve the model's robustness against rotational variances, the data was augmented by rotating each image randomly by $0, 90, 180 or 270$ and thereafter, an additional set of images were produced by mirroring each of the rotated images, increasing the number of training examples eight-fold.

7.3.2 Description of Neural Network Architecture

To simplify implementation, each of the eleven networks for the eleven questions had identical architectures and were initialized in the same manner. The architecture comprised of a hardwired layer that produced three distinct channels, to each of which two convolutional layers were applied. The output was then concatenated and inputted into two fully-connected Dropout layers and finally an output layer using the modified softmax output. See Figure 7.3 for an illustration of the networks' architecture.

The three separate channels were defined by their manipulation of the input data. The RGB channel did not adjust the input channel. The remaining two channels calculated the horizontal and vertical gradients in the image convolving their respective 3 by 3 Scharr gradient operators (Scharr, 2000) across the average of the image's colour channels, in order to improve the overall network's performance (Ji et al., 2013). Thereafter each of three channels were kept separate for the convolutional layers and recombined in the fully-connected layer.

Specifically, in the first convolutional layer C1, in the RGB channel the kernel was $25 \times 25 \times 3$ large and 40 feature maps were returned. In the gradient channels a kernel

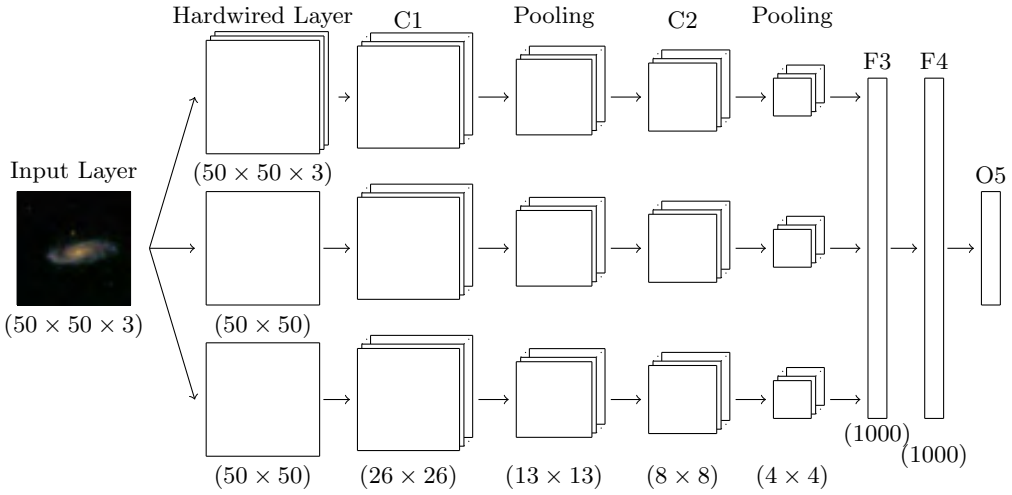


Figure 7.3: Architecture of the CNN used in the Galaxy Zoo competition. Cx refers to a convolutional layer, Fx refers to a fully-connected layer and O5 is the output layer. The size of the dimensions of each layer's output is included beneath each layer. Refer to Section 7.3.2 for the size of the receptive fields and further specific details of the network's architecture.

of size 23×23 was used and 20 feature maps were produced for each of the channels. Following C1, max pooling was applied with a non-overlapping window size of 2×2. The resulting output had a dimensionality of 13×13 for all channels, and 40 feature maps for the RGB channel and 20 feature maps for each of the gradient channels. In the second convolutional layer, C2, kernels received input from all feature maps in their channel, had a size 6×6 and 6×6 and 80 and 10 feature maps were produced across the RGB channel and the gradient channels, respectively. The output was max pooled across 2×2 non-overlapping windows. The resulting 80 4×4 feature maps for the RGB channel and 10 4×4 feature maps for the gradients channels were flattened to create a vector of 1600 elements, these were fully-connected to a 1000 neurons in layer F3. An additional fully-connected layer, F4, which also consisted of 1000 neurons, followed F3. Finally, output layer O5, employing the modified softmax was connected to F4. Dropout was applied to each of the fully-connected layers.

Weights were initialized across all layers according to Glorot and Bengio (2010). In the convolutional layers, the learning rate was initialized at 0.01 and an exponential decay rate of 0.993 was applied at each epoch. Momentum was applied to all layers with $\alpha = 0.9$. In the Dropout layers, activation values were dropped with a 50% chance, a learning rate of 1 was used initially and decayed, exponentially, using a rate of 0.993. The parameter update constraint recommended by Hinton et al. (2012) was implemented to prevent the parameter updates from being too large in the Dropout layers.

7.3.2.1 Modified Softmax

Considering that the output at each class level was normalized to sum to one before being multiplied by the parent node value, a softmax output layers seemed appropriate as it would normalize the output to one.

In the case where the questions total was normalized to the value of it's parent, rather than to re-normalize each of the class level outputs to sum to one, an additional output was fitted to each class that was the difference between the sum of the actual outputs and one to avoid overemphasis on cases that weren't relevant. For example, let $\mathbf{y} = (y_1, \ldots, y_n)^{\mathrm{T}}$ represent the outputs of a class. Adapting it to the above methodology results in $\tilde{\mathbf{y}} = (y_1, \ldots, y_n, \tilde{y}_{(n+1)})^{\mathrm{T}}$ where $\tilde{y}_{(n+1)} = 1 - \sum_{i=1}^{n} y_i$. This approach allowed the use of a softmax function as an output and had the benefit of allowing the network to learn about the parent node and the relevancy of the image to the question. This additional output by each network would be taken into account when the Random Forest was applied to the outputs of the NNs.

7.3.3 Random Forest

The final component of the model was to combine the outputs from the NNs and use them to grow a Random Forest (Breiman, 2001) (See Appendix A for further details on Random Forests). Each network returned one more than the number of outputs from each question due to the modified softmax output resulting in a total of 48 outputs. Additionally, each of the eight rotations of the image were used in order to improve the model's robustness to rotational variance, hence the total size of the input vector for the Random Forest (RF) was 384.

To capture interclass information described in Section 7.3.1, these predictions and their labels were used to train a RF. For the final model, 37 RFs of 300 fully grown trees were grown. Each for a separate output class.

7.4 CNN Training and Results

Each net was trained upon a Nvidia Tesla M2090 GPU and an Intel Xeon 2.0GHz core and implemented using the Theano Python library (Bergstra et al., 2010). For training the CNN, the labelled training data was divided randomly into training and validation sets consisting of 51,000 images and 10,000 images, respectively, to allow for a sufficiently large validation set for the RF to train upon. Each epoch consisted of training across the training set after each image had been assigned a random rotation. CNNs were

trained for either 620 or 1000 epochs, for a total of 31.6 million or 51 million images. The number of epochs trained for was constrained by time and if the validation error was observed to no longer decrease after 620 epochs training was then stopped. The validation error did not increase at any point for any class, suggesting that the network did not overfit. Training took approximately 40 hours for 1000 epochs.

Once training of the networks was complete, the networks predicted each of the validation cases for each of the eight different rotations, upon which the RF was trained. To determine the improvement made by fitting the RF upon the CNN's outputs, the RMSE of CNNs without the RF was also calculated. For reference, a multi-output RF (Linusson, 2013) with ten trees was grown upon the training data, which was preprocessed and augmented in the same manner as in the CNN. The results of the model at each stage are presented in Table 7.2.

The predictions upon the competition's test set were made in the same manner as the validation error and a final RMSE of 0.0963 was achieved.

Table 7.2: RMSE upon validation dataset. Note that for the RF, which was built upon the CNN predictions for the validation dataset, the Out of Bag error is provided.

Algorithm	**RMSE**
11 channel CNN without the RF	0.09795
11 channel CNN with RF	0.09696
Multi-output RF	0.13196

7.5 Conclusion

The model presented here demonstrates superior performance when compared to a multi-output RF and provides a competitive result upon the test set, ranking 35^{th} of 326 competitors and compared to the winning solution which achieved a RMSE of 0.07492 (Dieleman, 2014). Other competitive models used CNNs as well as SVMs and Gradient Boosted Models (GBMs) that were applied to carefully extracted features (Harvey et al., 2013).

The addition of the RF upon the CNN improved the result. This was expected as the CNN were trained upon outputs of specific questions, but the questions' outputs were dependent upon each other due to the decision tree type structure that was posed to the labellers.

The model presented by Dieleman (2014) employed a similar methodology to that described above, but improved upon a number of aspects of the model. These improvements included the use of an ensemble of CNNs with many more parameters, trained upon many more augmented training cases and an improvement in computing speed of the Theano library.

Further improvements upon the model presented here could have been realised by adjusting the CNN architecture to accommodate different feature complexities in each class. Specifically, the error rate of certain classes' CNNs quickly stopped improving which may be because the CNN architecture used did not have enough capacity to sufficiently identify the features necessary for that class. However given the time constraint imposed by the competition and the difficulty of exploring different NN architectures, it was not possible to investigate the cause of this behaviour.

Chapter 8

3D Convolutional Neural Networks

8.1 Introduction

3-dimensional (3D) CNNs, first proposed by Yang et al. (2009), extend the 2-dimensional (2D), image classification CNNs to video classification by simply including an additional temporal dimension. Video data is expected to improve classification of actions as some discriminative features will arise from motion. While good accuracy can be achieved using single frames, increasing the number of frames in a clip when classifying an action has been found to improve the accuracy of a classifier, although with rapidly diminishing returns (Schindler and Van Gool, 2008; Ji et al., 2013).

The similarity between video and image data results in the rationale behind the constraints imposed upon the architecture for regular 2D CNNs applying to 3D CNNs. Specifically, features are still expected to be constrained to local regions in both spatial and temporal dimensions, and the variances in video data are expected to be inclusive of those in image data. Furthermore, the addition of the temporal dimension will substantially increase the dimensionality of the input. As such the application of convolutions to the network architecture is still justified. Downsampling, as with 2D CNNs, is still required to sufficiently decrease the dimensionality of the input such that the network remains tractable to update.

Video classification is commonly accomplished using specialised feature extractors that are robust to variances in the data and a regular classification method (Weinland et al., 2011). A wide array of different feature extractors and classification techniques exist and, depending on the video data, different methods may be applicable (Weinland et al.,

2011). In the same manner as in image recognition, CNNs are advantaged over these techniques, as they perform the feature extraction and classification simultaneously and do not need to be substantially modified to be applied to new data. However there has only been limited development of 3D CNNs (Baccouche et al., 2011; Ji et al., 2013; Le et al., 2011) and they are typically outperformed by the afore mentioned specialised feature extractor combined with a regular classification technique.

The following section briefly examines the parameterization of 3D CNNs. The work presented in this research expands upon the work done by Ji et al. (2013) which is examined in further detail in Section 8.3. The final section concludes the chapter and discusses a couple of alternative 3D CNNs.

8.2 3D Convolutions and Subsampling

The receptive field of each convolutional neuron is defined across two spatial dimensions and a temporal dimension. Formally, a neuron in the convolutional layers is defined as:

$$x_{(r,t,s)}^{(n,l)} = f(a_{(r,t,s)}^{(n,l)}) = f\left(\sum_{(i,j,k,m)} w_{(i,j,k)}^{(n,m,l)} x_{(r+i,t+j,s+k)}^{(n-1,m)} + w_{(0)}^{(n,l)}\right)$$

The additional subscripts s and k accommodate the temporal dimension for the input and the weights, respectively. Otherwise the parameterization is the same as the convolutional neuron of 2D CNN. Downsampling in between convolutional layers is usually implemented only across the spatial dimensions, as the temporal dimension is typically small enough such that convolving across it, will sufficiently reduce its size (Ji et al., 2013).

8.3 Example of a 3D CNN's Architecture

Extending on the work done by Yang et al. (2009) and Jhuang et al. (2007), Ji et al. (2013) developed a 3D CNN to classify human action in video. The network was applied to the TRECVID (Smeaton et al., 2006) and KTH (Schuldt et al., 2004) datasets . The former consists of labelled security footage of London's Gatwick Airport consisting of multiple subjects, often performing actions at the same time. The KTH dataset consists of posed videos of single subjects performing actions upon homogeneous backgrounds, see Section 9.2 for further detail. The 3D CNN model outperforms common video classification methods as well as a 2D CNN when applied to the TRECVID data and provided a competitive error rate upon the KTH data. The focus of the work performed

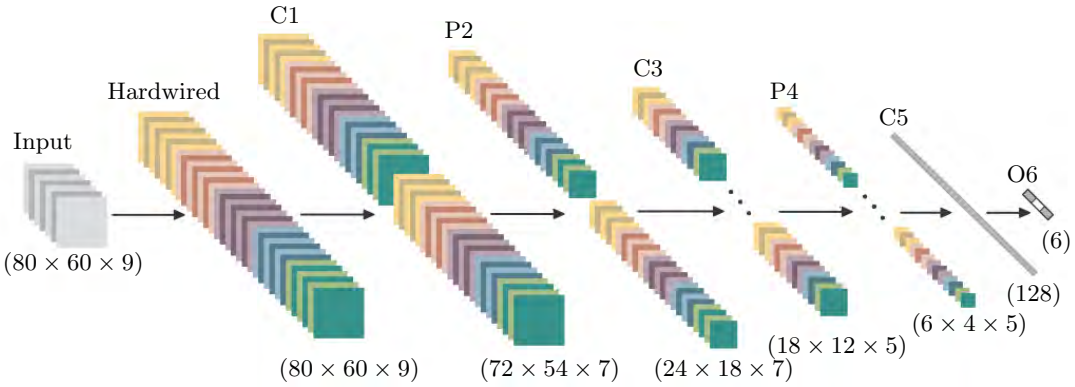


Figure 8.1: Illustration of the 3D CNN developed by Ji et al. (2013). The different channels are illustrated in different colours, with frames illustrated as squares. Note that the number of squares does not reflect the actual number in the channel. Cx, Py and Oz indicate a convolutional, pooling and the output layers respectively. The dimensions of the output of feature maps of a layer with respect to width, height and time are included below the layer.

by Ji et al. (2013) was upon the TRECVID dataset, however given the relevance of the work performed upon the KTH data, the architecture of the 3D CNN applied to the KTH dataset is discussed here. The architecture used upon each dataset was similar, with adaptations applied to the kernel size to accommodate the different dimensionality of the data.

Ji et al. (2013)'s network consisted of a hardwired layer and six trainable layers. The hardwired layer divided the input into five different channels according to the different features from prior knowledge that Ji et al. (2013) had found to improve performance. These five channels included horizontal and vertical gradient, horizontal and vertical optical flow and the original video as is.

The trainable layers were composed of two convolutional and pooling layers, followed by a convolutional layer, that, due to the size of the receptive field and the input, functions exactly like a fully-connected layer, and an output layer which employed a softmax output unit. An illustration of the network is depicted in Figure 8.1.

In the first convolutional layer, kernels with a size of $9 \times 7 \times 3$ with respect to width, height and time were convolved upon the input of $80 \times 60 \times 9$. For each channel two feature maps were trained, resulting in total of 1,900 trainable parameters. The succeeding pooling layer applied a common trainable parameter and bias to the simple average of non-overlapping 3×3 local spatial regions, allowing trainable parameters to vary across different frames resulting in 132 trainable parameters. The second convolutional layer's kernel size was $7 \times 7 \times 3$, and three feature maps were trained per a feature from the income layer, resulting in 4,440 trainable parameters. The second pooling layer, used the same structure as the first pooling layer, but as a result of the increased feature

maps had 276 trainable parameters. The third convolutional layer functions exactly the same as a fully-connected layer with 128 neurons. This is because it's kernel, applied only to the spatial dimensions, had a size of 6×4 which is the same size as the input maps, furthermore, every neuron is connected to every feature map and temporal map. This results in 441,600 trainable parameters. The final layer is a softmax layer that is fully-connected to the previous layer and has 768 trainable parameters.

In each neuron, the hyperbolic tangent function is used as an activation function and the learning rate is adapted using the SDLM (See Section 5.5.1).

8.4 Conclusion and Remarks

3D CNNs are a natural extension to 2D CNNs, and, while 2D CNNs are well recognised at image classification, 3D CNNs have not been widely used for video classification. This is likely in part due to there being fewer large video datasets (Weinland et al., 2011) and the higher dimensionality of videos, rendering CNNs less practical when compared to more efficient, but expert encoded, explicit feature extraction and classification models.

When applied to the KTH dataset the 3D CNN developed by Ji et al. (2013) achieved an average error rate of 9.8% across a 5-folds of the data. While not superior to common video classification techniques, the result is competitive (See Section 9.5 for further details).

Upon the TRECVID, Ji et al. (2013) applied two extensions to the basic model described above. First, in order to regularize the network, Ji et al. (2013) trained the network on auxiliary outputs that were calculated from a sequence of frames larger, but inclusive of the input frames. Auxiliary outputs were calculated using a bag of words features from a SIFT features extracted from the raw and motion edge history images. This additional regularization component was found to improve results, although not substantially. The second extension involved combining several models, each with a different architecture, together in an ensemble. This extension returned substantially improved results. Additionally, the 3D CNN was demonstrated to outperform a 2D CNN, suggesting that important discrimination features were found across the temporal dimension.

There are several other examples of 3D CNNs and models similar to 3D CNNs applied to the KTH dataset. Baccouche et al. (2011) used the output of a 3D CNN for short sequences as input to a Recurrent NN (RNN), which are NNs that have been adapted to classify sequential information. Using an architecture consisting of two convolutional and pooling layers, a convolutional layer, a fully-connected layer and finally an output layer, Baccouche et al. (2011) achieved an average error rate of 8.96% across 5-folds of

the data without the RNN. Applying a RNN to the set of outputs of the 3D CNNs for the set of sequences that made up a video, the error rate was reduced to 5.61%. Notably, the KTH data was preprocessed to extract a narrow bounding box containing the subject to reduce the size of the input, furthermore, the architecture included a rectification layer after each of the first two convolutional layers, in which the absolute value of the input is returned. Alternatively, Le et al. (2011) demonstrate the use of the Independent Subspace Algorithm (ISA) (Hyvärinen and Hoyer, 2000) as a method to perform unsupervised pretraining upon a 3D convolutional network type structure. Upon the KTH dataset they achieve an error rate of 6.1%.

The architecture developed by Ji et al. (2013) was used to inform initial design choices as it did not require localization as a preprocessing component such as the model presented by Baccouche et al. (2011), which is a problem in and of itself. Furthermore, the architecture had provided competitive performance amongst 3D CNN models.

Chapter 9

Methodology and Discussion

9.1 Introduction

In this chapter, the implementation of a 3D CNN and results achieved upon the KTH data are described. The aim of this research (See Chapter 1) is to develop a 3D CNN that employs recently developed 2D CNNs methods to present an up-to-date human action recognition 3D CNN model. The implementation of the final model includes numerous choices which are detailed in this chapter. These choices include the choice of hyperparameters to define a CNN, the treatment of the data, the method of evaluation and the choice of implementation environment. All of which will influence the outcome of the final model (Gao et al., 2010; Ji et al., 2013; Zeiler and Fergus, 2014). This chapter details each of these choices and provides rationale behind them where research in previous chapters have not.

To begin, the KTH dataset, which consists one of 25 people performing one of six actions in each of approximately 600 videos, is introduced in Section 9.2. The method of evaluation, which is not standardized for the KTH dataset, is discussed. For this research a 5-fold cross-validation evaluation of the error is proposed (See Section 9.2.2), in order to remain comparable to Ji et al. (2013). The KTH dataset is preprocessed to make it more suitable to be inputted into a CNN. Importantly the data is downsampled to 9-frame clips with a spatial dimension of 60×80 such that the task is tractable. Finally, to reduce the effect of overfitting, the data is augmented as described in Subsection 9.2.3.

Section 9.3 describes Theano, a functional, symbolic, mathematical Python library that was used to program the 3D CNN. Theano enables an optimized implementation of the code upon the GPU such that the 3D CNN presented later in the chapter can run in a reasonable period of time.

The architecture of the models used in this research are detailed in Section 9.4. The theoretical implications of different architectures are not well understood, subsequently the architecture of CNNs is typically informed by suggestions from the literature, such as those made by Zeiler and Fergus (2014). In this research, the model employed by Ji et al. (2013) is used as the initial inspiration for the architecture. Thereafter several extensions made by recent work are applied, such as the use of Dropout upon the fully-connected layers (Hinton et al., 2012). Three different architectures referred to as Net-A, Net-B and Net-C are suggested. The code for the architectures is included in Appendix B.

In Section 9.5 the results are presented and discussed. The emphasis of the results is the comparative performance of the model presented here to previous work done by Ji et al. (2013). In addition, insight into the performance of the models across different divisions of the data as well as across the different classes and scenarios is also given.

The final section concludes the chapter.

9.2 Description of KTH Dataset

The KTH dataset (Schuldt et al., 2004) is a widely used, posed human action recognition video dataset. It consists of approximately 600 videos of human subjects performing actions in front of homogeneous backgrounds. Each video can be further divided into four shorter videos as the action in each video is repeated four times. These shorter videos are referred to as sequences to distinguish them from the original videos. Examples of frames from different sequences are illustrated in Figure 9.1.

While widely used, evaluation techniques upon the KTH have not been formalised and different methods of evaluation can lead to substantially different results (Gao et al., 2010). As such care needs to be taken when defining the method of evaluation to remain comparable. In this research, 5-fold cross-validation upon the videos is evaluated to ensure that the results are comparable to Ji et al. (2013).

In the data's raw format, each sequence has a resolution of approximately 160×120 grayscale pixels and is approximately four seconds in length at 25 frames per second. Subsequently, to feasibly apply a CNN to the problem, the input needs to reduced in size. In addition to preprocessing the data to a tractable size, the data was augmented to reduce the impact of overfitting.

In the following subsection the original KTH data is described in depth, thereafter, in Subsection 9.2.2, the method of evaluating models upon the KTH data is discussed.

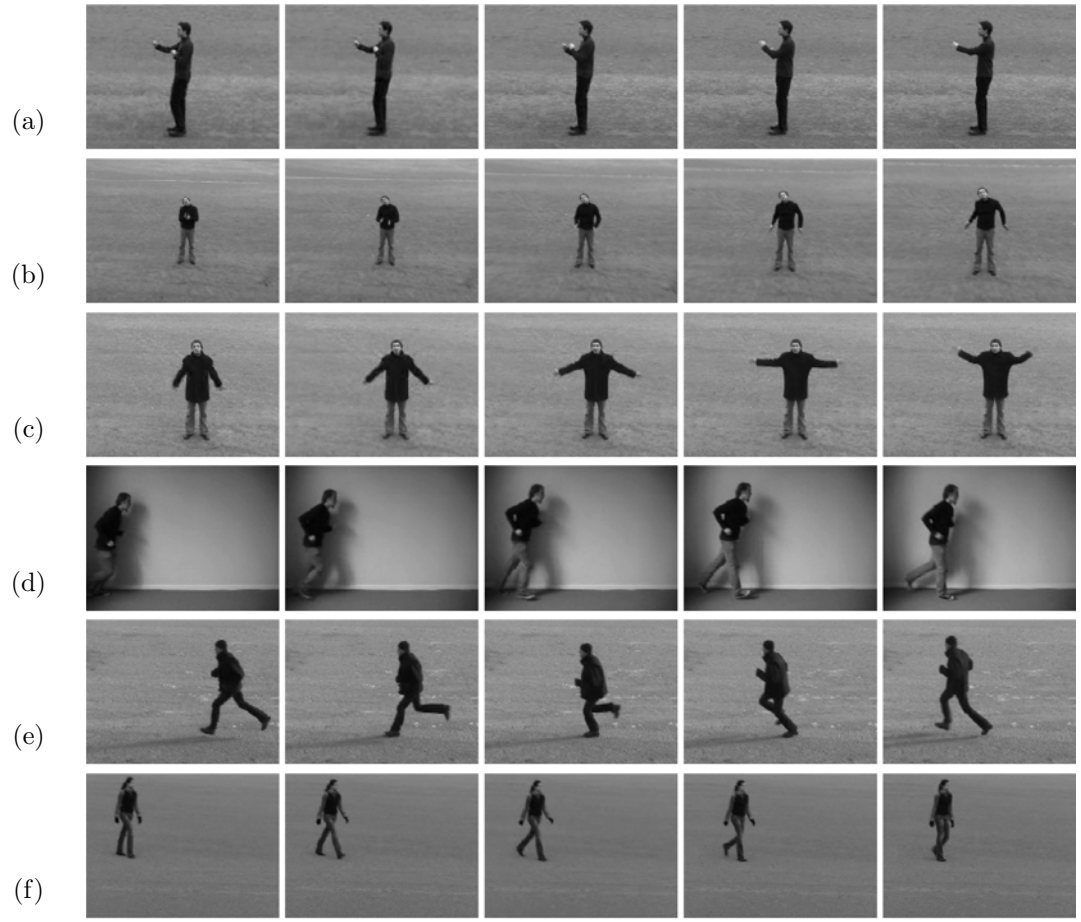


Figure 9.1: Examples of each of the actions and scenarios in the KTH data. (a) - (f) illustrate, respectively, boxing, handclapping, handwaving, jogging, running and walking. The standard outdoors scenario, $s1$, is portrayed in (a) and (e) and the outdoor with changing scale, $s2$, is portrayed in (b) and (f). The subject in (b) features again in (c) to illustrate, $s3$, the scenario whereby the subject has different clothes. Finally, the indoor scenario is portrayed in (d). Source: Schuldt et al. (2004)

Subsection 9.2.3 discusses the preprocessing steps that were applied to make the data tractable in the context of CNNs as well as augmentations that were applied to mitigate overfitting.

9.2.1 Overview of KTH Dataset

The KTH dataset consists of 25 different subjects performing one of six different actions in each of four different scenarios to produce 599 videos[1]. In each video, the action is usually repeated four times, such that there are 2391 different sequences. The six actions are boxing, handclapping, handwaving, jogging, running and walking. The four different scenarios are: outdoors ($s1$), outdoors with changing scale ($s2$), outdoors with different clothes ($s3$) and indoors ($s4$). In each scenario the background is homogeneous

[1] One video is corrupted and subsequently unreadable.

and no other humans are included in the scene. The camera is approximately fixed, with the subject in the centre of the frame when performing a stationary action (i.e. boxing, handclapping, handwaving). When performing a moving action (i.e. walking, jogging and running) the subject starts out of the picture and passes through the centre of the scene. With the exception of scenario $s2$, the background and subject remain at a fixed distance from the camera. For scenario $s2$ the scaling of the subject is achieved in one of two manners. If the action is stationary, the camera zooms in and out upon the subject, subsequently the scale of the background changes as well. Alternatively, if the action includes movement, the subject enters the scene from a corner and leaves the scene by the diagonally opposite corner, thereby introducing scale variation due to the depth of the scene. In this case the background does not scale with the subject. See Figure 9.1(f) and Figure 9.1(b) for examples of the two methods of varying scale. For example, the model could be trained and tested upon the entire dataset or each scenario could be considered a separate problem and the model can be trained and tested separately upon each scenario.

9.2.2 Evaluation Upon the KTH Dataset

While widely used, the KTH dataset has no well defined method of evaluation, which prevents results from being comparable. Specifically, there are three unstandardized decisions that will influence the results. Firstly, the classification of the action can be performed upon the videos or upon the sequences. Secondly, there is no standardized test set, rather the method of division of the dataset into a training and test set is left to the researcher. Finally, the training and testing of the model can be performed upon each scenario separately or a combination thereof.

The choice between training and testing upon the entire video or the sequences varies between different research, however, by classifying the entire video rather than separately classifying each of the four sequences that a video is composed of, Gao et al. (2010) found that results generally improve.

The authors of the dataset, Schuldt et al. (2004), divide the data into training, validation and test sets according to the person in the scene such that the same person does not feature in more than one subset. While the division of the dataset by person is common practice, which subjects are allocated to which dataset is not standardized. This allocation can have substantial impact upon the final accuracy of the model. Gao et al. (2010) demonstrated that with different divisions of the 25 subjects into a test set of 9 persons, a validation set of 8 persons and a training set of 8 persons the results of their model could change by $\approx 9\%$ in absolute value, when sequences are classified

or $\approx 5.6\%$ in absolute value when the entire video is used. In the afore mentioned experiment only 30 different allocations were considered out of approximately 26 billion unique methods of assigning subjects into the different datasets, however the experiment serves to illustrate that the allocation could influence the result. To mitigate this effect, models are typically reported using a n-fold cross-validation error, whereby the average accuracy of a model trained and tested across n different divisions of the data is reported. Gao et al. (2010) suggested the use of leave-one-out cross-validation, where, in the case of the KTH dataset, the average error across 25 different runs is performed, each using a single person as a test case such that each subject features once as a test case. The advantages of this method are that the testing conditions are consistent and the size of the training set is maximized. However, in the case of NNs which commonly take multiple days to train, this approach becomes impractical. As a compromise, a 5-fold cross-validation test error, using a test set of 9 subjects is suggested.

The scenarios upon which the model is trained upon and tested upon will influence the results. For example the model built by Gao et al. (2010) when trained upon $s1$ and $s3$ performed worse when tested upon $s2$ when compared to $s4$. Typically all scenarios are trained upon and tested upon although this choice can depend upon the objective of the test, for example the ability of a model's features to generalize can be tested when testing them upon an unseen scenario.

To remain comparable to work done by Ji et al. (2013), the results in this research are reported using a 5-fold cross-validation upon the videos. A similar division by people is made, whereby for each fold, 9 subjects are used for testing and the remainder for training. Training and testing were performed upon all scenarios simultaneously to remain consistent with the test conditions presented in Ji et al. (2013).

9.2.3 Preprocessing and Augmentation

The original KTH dataset needed to be preprocessed to improve its conditioning and reduce the dimensionality to a more manageable size. Furthermore, upon training, the data was augmented to reduce overfitting. This section examines the details behind each process.

The original KTH dataset consists of videos with a spatial resolution of 120×160 grayscale pixels. This resolution is downsampled to a spatial resolution of 60×80 whereby each 2×2 non-overlapping region in each frame is reduced to the unweighted average. Thereafter each pixel is divided by 255 such that each pixel value is then in the $[0, 1]$ range. Controlling the range of the inputs is primarily done for convenience. If not performed and the weights were not initialized correctly for large input values, sigmoidal

neurons would start saturated, slowing learning initially. The final preprocessing step is to centre the data by subtracting each frame in the sequence by the average of all frames in a sequence resulting in each pixel value being in the $[-0.5, 0.5]$ range. See Figure 9.2 for an example of this preprocessing.

Upon training, each video is divided into 9-frame clips and then augmented immediately prior to being inputted into the network. Augmentations were carefully introduced such as not to interfere with the interpretation of actions. For example, no noise to the frame rate was introduced as the speed at which the action is performed is helpful in determining the type of action. Prior to dividing the sequence into the 9-frame clips to be consistent with the input size of the network, a random number of the first frames of a sequence were dropped such that the specific starting point of the sequence varies by $0 - 0.25$ seconds. This was achieved by selecting the number of frames to drop from the range $[0, 4]$ of integers such each value has equal chance. Thereafter the frames per second (fps) was reduced to half, from the original 25 fps, following the preprocessing approach taken by Ji et al. (2013) and Jhuang et al. (2007). Rather than discard every other frame, the sequence was divided into two, with the alternating framed allocated to the second sequence.

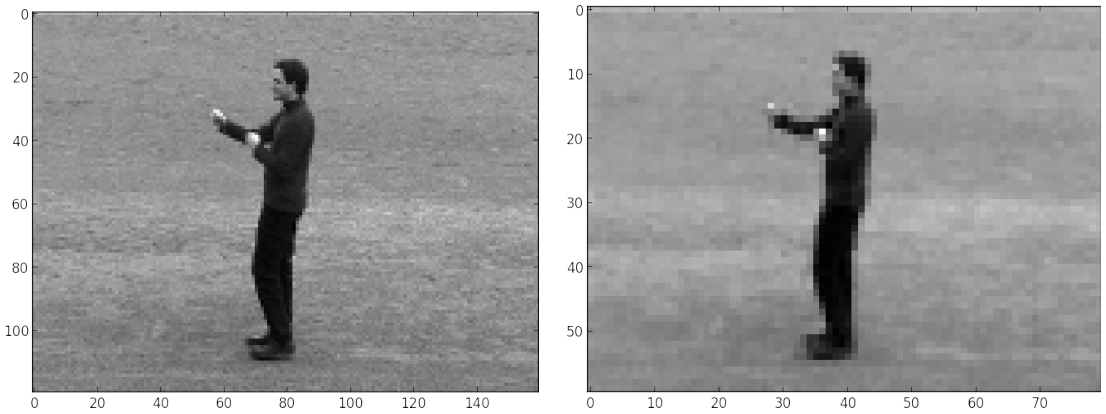
The subsequent augmentations inspired from recent large CNNs (Dieleman, 2014; Krizhevsky et al., 2012) were applied to individual 9-frame clips:

- Taking advantage of the natural orientation of the ground at the bottom and actions that are symmetrical around the vertical axis, each clip was randomly flipped about the vertical axis with a 50% chance.
- Small scale noise was introduced to each clip whereby it was scaled by a random value drawn from a uniform distribution with bounds $[0.98, 1.03]$.
- A small translation noise was also introduced whereby each clip was also translated between $[0, 3]$ pixels in the vertical and horizontal axes, independently.

The final input to the network is a 9-frame clip with a spatial resolution 80×60, where each pixel value is in the range $[-0.5, 0.5]$. To return a single classification for a video, vectors returned by the network for each 9-frame clip that compose a video are summed and the video is classified as the class corresponding to the maximum value.

9.3 Implementation Environment

The models were implemented in Python using the Theano library (Bergstra et al., 2010) to achieve an efficient implementation for the CNNs. The Theano library is a



(a) Original frame, Mean: 130.1 Maximum: 227.0 Minimum: 17.0

(b) Preprocessed frame, Mean: 0.0 Maximum: 0.3 Minimum: −0.4

Figure 9.2: Figure (a) illustrates a frame from the original data, Figure (b) depicts the same frame after having been preprocessed.

compiler for functional mathematical expressions which uses a symbolic representation that affords Theano the ability to perform optimizations upon the code and translates the Python code into C++ or CUDA depending on whether the code is implemented upon the CPU or GPU. By translating into C++ or CUDA, the code gains further speed ups. Additionally, Theano's symbolic nature allows it to perform symbolic differentiation for complex functions, which is especially necessary for large CNNs. See Listing 1 for an example of the symbolic definition of matrix multiplication in Theano.

Listing 1 Matrix multiplication in Theano.

```
1  import theano
2  import theano.tensor as T
3  import numpy as np
4
5  #Declare symbolic variables
6  X = T.matrix()
7  W = T.matrix()
8
9  #Compile functions in C++, check for optimizations
10 dot_fn = theano.function(inputs=[X,W], outputs=T.dot(W,X))
11
12 #Perform calculation given X_data and W_data
13 dot_fn(X_data, W_data)
```

When comparing the speed of computational times of traditional machine learning algorithms, Theano improves upon alternatives, including the widely used Numpy Python library, by $1.6-7.5\times$ when implemented upon the CPU (Bergstra et al., 2010). These performance gains are especially advantageous when considering they are achieved through

automated optimizations with no input required by the coder.

As mentioned in Chapter 4, to train a large CNN in a practical amount of time, it is necessary to implement the computation upon a GPU (Cireşan et al., 2012b; Krizhevsky et al., 2012). The architecture of GPUs have been specifically designed for matrix manipulation due to its prevalence in rendering graphical environments. Theano supports GPU implementation through translation of the Python code to CUDA. Using the same benchmark tests as the CPU test environment above, Theano improved the speed of computation by $6.5 - 44\times$ compared to alternative libraries (Bergstra et al., 2010). Upon a tutorial which implements a CNN similar to LeNet-5 (See Chapter 4) the use of a GPU increases the speed of computation by approximately ten-fold when compared to the same Theano code upon the CPU (LeNet tutorial, 2013).

While specific performance gains are dependent upon the algorithm as well as the hardware upon which the program is implemented, the above serves to demonstrate the benefit achieved by Theano's optimizations and a GPU implementation.

The facilitation of optimization and the symbolic computational nature of Theano affords rapid prototyping of different algorithms. In the specific context of CNNs, this benefit is especially necessary as the implementation of backpropagation algorithm is arduous and hinders the experimentation with different architectures when performed manually. Listing 2 demonstrates the implementation of logistic regression, an abbreviated version of the program for the CNNs used in this research is included in Appendix B.

Theano supports automated differentiation of NNs through an application of the chain rule. This requires gradient defined operations for individual functions from which the chain is constructed. When requesting the gradient for NN, Theano precomputes the chain rule, symbolically, which is then employed to provide specific values when given input. For second order learning rate adaptation techniques, the Hessian is approximated due to computational complexity making it's computation intractable. To impose the SDLM approximation would require a substantial rework of the basic operations of the Theano library. As a result, the SDLM and vSGD learning rate adaptation methods were not implemented.

Figure 9.3 provides a simplified overview to the 3D CNN training program, depicting important classes and their dependencies within the program, where yellow classes were partially or wholly built by the author for the work done in this research (See Appendix B for further details). Upon development, attention was paid to the modularity of layers to allow different architectures to be easily compiled. Specifically, the `3D_CNN` class combined instantiations of layer classes to create different architectures. Data extraction was performed separately to the compilation and training of the networks and

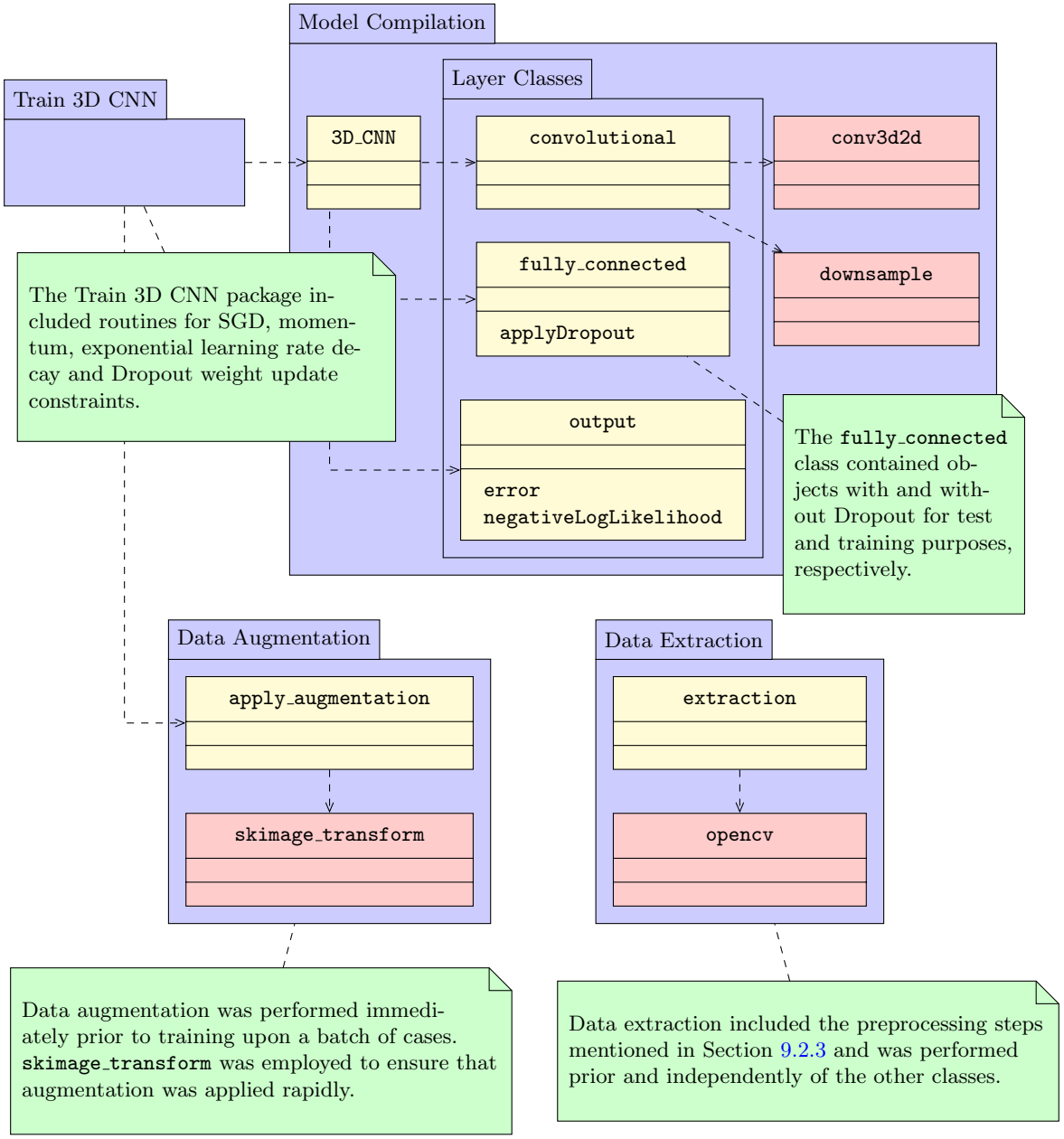


Figure 9.3: A simplified Unified Modeling Language (UML) package diagram of 3D CNN training program. Only important methods are included and object names are simplified to improve interpretability. Classes depicted in red are wholly externally developed modules (See Section 9.3.1 for further details.) and while Theano was implemented throughout the program, `conv3d2d` and `downsample` Theano modules are depicted for emphasis.

included the preprocessing steps mentioned in Section 9.2.3 before saving the extracted data. The OpenCV library (Bradski, 2000) was used to convert the video data to a Python interpretable data type. The Train 3D CNN module instantiated a network using a `3D_CNN` class, employing the `negativeLogLikelihood` method from the `output` class as a minimization goal for SGD. In addition to performing SGD, the Train 3D CNN module implemented momentum, exponential learning rate decay and the Dropout weight update constraints. The backpropagation of the error derivative was achieved automatically by Theano as described earlier. Upon training, the Data Augmentation module was used to augment the data immediately prior training upon a batch of cases. The `skimage_transform` module (van der Walt et al., 2014) was used to ensure that augmentation did not slow training substantially. Finally, upon testing (Not depicted in Figure 9.3), the same `3D_CNN` class was used, but with the trained weights, upon the extracted data from the Data Extraction module.

Listing 2 Logistic regression with Theano

```
#Symbolic variable declaration
X = T.vector()
y = T.scalar()

#Initialization of W vector
W = theano.shared(np.random.normal(loc=0.0, scale = 0.1, shape = n_in))

#Definition of functions
logistic_regression = 1/(1+T.exp(-T.dot(X,W))
mean_square_error = T.mean(T.sqr(y - logistic_regression))

#Definition of gradient for function
error_gradient = T.grad(cost = mean_square_error, wrt = W)

#Compile functions in C++, check for optimizations, including the
# weight update rule
training_fn = theano.function(inputs = [X, y], \
                outputs = logistic_regression, \
                givens = [(W, W - learning_rate * error_gradient)])

#Train for one epoch upon X_data and y_data
training_fn(X_data, y_data)
```

9.3.1 Hardware and Software Details

In this research each CNN was implemented upon a Tesla M2090 GPU and two Intel Xeon E5-2650 2.0GHz CPUs in a 64bit SUSE Linux Enterprise Server 11 SP3 environment provided by the University of Cape Town's ICTS High Performance Computing unit (`http://hpc.uct.ac.za`).

Theano along with supporting libraries was used to develop the network, additional specialised libraries were used to preprocess the video data and ensure timely augmentation of the data. The specific versions of the relevant software used is described below.

The CNN models were developed in Python 2.7.6 using Theano 0.6rc5 (Bergstra et al., 2010), Numpy 1.7.1 (Oliphant, 2007), Scipy 0.11 (Jones et al., 2014) and their respective dependencies. In order to implement the models upon the GPU, CUDA 6.0 (Nickolls et al., 2008) was used to translate the necessary code. Of note, the conv3d2d and downsample Theano modules was used to perform 3D convolutions and max pooling, respectively.

Data preprocessing was achieved with OpenCV 2.4.2 (Bradski, 2000) and fast augmentation of 9-frame clips was achieved using Scikit-Image 0.9 (van der Walt et al., 2014). Finally, all graphs were produced using the Matplotlib 1.1.1 (Hunter, 2007).

9.4 Architecture of Models

The approach to selecting an optimal architecture of a CNN is not well defined. A CNN's architecture needs to balance practical computation time and overfitting with the necessary capacity to sufficiently model features. Commonly, recommendations are made from testing different design decisions such as the use of more feature maps (Zeiler and Fergus, 2014) or how many layers a network should have (Glorot and Bengio, 2010; Jarrett et al., 2009). These recommendations are usually made within specific contexts and are not extensively tested. As a result, the exploration in the space of different models is still performed manually, governed by several design heuristics. This presents a shortcoming of CNNs in general, searching for an optimal architecture is combinatorially infeasible and impractical given the lengthy training times of CNNs.

The architecture employed by Ji et al. (2013) was used to inform the initial design of the architecture used in this research given its prior performance upon the KTH dataset (see Section 8.3 for an illustration of the architecture employed by Ji et al. (2013)). Three models were developed, in doing so the objective was not to find the single optimal architecture for the KTH which would have likely resulted in substantial overfitting without

the use of a holdout test set. Rather the models were used to better understand general design principals and produce models with improved independence of the variance component of the error when compared to models with the same architecture but different starting weights (Naftaly et al., 1997). By decreasing the structural similarities, the level of independence of the outputs of the models is expected to decrease (See Chapter 6).

The general structure of the first model, referred to as Net-A, consisted of a hard-wired layer that divides the input into five separate convolutional channels each of which have two convolutional and max-pooling layers. The five channels were then recombined in two succeeding fully-connected layers and finally employs a softmax output with six separate outputs, one per an action class. See Figure 9.4 for an illustration of the Net-A's architecture.

Following advice by Ji et al. (2013), the raw input is divided into five separate channels by means of a hardwired-layer, these five separate channels are defined by the features that they encode, namely: horizontal gradient, vertical gradient, horizontal optical flow, vertical optical flow and the original, grayscale, raw input. The gradient channels were calculated using the horizontal and vertical Scharr gradient operators with a kernel size of 3×3 (Scharr, 2000). Optical flow was calculated using the algorithm presented by Farnebäck (2003).

After splitting the data into the five channels, two convolutional and pooling layers were applied to each channel separately. The dimensions of the kernels and the number of feature maps learnt between the different channels were kept similar for simplicity.

In the channel for the unmodified input, the first convolutional layer, C1, trained 10 feature maps, each with a $9 \times 7 \times 5$ receptive field with respect to width, height and time. Thereafter a max-pooling layer, P2, with no trainable parameters, was applied to non-overlapping 3×3 spatial regions. The second convolutional layer, C2, consisted of 10 feature maps, each with a $7 \times 7 \times 5$ receptive field. The optical flow channels were modified slightly to account for the temporal dimension being diminished by one due the optical flow calculation. As a result, the receptive field in C2 has a size of $7 \times 7 \times 4$. Note that each neuron received input from receptive fields in the same position across all incoming feature maps. This layer was also followed by a max-pooling layer, P4, which downsampled across non-overlapping 3×3 spatial regions. In the horizontal and vertical gradient channels kernels of the same size were applied.

The output of the second max-pooling layer is flattened into a vector composed of 1,200 elements which is fully-connected to, F5, a layer of 300 neurons. The output of F5 is fully-connected to F6, a layer of 100 neurons which is subsequently connected to the

output layer, O7, that consists of six neurons, one per action class. The resulting number of parameters per a layer are detailed in Table 9.1.

The rectified linear unit was used as an activation function across the entire network, with the exception of the output layer which employed a softmax unit in order to be consistent with classification task. Weights were initialized following the recommendations of Glorot and Bengio (2010). Dropout, with a 50% chance of omission, was applied to the neurons in the two fully-connected layers to assist in regularising the network. Parameters in the Dropout layers were regularized as described in Chapter 6 with the squared L2 norm of the incoming weights constrained to a maximum value of 15. Momentum of 0.9 was applied across the whole network and, as detailed in Chapter 2, the negative log likelihood is minimized.

As mentioned in Section 9.3, second order learning rate adaptation techniques were not possible to implement. Subsequently, as exponential learning rate decay technique was implemented due to the good performance and simplicity when compared to first order learning rate adaptation techniques (See Chapter 5). An exponential learning rate decay was applied across both the convolutional layers as well as the Dropout layers, however it was initialized at different rates and decayed at different rates to allow a large learning rate to be applied to the Dropout layers, following the advice of Hinton et al. (2012). Across the convolutional layers the learning rate was initialized at 0.002 and exponentially decayed by 0.985 per an epoch such that the learning rate would anneal to ≈ 0.0001 over the course of training for 200 epochs. The learning rate for the Dropout layers was initialized at 0.3 and decayed by 0.98 per an epoch to anneal to ≈ 0.005. Weights were initialized by sampling each weight from a narrowly defined uniform distribution that is symmetric around 0, with the bounds according to Glorot and Bengio (2010). Finally, SGD employed a mini-batch size of 128 9-frame cubes.

In addition to Net-A, two other architectures were built, both used a similar architecture and maintained the 5 separate channels. Net-B was produced by removing the final fully-connected layer of Net-A, increasing the size of the receptive field in the spatial dimensions and reducing the number of neurons in the remaining fully-connected layer to curb the number of parameters outputted from the second pooling layer. Specifically, the first convolutional layer, C1, employed receptive fields with a size $18 \times 14 \times 5$ and the following convolutional layer, C3, had a receptive fields with a dimension of $10 \times 6 \times 5$. In the remaining fully-connected layer the number of neurons was reduced to 200. The intention was to reduce the network's capacity to overfit as there would be fewer free parameters. Furthermore, previous work by Zeiler and Fergus (2014) suggested that removing one of two fully-connected layers would not have a substantial negative effect on a networks performance (Zeiler and Fergus, 2014). The afore mentioned amendments

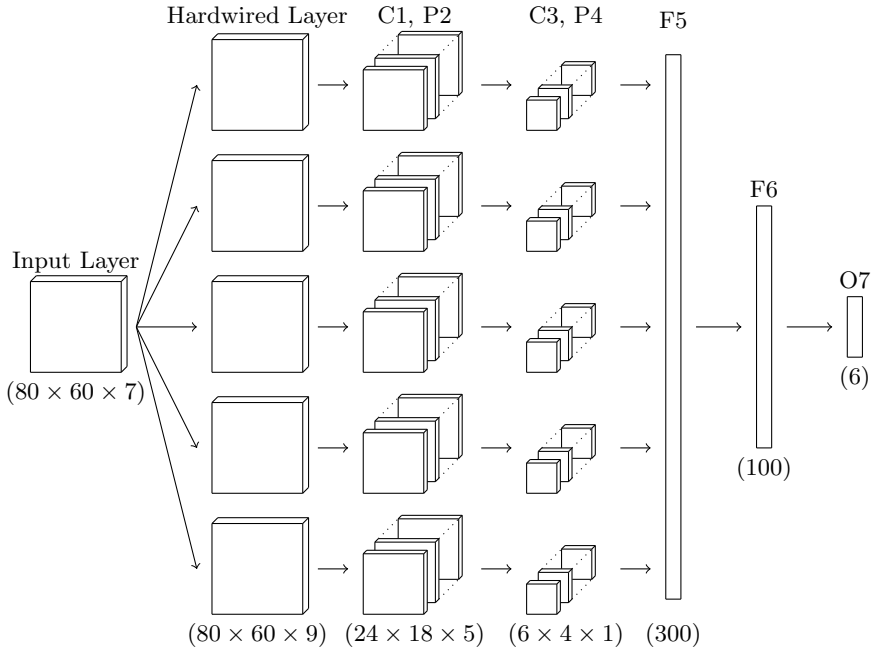


Figure 9.4: Illustration of the output of each layer of Net-A. Cx, Px, Fx and Ox indicate convolutional, downsampling, fully-connected and output layers, respectively and the location in the hierarchy. Beneath each layer, the dimensions of an outputted feature map are detailed. Note that, in different channels the dimensions are slightly different, specifically, in the optical flow channels, the temporal dimension is reduced by one in C1, P2, C3, P4 and, in the gradient channel, the spatial dimensions inputted from the hardwired layer is reduced by two in both the vertical and horizontal dimensions.

resulted in 157,050 fewer parameters compared to Net-A, see Table 9.1 for further details. The remaining details, including learning rate decay, activation unit choice were the same as Net-A.

Net-C had a similar structure to Net-B with two exceptions. The size of the receptive field in the spatial dimensions were further increased and the number of feature maps learnt in the second convolutional layer, C3, was increased. The receptive fields in C1 and C3 were increased to $27 \times 22 \times 5$ and $13 \times 8 \times 5$, respectively. Due to the larger receptive fields, the number of inputs to the fully-connected layer, F5 was reduced, decreasing the number of free parameters between P4 and F5. The number of receptive fields in C3 was doubled, to 20 in each channel. The intention behind this design was to allow larger and more features to be learnt. The architecture resulted in many more free parameters when compared to Net-A or Net-B, the majority of which are situated in the convolutional layers. It was expected that the additional parameters would not result in substantial overfitting because of the regularization imposed by the parameter sharing nature of the convolutional architecture. As with Net-B, the remaining details of Net-C were the same as Net-A.

Layer	Net-A
C1	15,800
P2	-
C3	112,750
P4	-
F5	360,300
F6	30,100
O7	606
Total	519,556

Layer	Net-B	Net-C
C1	63,050	148,550
P2	-	-
C3	138,050	478,500
P4	-	-
F5	160,200	80,200
O6	1,206	1,206
Total	362,506	708,456

Table 9.1: Number of trainable parameters in each network by layer. Cw, Px, Fy and Oz indicate a convolutional, max-pooling, fully-connected and output layer, respectively. Note that the pooling layers did not include trainable parameters. Net-A had an extra fully-connected layer.

9.5 Results and Discussion

This section presents observations made during training and reports results achieved upon the KTH dataset. In each case, a network was trained for 200 epochs, where an epoch consisted of 3072 9-frame training clips, selected randomly without replacement from the training set. Each network took approximately 44 hours to train upon the GPU. Upon the CPU, networks took approximately 9.6× longer to train.

Throughout training, the error upon the training set was observed to decrease, initially rapidly and then more gradually. The test error over the course of training did not increase, and converged in the latter epochs, as illustrated in Figure 9.5. After approximately 30 epochs the training error was substantially smaller than the test error furthermore the training error continued to decrease throughout the 200 epochs. The training and test error of Net-B decreased more rapidly than the other networks, although the test error of Net-A converged to a lowest final value, when compared to Net-B and Net-C. It was also observed the final training error for each of the networks is approximately the same.

That the training error was substantially smaller than the test error and continued to decrease is an indication that the models were overfitting although the networks' test error was not observed to increase. Had the test error increased during training, it would have indicated that the networks were fitting specific features in the training cases that did not generalize to the test set. In this regard, the regularization techniques implemented appear sufficient at preventing that extent of overfitting.

The smaller number of parameters in Net-B result in faster convergence as the weight space is smaller and hence quicker to search. However the smaller number of parameters also reduced the network's capacity which would result in a larger bias error component in the error.

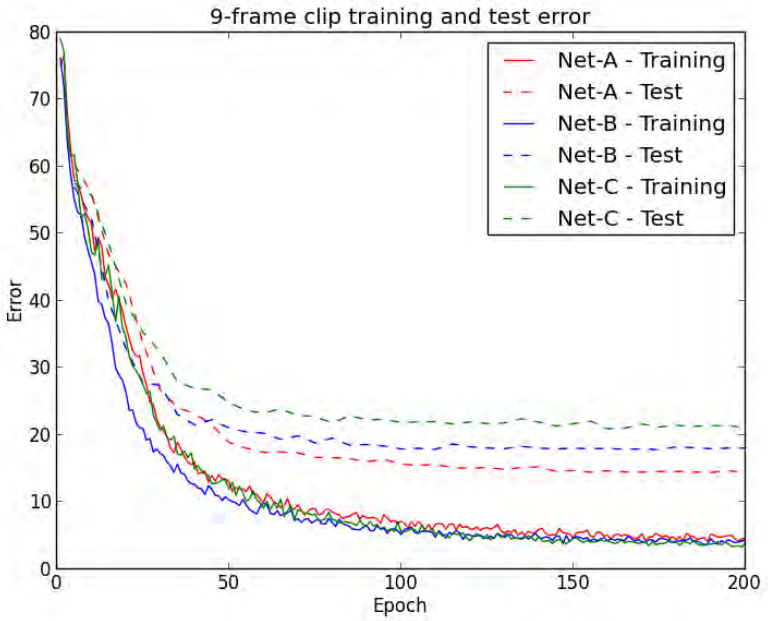


Figure 9.5: Average training and test error of Net-A, trained upon all scenarios simultaneously.

The test error achieved upon the videos by each network across each of the folds is detailed in Table 9.2. As examined by Gao et al. (2010), the error rate across different folds varied substantially with the largest difference in result of 9.3% occurring upon Net-B between fold-1 and fold-5. As such, the assignment of subjects to the training and test sets can have a large effect upon the test error experienced by the network. To mitigate this effect, a 5-fold cross-validation, as suggested by Gao et al. (2010) was reported. Of the three architectures Net-A achieved the lowest error rate of 10.4%, which is lower than Net-B and C, which achieved an error rate of 13.3% and 16.0%, respectively.

As mentioned earlier the higher error rate of Net-B when compared to Net-A could arise due to the smaller number of parameters not allowing the model to establish as much capability to distinguish the classes. Equally, the afore mentioned shortcoming could arise from Net-A's additional fully-connected layer allowing the aggregation of neurons from layer F5 to make improved decision boundaries between classes, contrary to the experience of Zeiler and Fergus (2014). Conversely, the relatively worse performance by Net-C could arise from too many parameters allowing the network to overfit upon the training data.

Two ensemble models are constructed by consecutively aggregating the outputs of the networks in order of the error rate they achieved. The first ensemble, consisting of Net-A and B, improved on the result of Net-A, achieving an error rate of 10.2%. The second

Model	Fold-1	Fold-2	Fold-3	Fold-4	Fold-5	Average
Net-A	13.95	13.95	**8.37**	**6.48**	9.25	10.40
Net-B	17.21	15.35	12.56	13.43	**7.87**	13.28
Net-C	18.14	20.47	14.42	12.96	13.89	15.98
Average	16.43	16.59	11.78	10.96	10.34	13.22
Ensemble {A, B}	**11.63**	**12.56**	9.30	9.72	**7.87**	**10.22**
Ensemble {A, B, C}	12.09	14.88	10.23	10.19	8.33	11.15

Table 9.2: Test error rates achieved upon the videos, by fold. Bolded figures are the lowest error rate for the particular fold. Each fold is a random division of the 25 subjects in the KTH dataset into a test set of 9 subjects and a training set of 16 subjects.

ensemble, consisting of Net-A, B and C achieved an error rate of 11.15%. As examined in Chapter 6, the ensemble error rate when compared to the average error of the individual models will be reduced as the variance component of the error will decrease depending upon the independence of the models. This is observed when comparing the average error of 13.2% with the error rate of 11.2% of Ensemble {A, B, C}. The addition of Net-C to Ensemble {A, B} degraded the results. This suggests that the variance component of the error increased, or the bias component of the error's increase was not offset by the change in the variance component of the error. The result achieved by Ensemble {A, B} did not substantially improve upon Net-A's result, which might from the increase in the bias component from including Net-B in the ensemble was not strongly offset from the decrease in the variance component of the error. Note that Ensemble {A, B} did perform more consistently across the different folds suggesting that the variance component of the error of the Ensemble was improved.

Table 9.3 is the confusion matrix for Ensemble {A, B} for average classification of videos across the five folds. Upon the stationary actions (boxing, clapping and waving), the model performs well, with some confusion upon the clapping class. Upon the moving actions (jogging, running and walking), there was large amount confusion between running and jogging and between jogging and walking. Considering the similarity between running and jogging and jogging and walking, confusion between these actions appears reasonable. For example, what some people might consider running, others might consider jogging, as such the speed at which a person runs or jogs will vary between different people. Finally, the ensemble did not confuse any actions between the stationary and the moving actions which, given the dissimilarity between the two types of actions, was expected.

Table 9.4 details the error rate by the four different scenarios upon the KTH dataset. The different scenarios had substantial impact upon test error. Specifically, scenario $s2$, which included scale variance, and $s3$, which had the subjects wear different and often shape distorting clothes, performed substantially worse than scenarios $s1$ and $s4$,

	Actual					
Predicted	Boxing	Clapping	Waving	Jogging	Running	Walking
Boxing	**97.77**	3.37	0.56	-	-	-
Clapping	-	**93.26**	1.67	-	-	-
Waving	2.23	3.37	**97.78**	-	-	-
Jogging	-	-	-	**78.33**	25.00	1.11
Running	-	-	-	7.78	**72.78**	-
Walking	-	-	-	13.89	2.22	**98.89**

Table 9.3: Confusion matrix for Ensemble {A, B}, bolded figures are the accuracy of the model upon that particular action

Test Scenario				
$s1$	$s2$	$s3$	$s4$	All
6.30%	17.04%	12.34%	5.21%	10.22%

Table 9.4: Ensemble {A, B} error according to the different scenarios.

which did not include substantial variance but consisted of homogeneous backgrounds in an outdoors and indoors location, respectively. This result is not unexpected as the increased variance introduced in scenarios $s2$ and $s3$ would naturally degrade performance. Scale invariance incorporated through max-pooling and the training of multiple feature maps is not unbounded, it may be that this scale invariance is not sufficient to capture the fluctuations in the scale introduced in scenario $s2$.

The result achieved by Ensemble {A, B} of 10.2% is comparable to the result achieved by Ji et al. (2013) of 9.8% (See Table 9.5). However performance by the two models across the different actions differed. Specifically, Ensemble {A, B} provided an improved performance upon the boxing action (98% vs 90%), but performed comparatively worse upon the jogging (78% vs 84%) and running (73% vs 79%) classes. In addition to Ji et al. (2013), the results of other 3D CNN and similarly structured models are included in Table 9.5. Baccouche et al. (2011) produced and evaluated a 3D CNN using a five-fold cross-validation error rate. Notably the 3D CNN presented by Baccouche et al. (2011) used a smaller input with a dimension of $34 \times 54 \times 9$, having extracted a person-centred bounding box as a preprocessing step. Furthermore, two methods of aggregating the outputs of the 9-frame sequences that made up each video. The first, employed a simple vote of the outputs referred to as 'Voted' in Table 9.5. The second used a method explicitly designed sequence classification method to combine the outputs referred to as a Long Short Term Memory (LSTM) RNN. The addition of which can be seen to improve the result of the network. The models presented by Jhuang et al. (2007) and Le et al. (2011) used a hierarchical, convolutional structure akin to a 3D CNN, however employed different methods of adjusting parameters within the network. Note that Le et al. (2011) only employed a single fold, as opposed to the five folds, subsequently the results are not directly comparable.

	Accuracy						
Model	Box	Clap	Wave	Jog	Run	Walk	Average
Ensemble {A,B}	**98**	**94**	**97**	**78**	**73**	**99**	**89.8**
Ji et al. (2013)	90	94	97	84	79	97	90.2
Voted Baccouche et al. (2011)	-	-	-	-	-	-	91.0
LSTM Baccouche et al. (2011)	-	-	-	-	-	-	94.4
*Le et al. (2011)	-	-	-	-	-	-	93.9
Jhuang et al. (2007)	92	98	92	85	87	96	91.7
Gao et al. (2010)	-	-	-	-	-	-	95.0
*Wang et al. (2009)	-	-	-	-	-	-	92.1

Table 9.5: 5-fold cross-validation performance of different models upon the KTH dataset. * Test error reported for only one fold.

Furthermore the error rate of several alternative methods that extract features through more traditional video processing methods and apply a classifier to the extracted features (Gao et al., 2010; Wang et al., 2009). Note that the result reported by Wang et al. (2009) only reported an error rate of a single fold.

9.6 Conclusion

The KTH dataset provides a diverse set of challenges for a video classification model, facilitating the testing of several different aspects of a model with a controlled dataset. For example, the division of the data into four different scenarios according to different variances allows models to test their strength in different situations. In the context of the 3D CNN presented here, the KTH dataset was used due to it's wide recognition and as it did not require subject localization, which is a research problem in and of itself. For example other video datasets, such as the TRECVID dataset, require that each subject is located within the frame prior to classification.

Due to the computational requirements of training a 3D CNN, the implementation environment is important to consider such that the model trains within a practical amount of time. Without implementation upon the GPU each model would have taken over two weeks to train, making the training of an ensemble and 5-fold cross-validation error computation impractical. The development of different architectures and components (such as activation functions, momentum and Dropout) requires an environment that is flexible and provides sufficient performance to perform the calculations timely. In this regard, Theano provides a good environment by allowing the user to focus upon the development of the 3D CNN in the high-level Python programming language, while automating the optimizations in C++ and CUDA.

Three different architectures of 3D CNNs were designed in order to explore broad design principals and facilitate the construction of an ensemble network. Inspired by the architecture of the 3D CNN built by Ji et al. (2013), the different models included additional feature maps, large receptive fields and varying amount of neurons in the fully-connected layers which resulted in an almost 2-fold difference in the number of parameters between the smallest and largest model. The three different models achieved different error rate where Net-A provided the single best error rate of 10.4%. Building ensemble models from the three networks did not substantially improve average performance of the model however did improve the consistency of the results suggesting an improvement in variance component of the error. The best performance of 10.2% realised by Ensemble {A, B}.

Assessing Ensemble {A, B} across the different action classes, the model performs as expected, accurately distinguishing between stationary and moving subjects but confusing between the walking and jogging and jogging and running actions. Across the different scenarios, the model also functions as expected, performing considerably worse upon the scenarios that contain more variance.

Ensemble {A, B}'s performance is comparable to other 3D CNN models upon the KTH dataset. Relative to more traditional feature extraction and classification methods, the model presented here performs comparatively worse. This is likely in part due to overfitting.

Chapter 10

Conclusion

10.1 Conclusion

Human action recognition has practical applications including security surveillance, patient monitoring and self-driving vehicles. 3D CNNs have achieved competitive performance upon human action recognition datasets (Ji et al., 2013; Baccouche et al., 2011), however recent developments upon 2D CNNs have not been applied in to 3D CNNs. The purpose of this research was to build a 3D CNN that emulated recently developed components that have been attributed with improving the accuracy of successful 2D CNNs.

Originally inspired from the visual cortex of a cat, CNNs are purposefully designed for image recognition and employ a hierarchical structure to develop successively more complex feature detectors while exploiting the commonality of low level features and local region dependencies in image data (See Chapter 2). Constraining feature detectors to only receive local inputs and then convolving each feature detector over the global input introduces scale and translation invariances and reduces the number of trainable parameters while regularising them. Downsampling layers are employed to help reduce the size of input dimension more rapidly while maintaining the relative position of features as well as introducing further scale invariance. Whereas fitting a fully-connected NN to raw pixel data would result in a very large number of trainable parameters which would make training intractable or could result in substantial overfitting given the relatively small size of image datasets, CNNs provide a well regularized and more tractable approach.

2D CNNs have recently demonstrated prowess at classifying images, achieving superior results upon numerous image datasets including the MNIST, CIFAR-10, SVHN, NORB

and ImageNet datasets (See Chapter 4). In the case of the ImageNet dataset, 2D CNNs substantially improved upon the next best alternative which employed a more typical image classification technique of extracting hand-built discriminative features and applying a classifier upon them (Krizhevsky et al., 2012). In Chapter 7, upon the Galaxy Zoo dataset, a 2D CNN model is demonstrated to provide comparatively good performance at identifying morphological features in images of galaxies. In this regard CNNs are advantaged over typical approaches as they do not require expertly informed feature extraction techniques, rather they learn their own feature extractor in conjunction with the classifier. As a result, CNNs are more transferable between different image recognition problems.

An informed local learning rate could improve the speed of convergence to a minimum and prevent divergence upon non-linear gradient descent (Le Cun et al., 2012). Several methods that employ first order and second order error derivative information to inform local learning rate adjustments are discussed in Chapter 5. While first order methods only imposed additional memory requirements and a small computational burden, they have not been demonstrated to outperform a simple exponentially decaying learning rate (Schaul et al., 2013; Duchi et al., 2011). Second order methods, under the assumption that the error surface is locally quadratic, employ the inverse Hessian matrix to inform the learning rate optimally. Due to computational complexity, approximations to the inverse Hessian are employed, such as the SDLM method proposed by Le Cun et al. (1998). Schaul et al. (2013) improved upon the SDLM technique with the vSGD method, which was was demonstrated to provide consistently accurate results without the need to predefine hyperparameters.

The recent successes of 2D CNNs have been facilitated by an improvement in hardware, allowing for CNNs with more layers and additional feature maps and neurons within layers and ensembles of networks to be trained upon larger, commonly augmented, datasets. Furthermore, new methods to assist in regularization have been developed which are increasingly important due to the larger number of parameters found in recent networks that would have otherwise resulted in substantial overfitting. Chapter 6 examines the Dropout and DropConnect methods from the perspective of ensemble models that attempt to reduce a model's error rate by reducing the variance error component of the bias-variance error decomposition. While in the context of small image classification the advantage of these methods over a regular ensemble is not apparent (Wan et al., 2013), in large image classification, regular ensemble models are often infeasible, subsequently, methods such as Dropout can improve performance without imposing a large computational burden (Krizhevsky et al., 2012).

Whereas large 2D CNNs employing the Dropout or DropConnect technique are increasingly more common, these trends have not been observed in 3D CNNs. However, as examined in Chapter 8 the extension of 2D CNNs to 3D CNNs is simple and the rationale behind the design choices still apply to video due to the similarity between video and image data.

Three 3D CNNs with different architectures were built, to provide insight into different design influences and enable the construction of an ensemble. The general structure of the architecture was informed by the model built by Ji et al. (2013), thereafter Net-A featured an additional fully-connected layer when compared to the other two. Net-B had the smallest number of parameters by reducing the number of outputs returned by the convolutional layers and the number neurons in the hidden layer and, finally Net-C had additional feature maps and a larger receptive field size in the convolutional layers. Each network also included several more recently developed components in accordance with the purpose of the research. This included the use of Dropout upon the fully-connected layers as a means of an efficient ensemble and to improve regularization. Furthermore the architectures included additional feature maps in the convolutional layers when compared to Ji et al. (2013) and to speed convergence the rectified linear unit was used as an activation function. Unfortunately, in this work, the implementation environment prevented the use of approximations to the inverse Hessian, subsequently an exponentially decreasing global learning rate was used in conjunction with momentum. To be comparable to other 3D CNNs and action recognition models, the 3D CNNs were implemented upon the widely studied KTH human action recognition dataset.

In order to train the 3D CNNs in a reasonable amount of time, it was necessary to ensure an efficient implementation. In this regard, Theano, a well optimized, mathematical Python library was used to build the models. Theano also facilitated implementation upon a GPU, which afforded a 9.6× decrease in computation time when compared to an implementation upon a single core CPU. Each 3D CNN model took approximately 44 hours to train.

The different network architectures appeared to influence the final test error achieved where Net-A, B and C achieved an error rate of 10.4%, 13.3% and 16.0%, respectively. The smaller number of parameters in Net-B when compared to the other nets appeared to increase the speed of training. When comparing the result of Net-B and Net-C, the increased number of parameters in Net-C at each layer appeared to have been detrimental to the final test error, suggesting that Net-C overfitted more readily than Net-A and B.

Two ensemble models were built using aggregated outputs from the three 3D CNNs. As expected the ensemble models improved upon the average error rates of the three models, furthermore Ensemble {A, B} produced a similar result to Net-A. In addition

Ensemble {A, B} produced more consistent results across the different folds, which given the reduction of the variance component of the error realised by ensemble models, was to be expected.

When assessing Ensemble {A, B} across the different actions, the model performed as expected. While clearly distinguishing between the moving and stationary tasks, the model confused moving tasks with each other. Specifically, running and jogging and jogging and walking were the mostly commonly confused, which appears reasonable given the similarity between tasks and the subjectivity which may have been introduced by the person performing the action.

While the inclusion of the recently developed elements in 3D CNN model did not lead to an accuracy comparable to state-of-the-art results upon the KTH dataset, the results are comparable to other 3D CNN models applied to the KTH dataset. Specifically, the 5-fold cross-validation error rate of 9.8% achieved by Ji et al. (2013) and 9.0% achieved by Baccouche et al. (2011) is similar to the final error rate of 10.4% of Net-A and 10.2% of Ensemble {A, B}. However, the error rate of 5.0% achieved by Gao et al. (2010) is substantially lower than the error rates achieved by the models presented here. The 3D CNN model presented in this work and those built by Ji et al. (2013); Baccouche et al. (2011) have numerous differences between components and architectures suggesting that there are multiple approaches to building a 3D CNN to achieve a similarly competitive performance, whereas the different architectures of Net-A, B and C appeared to influence the accuracy each network achieved. This suggests, that while different 3D CNN models can achieve similar performance, the interaction between the architecture and the components of the model are necessary to take into account. While it would be impractical to search through all possible combinations of different design components and architectures the 3D CNN presented here provides a competitive combination.

10.2 Limitations

The training and testing of models could not be repeated to provide a large enough set of results to perform statistical testing, as such the differences in accuracy between the different architectures and ensembles cannot be stated with any degree of significance. This shortcoming is common amongst CNN models due to the lengthy training time of each model, which makes attaining a large enough sample of results impractical. For example, work done by Ji et al. (2013) and Baccouche et al. (2011) do not provide sufficient repetitions to be able to statistically distinguish results.

The ability to generalise to different image or video datasets is one of the characteristics that has motivated the development of CNNs. The focus of this research was upon the KTH datasets, and as such the 3D CNN models presented in this work were not applied to other datasets in order to test their applicability in a more general context. To produce results comparable to previous research upon 3D CNNs, it was necessary to focus upon a common dataset. In addition the KTH dataset did not require localization of the subject prior to classification. Localization is in itself a research problem which would have had an impact upon results and was beyond the scope of this work. Practical time constraints prevented the extension of the model to other datasets due to the long training time of individual networks as well relatively low-level of programming that the model was constructed at. Further work could be performed upon extending the 3D CNN model presented here to further human action recognition and video datasets.

The impact that different folds of the KTH dataset have upon the outcome suggest that there is substantial variance between how different subjects portray the actions, suggesting that in some cases the subjects in the training data were not representative of the subjects in the test data. This alludes to the smallness of the KTH data from the perspective of the subjects and that results of the models could have been improved by training upon additional subjects. However to remain comparable to other 3D CNNs a 5-fold cross-validation error was reported using a test set of 9 subjects, but the accuracy reported could improve if a larger proportion of the data was used to train upon. The suggestion by Gao et al. (2010) to use a leave-one-out cross-validation methodology, would maximise the training set but have required the training of more models.

vSGD can improve the final accuracy a network achieved when compared to an uninformed exponential decay learning rate Schaul et al. (2013), however the benefit of automated, symbolic differentiation provided by Theano, meant that implementation of the vSGD was not possible. Implementing the CNN models at a lower level programming environment would have allowed the implementation of the vSGD, although it would have increased the development time of the model.

10.3 Further Research

The large number of design choices that are required when building a CNN allow for numerous possible modifications to the model. Two particular areas of research that have received increased attention recently are training upon unlabelled data, and the dissection of features learnt to inform an improved specific network design.

Unsupervised learning involves training upon unlabelled data, it can be performed in isolation to other training, when no labelled data is available, or in conjunction with labelled data where it is referred to a semi-supervised learning. It is particularly helpful when the labelled training data are small. Semi-supervised learning is commonly used in the case of pretraining the parameters and has been found to improve the accuracy of the network as well as the network's ability to generalize (Erhan et al., 2010). Hinton and Salakhutdinov (2006) introduced the method of stacking autoencoders which reinvigorated research into unsupervised learning as it facilitated the training of deep NNs. Stacked autoencoders adjust weights in an unsupervised manner by attempting to minimise an error function tied to the recreation of the input on a layer by layer basis. In the context of 3D CNNs, Le et al. (2011) used Independent Subspace Analysis (Hyvärinen and Hoyer, 2000) to pretrain a 3D convolutional network structure and achieved an error rate of 6.1% upon the KTH dataset as well as improving upon other video datasets.

The size of the receptive field can influence the type of features that are learnt and can influence the outcome of the model. Zeiler and Fergus (2014) demonstrated that through careful dissection of the feature maps that have been learnt, adjustments can be made to their size to improve the networks overall outcome. This approach provides an alternative to a naïve exploration of feature size and provides further insight into features of the data. Extending upon this approach to video data could provide helpful insight into the effect of the temporal information when identifying actions.

Appendix A

Random Forests

A.1 Decision Trees and Random Forests

A.1.1 Introduction

The success of Random Forests is predicated by bagging. Random forests are an ensemble of decision trees and were introduced by Breiman (2001). Despite their simplicity they have been demonstrated to successfully perform regression and classification tasks with competitive performance (Caruana and Niculescu-Mizil, 2006).

In the following section decision tree theory is very briefly introduced, thereafter the Random Forest model is discussed. Much of the summarised work presented here is derived from Hastie et al. (2009), which provides a more complete explanation.

A.1.2 Decision Trees

Classification and Regression Trees (CART), otherwise referred to as Decision Trees, were introduced by Breiman et al. (1984). They are characterised by their simple and interpretable implementation and high variance, low bias nature (Hastie et al., 2009). A decision tree is grown from supervised data by recursively applying a binary decision to minimize an error function until a stopping criteria is met. By adjusting the error function decision trees can easily be applied to both regression and classification techniques, for the purposes of this introduction only the regression case will be considered. Given a labelled set of data $((y_1, \mathbf{x}_1), (y_2, \mathbf{x}_2), \ldots, (y_n, \mathbf{x}_n))$ where $y_i \in R$ and $\mathbf{x} \in R^d$, a decision tree will divide the space R^d up using successive binary splits into M regions,

such that:

$$f(\mathbf{x}) = \sum_{m=1}^{M} c_m I(\mathbf{x} \in R)$$

where I is the indicator function. The model needs to resolve which variable to split upon and at what value as well as when to stop splitting the R^d space and the value of c_m. The first two decisions are typically done in a greedy minimization of an error function. Usually the mean square error function is used in the case of regression, which simplifies the best estimate of c_m to be the average of the y_i values associated with the x_i values in region R_m. The splitting point s and variable j is then determined by:

$$\arg\min_{(j,s)} \left(\sum_{x_i \in R_r(j,s)} (y_i - \hat{c}_r)^2 + \sum_{x_i \in R_{r+1}(j,s)} (y_i - \hat{c}_{r+1})^2 \right)$$

where $\hat{c}_r$ is the average of the y_i values in $R_r(j,s)$. This splitting process is repeated until a stopping criteria is met. Commonly a minimum node size is used as a stopping criteria, for example, no further splits are performed when the node only contains 5 or fewer cases, which is typical for regression trees.

A.1.3 Random Forests

As mentioned above Decision tree models have high variance and low bias. Small changes in the input variables can result in substantially different models. This positions them as a good model to apply bagging to. Random Forests are the implementation of this idea. One important further addition in Random Forests is the random selection of a subset of variables to decrease the trees' correlation. Algorithm 3 describes the method of growing a Random Forest.

Algorithm 3 Random Forests (Hastie et al., 2009)

for b=1 **do** B
  Draw bootstrap sample b from training data
  repeat ▷ Build Decision tree from a random subset of the input variables
    1. Randomly draw subset M from complete set of variables p
    2. Split at s for variable $j \in M$ to minimize error function E
  until Terminal node conditions met
end for
  return Ensemble of Trees: $\{T_b\}_{b=1}^{B}$

Prediction with Random Forests involves aggregating the votes by the trees and for new data point $\mathbf{x}$ is made as follows:

$$f(\mathbf{x}) = \frac{1}{B} \sum_{b=1}^{B} T_b(\mathbf{x})$$

The number of variables, l, to be randomly drawn from the full set of variables is a hyperparameter, that would be predetemined. Consider the average variance of B variables that have a positive pairwise correlation ρ and identical variance σ^2:

$$\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$$

As B increases the right hand term becomes increasingly insignificant. As l decreases then the correlation ρ will decrease, however the variance of each variable σ will increase. For regression the suggested value for l is $\lfloor p/3 \rfloor$. Because of the imposition of bootstrapping and random variable selection the bias error component will usually be larger than that of an unrestrained decision tree, however the improvement of in the variance component of the error offsets the bias' component in the case of Random Forests.

Random forests can be trained without an explicitly separate validation dataset through the reuse of Out Of Bag (OOB) samples. OOB samples refer to bootstrapped samples that were not used to grow that particular tree. These samples can then be used to determine the validation error by having them predicted by the set of trees that were not grown on them and then the Random Forest's prediction is estimated by the their aggregate prediction. The estimated error is a good estimator of the test error (Hastie et al., 2009). Furthermore increasing the number of trees in a forest will not overfit the data, although the individual trees can overfit the data if fully grown. Some benefit can be realised by pruning the trees Segal (2004), however Hastie et al. (2009) recommend fully growing trees to avoid an additional tuning parameter and because the benefit of pruning is not large.

Appendix B

3D CNN Computer Code

B.1 Introduction

The Python code for the model described in Chapter 9 is included in this Appendix. Note that the coding style and structures were informed by LeNet tutorial (2013). Furthermore in order to increase the speed of augmentations in Section B.7 the `fast_warp` method by Dieleman (2014) was used. For brevity and due to its relative importance the data extraction methods are excluded. The complete working program is available from the author on request and included on the CD distributed with the hard copy of the dissertation. To implement the program refer to the `Readme.txt` file included with the complete working program.

B.2 3D CNN Class

This Section contains the code for Net-A as described in Section 9.4. The class defines the over architecture of the network, but specific dimensions of feature maps downsampling receptive fields and fully-connected layers are defined when the class is instantiated in the training method.

```
1 import numpy as np
2 import theano.tensor as T
3 import theano, time, random, os, sys, csv, cPickle
4 from layer_classes.util import     load_data, save_data, load_data_gpu,
5                                 swap_gpu_dataOF, swap_aug_gpu_dataOF,\
6                                 soft_sign, mod_tanh, ReLU
7 from layer_classes.MLP import FullyConnectedLayer, DropoutFullyConnectedLayer
```

```
from layer_classes.ConvolutionalClasses import Convolutional3DPoolingLayer,\
                                               Convolutional3DLayer,\
                                               GradXScharr, GradYScharr
from layer_classes.OutputClasses import SoftmaxLayer
from layer_classes.data_augmentation import cube_all_seq

class ConvolutionalNeuralNet(object):
    def __init__(self, input, OF_input, input_shapes, grad_input_shapes,
                       OF_input_shapes, filter_shapes, opt_flow_filter_shapes,
                       pooling_sizes, num_of_output_classes, dropout_chance,
                       minibatch_size, activation_fn, all_W, all_b,
                       uni_seqID = None):

        ## HARDWIRED LAYER
        # Grad x, y channels
        self.ChannelGradXHardLayer =  GradXScharr(input, input_shapes[0])
        self.ChannelGradYHardLayer =  GradYScharr(input, input_shapes[0])
        # NOTE: h, w will lose 2 dim's, however, due ignore_border=True,
        #       after the first subsampling layer, the dimensions of the
        #        grad channels will be aligned with the asis layer.

        ## LAYER 1
        self.ChannelGradXLayer1 = Convolutional3DPoolingLayer( \
                                    input=self.ChannelGradXHardLayer.output,
                                    input_shape = grad_input_shapes[0],
                                    filter_shape = filter_shapes[0],
                                    next_filter_shape = filter_shapes[1],
                                    pooling_size = pooling_sizes[0],
                                    activation_fn = activation_fn,
                                    W = all_W[0], b = all_b[0])
        self.ChannelGradYLayer1 = Convolutional3DPoolingLayer( \
                                    input=self.ChannelGradYHardLayer.output,
                                    input_shape = grad_input_shapes[0],
                                    filter_shape = filter_shapes[0],
                                    next_filter_shape = filter_shapes[1],
                                    pooling_size = pooling_sizes[0],
                                    activation_fn = activation_fn,
                                    W = all_W[1], b = all_b[1])
        self.ChannelAsisLayer1 = Convolutional3DPoolingLayer( \
                                    input=input,
                                    input_shape = input_shapes[0],
                                    filter_shape = filter_shapes[0],
                                    next_filter_shape = filter_shapes[1],
                                    pooling_size = pooling_sizes[0],
```

```
52                                  activation_fn = activation_fn,
53                                  W = all_W[2], b = all_b[2])
54          self.ChannelOptXLayer1 = Convolutional3DPoolingLayer( \
55                                  input = OF_input[0],
56                                  input_shape = OF_input_shapes[0],
57                                  filter_shape = opt_flow_filter_shapes[0],
58                                  next_filter_shape =  \
59                                      opt_flow_filter_shapes[1],
60                                  pooling_size = pooling_sizes[0],
61                                  activation_fn = activation_fn,
62                                  W = all_W[3], b = all_b[3])
63          self.ChannelOptYLayer1 = Convolutional3DPoolingLayer( \
64                                  input = OF_input[1],
65                                  input_shape = OF_input_shapes[0],
66                                  filter_shape = opt_flow_filter_shapes[0],
67                                  next_filter_shape = \
68                                       opt_flow_filter_shapes[1],
69                                  pooling_size = pooling_sizes[0],
70                                  activation_fn = activation_fn,
71                                  W = all_W[4], b = all_b[4])
72
73          ## LAYER 2
74          self.ChannelGradXLayer2 = Convolutional3DPoolingLayer( \
75                                  input=self.ChannelGradXLayer1.output,
76                                  input_shape = grad_input_shapes[1],
77                                  filter_shape = filter_shapes[1],
78                                  next_filter_shape = filter_shapes[2],
79                                  pooling_size = pooling_sizes[1],
80                                  activation_fn = activation_fn,
81                                  W = all_W[5], b = all_b[5])
82          self.ChannelGradYLayer2 = Convolutional3DPoolingLayer( \
83                                  input=self.ChannelGradYLayer1.output,
84                                  input_shape = grad_input_shapes[1],
85                                  filter_shape = filter_shapes[1],
86                                  next_filter_shape = filter_shapes[2],
87                                  pooling_size = pooling_sizes[1],
88                                  activation_fn = activation_fn,
89                                  W = all_W[6], b = all_b[6])
90          self.ChannelAsisLayer2 = Convolutional3DPoolingLayer( \
91                                  input=self.ChannelAsisLayer1.output,
92                                  input_shape = input_shapes[1],
93                                  filter_shape = filter_shapes[1],
94                                  next_filter_shape = filter_shapes[2],
95                                  pooling_size = pooling_sizes[1],
```

```
                                    activation_fn = activation_fn,
                                    W = all_W[7], b = all_b[7])
        self.ChannelOptXLayer2 = Convolutional3DPoolingLayer( \
                                    input=self.ChannelOptXLayer1.output,
                                    input_shape = OF_input_shapes[1],
                                    filter_shape = opt_flow_filter_shapes[1],
                                    next_filter_shape = \
                                        opt_flow_filter_shapes[2],
                                    pooling_size = pooling_sizes[1],
                                    activation_fn = activation_fn,
                                    W = all_W[8], b = all_b[8])
        self.ChannelOptYLayer2 = Convolutional3DPoolingLayer(\
                                    input=self.ChannelOptYLayer1.output,
                                    input_shape = OF_input_shapes[1],
                                    filter_shape = opt_flow_filter_shapes[1],
                                    next_filter_shape = \
                                        opt_flow_filter_shapes[2],
                                    pooling_size = pooling_sizes[1],
                                    activation_fn = activation_fn,
                                    W = all_W[9], b = all_b[9])

         ### COMBINE ALL OUTPUTS FROM ALL CHANNELS
         layer2_output_recombined = T.concatenate(\
                                    [self.ChannelGradXLayer2.output.flatten(2),
                                    self.ChannelGradYLayer2.output.flatten(2),
                                    self.ChannelAsisLayer2.output.flatten(2),
                                    self.ChannelOptXLayer2.output.flatten(2),
                                    self.ChannelOptYLayer2.output.flatten(2) ],
                                    axis=1)

         ## LAYER 3
        self.DropoutFullyConnectedLayer3 = DropoutFullyConnectedLayer( \
                                        input = layer2_output_recombined,
                                        p = dropout_chance,
                                        minibatch_size = minibatch_size,
                                        n_in = input_shapes[2],
                                        n_out = filter_shapes[2],
                                        W = all_W[10], b = all_b[10],
                                        activation_fn = activation_fn)
        self.FullyConnectedLayer3 = FullyConnectedLayer( \
                                    input = layer2_output_recombined,
                                    n_in = input_shapes[2],
                                    n_out = filter_shapes[2],
                                    W = self.DropoutFullyConnectedLayer3.W,
```

```
                                        b = self.DropoutFullyConnectedLayer3.b,
                                        activation_fn = activation_fn)
        ## LAYER 4
        self.DropoutFullyConnectedLayer4 = DropoutFullyConnectedLayer( \
                                            input = \
                                            self.DropoutFullyConnectedLayer3.output,
                                            p = dropout_chance,
                                            minibatch_size = minibatch_size,
                                            n_in = filter_shapes[2],
                                            n_out = filter_shapes[3],
                                            W = all_W[11], b = all_b[11],
                                            activation_fn = activation_fn)
        self.FullyConnectedLayer4 = FullyConnectedLayer( \
                                    input = self.DropoutFullyConnectedLayer3.output,
                                    n_in = filter_shapes[2],
                                    n_out = filter_shapes[3],
                                    W = T.cast( \
                                        self.DropoutFullyConnectedLayer4.W*\
                                        (1-dropout_chance), theano.config.floatX),
                                    b = self.DropoutFullyConnectedLayer4.b,
                                    activation_fn = activation_fn)
        ## OUTPUT LAYER
        self.DropoutOutputLayer = SoftmaxLayer( \
                                    input = self.DropoutFullyConnectedLayer4.output,
                                    n_in = filter_shapes[3],
                                    n_out = num_of_output_classes,
                                    W = all_W[12],
                                    b = all_b[12])
        self.OutputLayer = SoftmaxLayer(
                            input = self.FullyConnectedLayer4.output,
                            n_in = filter_shapes[3],
                            n_out = num_of_output_classes,
                            W = T.cast(self.DropoutOutputLayer.W*(1 - \
                                    dropout_chance), theano.config.floatX),
                            b = self.DropoutOutputLayer.b)

        self.params =     self.ChannelGradXLayer1.params + \
                        self.ChannelGradYLayer1.params + \
                        self.ChannelAsisLayer1.params + \
                        self.ChannelOptXLayer1.params + \
                        self.ChannelOptYLayer1.params + \
                        self.ChannelGradXLayer2.params + \
                        self.ChannelGradYLayer2.params + \
                        self.ChannelAsisLayer2.params + \
```

```
184                         self.ChannelOptXLayer2.params + \
185                         self.ChannelOptYLayer2.params + \
186                         self.DropoutFullyConnectedLayer3.params +\
187                         self.DropoutFullyConnectedLayer4.params +\
188                         self.DropoutOutputLayer.params
189 
190         #Regularizers
191         #L1 regularizer #dropout layers excluded
192         self.L1 =          abs(self.ChannelGradXLayer1.W).sum() + \
193                         abs(self.ChannelGradYLayer1.W).sum() + \
194                         abs(self.ChannelAsisLayer1.W).sum() + \
195                         abs(self.ChannelOptXLayer1.W).sum() + \
196                         abs(self.ChannelOptYLayer1.W).sum() + \
197                         abs(self.ChannelGradXLayer2.W).sum() + \
198                         abs(self.ChannelGradYLayer2.W).sum() + \
199                         abs(self.ChannelAsisLayer2.W).sum() + \
200                         abs(self.ChannelOptXLayer2.W).sum() + \
201                         abs(self.ChannelOptYLayer2.W).sum() + \
202                         abs(self.OutputLayer.W).sum()
203 
204         # L2 regularizer #dropout layers excluded
205         self.L2 =          (self.ChannelGradXLayer1.W**2).sum() + \
206                         (self.ChannelGradYLayer1.W**2).sum() + \
207                         (self.ChannelAsisLayer1.W**2).sum() + \
208                         (self.ChannelOptXLayer1.W**2).sum() + \
209                         (self.ChannelOptYLayer1.W**2).sum() + \
210                         (self.ChannelGradXLayer2.W**2).sum() + \
211                         (self.ChannelGradYLayer2.W**2).sum() + \
212                         (self.ChannelAsisLayer2.W**2).sum() + \
213                         (self.ChannelOptXLayer2.W**2).sum() + \
214                         (self.ChannelOptYLayer2.W**2).sum() + \
215                         (self.OutputLayer.W**2).sum()
216 
217         self.DropoutNegativeLoglikelihood = \
218             self.DropoutOutputLayer.negativeLoglikelihood
219         self.negativeLoglikelihood = self.OutputLayer.negativeLoglikelihood
220         self.p_y_given_x = self.OutputLayer.p_y_given_x
221         self.y_pred = self.OutputLayer.y_pred
222         self.error = self.OutputLayer.error
223         self.DropoutError = self.OutputLayer.error
224         self.uni_seqID = uni_seqID
```

B.3 Convolutional Layer Classes

To perform the convolutions and max-pooling operation efficiently the `conv3d2d` and `downsample` are used from the Theano library.

```
import conv3d2d
import numpy as np
import theano
import theano.tensor as T
import theano.tensor.signal
import theano.tensor.signal.downsample
from util import soft_sign, mod_tanh

class Convolutional3DPoolingLayer(object):
    def __init__(self,input, input_shape, filter_shape, next_filter_shape,
                 pooling_size = (2,2), W=None, b=None, ignore_border = False,
                 activation_fn = mod_tanh):
        '''
        Activities included in the class:
            1) Weight initialization
            2) Convolution performed upon the input
            3) Pooling is performed upon the out of the convolutional layer
            4) Activation function is then performed upon the output of the
                convolutional layer
            5) Return output
        '''
        n_in = np.product(filter_shape)/filter_shape[0]
        n_out = np.product(next_filter_shape)/filter_shape[0]

        #Initialize the weights or load them in
        if W == None:
            self.W = theano.shared(\
                        np.asarray(np.random.uniform(\
                                        high = np.sqrt(6./(n_in+n_out)),
                                        low=-np.sqrt(6./(n_in+n_out)),
                                        size = filter_shape),
                                    dtype=theano.config.floatX),
                                   borrow=True,name='W')
        elif W.shape == filter_shape:
            self.W = theano.shared(W, borrow=True, name = 'W')
        else:
            raise ValueError('There is a shape mismatch between W and self.W')

```

```
39          if b == None:
40              self.b = theano.shared( \
41                          np.asarray(np.zeros(shape = filter_shape[0]),
42                              dtype=theano.config.floatX),borrow=True,name='b')
43          elif b.shape[0] == filter_shape[0]:
44              self.b = theano.shared(b, borrow=True, name='b')
45          else:
46              raise ValueError('There is a shape mismatch between b and self.b')
47
48          conv_out = conv3d2d.conv3d(signals = input,filters = self.W,
49                                     signals_shape = input_shape,
50                                     filters_shape = filter_shape,
51                                     border_mode = 'valid')
52          pool_out = T.signal.downsample.max_pool_2d(input=conv_out,
53                                                     ds=pooling_size,
54                                                     ignore_border=ignore_border)
55
56          self.output = activation_fn(pool_out + \
57                                          self.b.dimshuffle('x','x',0,'x','x'))
58
59          self.params = [self.W, self.b]
60
61  #Theano used for efficient implementation of Scharr Gradient
62  class GradXScharr(object):
63      def __init__(self, input, input_shape): #filter shape
64          self.W = np.asarray([[[[[-3., 0., 3.], [-10., 0., 10.],
65                                     [-3., 0., 3.]]]]],
66                              dtype=theano.config.floatX)
67
68          filter_shape = [1,1,1,3,3]
69
70          conv_out = conv3d2d.conv3d(signals=input, filters=self.W,
71                                     signals_shape=input_shape,
72                                     filters_shape=filter_shape,
73                                     border_mode='valid')
74
75          self.output = conv_out
76
77  #Theano used for efficient implementation of Scharr Gradient
78  class GradYScharr(object):
79      def __init__(self, input, input_shape):
80
81          self.W = np.asarray([[[[[-3., -10., -3.], [0. ,0. ,0. ],
82                                     [3. ,10. ,3. ]]]]],
```

```
                                dtype=theano.config.floatX)
        filter_shape = [1,1,1,3,3]

        conv_out = conv3d2d.conv3d(signals=input, filters=self.W,
                                   signals_shape = input_shape,
                                   filters_shape = filter_shape,
                                   border_mode='valid')

        self.output = conv_out
```

B.4 Fully-Connected Classes

```
import theano
import theano.tensor as T
import numpy as np
from util import soft_sign, mod_tanh

class FullyConnectedLayer(object):
    def __init__(self,input, n_in, n_out, W = None, b= None,
                 activation_fn = mod_tanh):
        '''
        Activities included in the class:
            1) Weight initialization or load in
            2) Linear combination of weight input product
            3) Application of activation function, elementwise
        '''
        self.n_in = n_in
        self.n_out = n_out

        if W == None:
            W = theano.shared(np.asarray( \
                                np.random.uniform(low = -np.sqrt(6./(n_in+n_out)),
                                                  high = np.sqrt(6./(n_in+n_out)),
                                                  size = (n_in, n_out)),
                                dtype= theano.config.floatX),name='W',
                                borrow=True)

        self.W = W

        if b == None:
            b = theano.shared(np.zeros(n_out,dtype=theano.config.floatX),
                                name='b', borrow=True)
```

```
31
32          self.b = b
33
34          linearly_combine = T.dot(input,self.W)+self.b
35          self.output = activation_fn(linearly_combine)
36
37          self.params = [self.W, self.b]
38
39  def apply_dropout(x, shape, p):
40      Shared_RNG_Stream = T.shared_randomstreams.RandomStreams()
41      #p is the probability of dropout
42      dropout_mask = Shared_RNG_Stream.binomial(size = shape, n = 1, p = (1-p))
43      return x*T.cast(dropout_mask, theano.config.floatX)
44
45  class DropoutFullyConnectedLayer(object):
46      def __init__(self,input, p, minibatch_size, n_in, n_out, W = None, b= None,
47                   activation_fn = mod_tanh):
48          '''
49          Activities included in the class:
50              1) Weight initialization or load in
51              2) Linear combination of weight input product
52              3) Application of activation function, elementwise
53              4) Apply dropout on output
54          '''
55          self.n_in = n_in
56          self.n_out = n_out
57
58          if W == None:
59              W = theano.shared(np.asarray(np.random.uniform( \
60                                              low = -np.sqrt(6./(n_in+n_out)),
61                                              high = np.sqrt(6./(n_in+n_out)),
62                                              size = (n_in, n_out)),
63                                  dtype= theano.config.floatX),
64                                  name='W', borrow=True)
65          self.W = W
66
67          if b == None:
68              b = theano.shared(np.zeros(n_out,dtype=theano.config.floatX),
69                      name='b', borrow=True)
70          self.b = b
71
72          linearly_combine = T.dot(input,self.W)+self.b
73          self.output = apply_dropout(activation_fn(linearly_combine),
74                                      shape = (minibatch_size, n_out), p = p)
```

```
75
76          self.params = [self.W, self.b]
```

B.5 Output Class

```
1  import numpy as np
2  import theano
3  import theano.tensor as T
4
5  class SoftmaxLayer(object):
6      '''
7      Activities included in the class:
8          1) Weight initialization or load in
9          2) Linear combination of weight input product
10         3) Application of Softmax function
11         4) Definition of error and negative log likelihood
12     '''
13     def __init__(self, input, n_in, n_out,W=None,b=None):
14         self.n_in = n_in
15         self.n_out = n_out
16
17         if W == None:
18             W = theano.shared(np.zeros(shape=(self.n_in,self.n_out),
19                                        dtype=theano.config.floatX),borrow=True)
20
21         self.W = W
22
23         if b == None:
24             b = theano.shared(np.zeros(shape=self.n_out,
25                                        dtype=theano.config.floatX), borrow=True)
26         self.b = b
27
28         self.params = [self.W, self.b]
29
30         self.p_y_given_x = T.exp(T.dot(input,self.W)+self.b)/(T.sum(T.exp( \
31                                 T.dot(input,self.W) +self.b),
32                                 axis=1)).dimshuffle(0,'x')
33
34         self.y_pred = T.argmax(self.p_y_given_x, axis=1)
35     #incorrect cases
36     def error(self, y):
37         if y.ndim != self.y_pred.ndim:
```

```
38              raise TypeError('y and y_pred must have the same dimensions')
39          if y.dtype.startswith('int'): #confirm both are integers
40              return T.mean(T.neq(self.y_pred, y))
41          else:
42              raise TypeError('Error with error method')
43      #negative log likelihood
44      def negativeLoglikelihood(self, y):
45          return -T.mean(T.log(self.p_y_given_x[T.arange(y.shape[0]), y]))
```

B.6 Training Method

The method included in this section instantiates an instance of the 3D CNN class, load in the data and then trains the network for prescribed number of epochs as described in Section 9.4. Included in the method is the definition of the dimensions of the feature maps, downsampling receptive fields and fully-connected layers.

```
1  import numpy as np
2  import theano.tensor as T
3  import matplotlib.pyplot as plt
4  import theano, time, random, os, sys, csv, cPickle
5
6  from layer_classes.util import load_data, save_data, load_data_gpu, \
7                                  swap_gpu_dataOF, swap_aug_gpu_dataOF, \
8                                  graph_training_results, soft_sign, mod_tanh, \
9                                  ReLU
10 from layer_classes.data_augmentation import cube_all_seq
11 from nets import ConvolutionalNeuralNetwork
12
13 def main_loop(filename, dataset_code, init_params_filename = None,
14                 minibatch_size = 128, minibatches_to_gpu = 1,
15                 num_of_output_classes = 6, momentum = 0.9,
16                 activation_fn = ReLU, freq_validation_check = 5,
17                 epoch_patience = 1, initial_learning_rate = 0.002,
18                 learning_rate_decay = 0.985,
19                 initial_dropout_learning_rate = 0.3,
20                 dropout_learning_rate_decay = 0.98,
21                 L1_lambda = 0.0, L2_lambda = 0.0, output_save_path = "/Outputs/",
22                 current_epoch = 0, squared_incoming_weight_constraint = 15.0,
23                 dropout_chance = 0.5):
24     '''
25     Structure of main loop as follows:
26         - (1) Load data into main memory
```

```
27          - (2) Load randomly selected minibatch onto GPU
28          - (3) Build theano functions
29              (3.1) Training function
30              (3.2) Learning rate update methodology
31              (3.3) Dropout constraint
32              (3.4) Momentum
33              (3.5) Test function
34          - (4) Setup training monitoring parameters
35          - (5) Train
36              -(5.1) Swap out randomly selected minibatch
37              -(5.2) Train for an epoch(s)
38              -(5.3) Update learning rates when necessary
39              -(5.4) Check test error
40          - (6) Output training results (e.g. change in validation error)
41                  and save results
42      '''
43
44      #(1) LOAD DATA ONTO MAIN MEMORY
45      print 'Loading data onto main memory...'
46
47      load_start_time = time.clock()
48      train, validate, test = load_data(filename)
49
50      y_train_set, x_train_set = train
51      y_train_labels = np.asarray([y[0] for y in y_train_set], dtype = int)
52      n_train_batches = 24
53
54      x_test_set, y_test_set = cube_all_seq(X_data = test[1], y_data = test[0])
55      y_test_labels = np.asarray([y[0] for y in y_test_set], dtype = int)
56      n_test_batches = x_test_set.shape[0]/minibatch_size
57      seqId_test_set = [y[1]+'_'+y[2]+'_'+y[3]+'_'+str(y[4])for y in y_test_set]
58      uni_seqId_test_set = list(set(seqId_test_set))
59      vidId_test_set = [seqId[:-2] for seqId in seqId_test_set]
60      uni_vidId_test_set =  list(set(vidId_test_set))
61
62      print '... %i training cases in %i batches, %i test cases '\
63              'in %i batches loaded onto main memory in %.fs' % \
64              (x_train_set.shape[0], n_train_batches, x_test_set.shape[0],
65                  n_test_batches, time.clock()-load_start_time)
66
67      train_action_ids = range(6)
68      count_train_action_ids = np.zeros(shape = len(train_action_ids))
69
70      count = 0
```

```
    for y_action_id in y_train_set[:,0]:
        count += 1
        if count < 5:
            print 'y_action_id for iter: ',count, ' y_action_id ',y_action_id, \
                    ' type(y_action_id): ', type(y_action_id)
        count_train_action_ids[int(y_action_id)] += 1

    print ' Count of training cases according to action id: ', \
                                zip(train_action_ids, count_train_action_ids)

    #(1.1) LOAD INIT PARAMETERS
    if init_params_filename != None:
        all_init_params = cPickle.load(open(init_params_filename, 'r'))
        all_W = all_init_params[::2]
        all_b = all_init_params[1::2]
    else:
        all_W = [None]*13
        all_b = [None]*13

    #Create a random index
    randomized_index = range(x_train_set.shape[0])
    random.shuffle(randomized_index)

     #(2) LOAD RANDOMLY SELECTED MINIBATCH ONTO THE GPU
     print 'Loading randomly selected set of cases onto GPU...'
    current_y, current_x, current_x_optx, current_x_opty = \
                    load_dataOF_gpu(y_train_labels, x_train_set,
                    minibatch_size, randomized_index, gpu_set_index=0,
                    minibatches_to_gpu = minibatches_to_gpu)

    print '...data loaded onto GPU, building theano functions...'
    model_build_start = time.clock()

    #(3) BUILD THEANO FUNCTIONS AND SYMBOLIC VARIABLES
    ftensor4 = T.TensorType(theano.config.floatX,(False,False,False,False))
    X = ftensor4(name='X')
    X_optx = ftensor4(name='X_optx')
    X_opty = ftensor4(name='X_opty')
    y = T.ivector(name='y')
    index = T.iscalar()

    input_shapes = [(minibatch_size, 9, 1, 60, 80), \
                    (minibatch_size, 5, 10, 18, 24), 1200]
    grad_input_shapes = [(minibatch_size, 9, 1, 58, 78), \
```

```
                            (minibatch_size, 5, 10, 18, 24)]
    OF_input_shapes = [(minibatch_size, 8, 1, 60, 80), \
                       (minibatch_size, 4, 10, 18, 24)]

    #Resize input in preparation for conv3d2d method
    X_resized = X.reshape(input_shapes[0])
    X_optx_resized = X_optx.reshape(OF_input_shapes[0])
    X_opty_resized = X_opty.reshape(OF_input_shapes[0])

    filter_shapes = [(10, 5, 1, 7, 9), (10, 5, 10, 7, 7), 300, 100]
    opt_flow_filter_shapes = [(10, 5, 1, 7, 9), (10, 4, 10, 7, 7), 300]
    pooling_sizes = [(3,3), (3,3)]

    architecture = [['input_shapes',input_shapes], \
                    ['grad_input_shapes',grad_input_shapes], \
                    ['OF_input_shapes', OF_input_shapes], \
                    ['filter_shapes', filter_shapes],\
                    ['pooling_sizes', pooling_sizes]]

    classifier = ConvolutionalNeuralNet(    input = X_resized,
                                            OF_input = [X_optx_resized,
                                                        X_opty_resized],
                                            input_shapes=input_shapes,
                                            dropout_chance = \
                                                np.asarray(dropout_chance, \
                                                  dtype=theano.config.floatX),
                                            minibatch_size = minibatch_size,
                                            grad_input_shapes=grad_input_shapes,
                                            OF_input_shapes=OF_input_shapes,
                                            filter_shapes= filter_shapes,
                                            opt_flow_filter_shapes = \
                                                opt_flow_filter_shapes,
                                            pooling_sizes= pooling_sizes,
                                            num_of_output_classes = \
                                                num_of_output_classes,
                                            activation_fn = activation_fn,
                                            all_W = all_W, all_b = all_b)

    #(3.3) LEARNING RATE UPDATE FUNCTION - Exponential decay and Momentum
    cost = classifier.DropoutNegativeLoglikelihood(y)
            # + L1_lambda*classifier.L1 + L2_lambda*classifier.L2
    gparams = [T.grad(cost=cost, wrt=param) for param in classifier.params]

    #initialize learning parameters
```

```
159     epoch = current_epoch
160     learning_rate = theano.shared( \
161         np.asarray(initial_learning_rate).astype(theano.config.floatX))
162     dropout_learning_rate = theano.shared( \
163         np.asarray(initial_dropout_learning_rate).astype(theano.config.floatX))
164 
165     #Momentum
166     delta_params = [theano.shared(np.zeros(param.get_value().shape,
167                     dtype=theano.config.floatX),
168                     borrow=True) for param in classifier.params]
169 
170     updates = []
171     for ind, param, gparam, delta_param in zip(range(len(classifier.params)),\
172                                     classifier.params, gparams, delta_params):
173         if param.ndim == 2 and len(classifier.params) - 2 !=  ind:
174         #fully connect layer W parameters (exclude the output connected layer)
175             updated_param = param - dropout_learning_rate*gparam + \
176                                 momentum*delta_param #updated value
177 
178             #L2 incoming weight vector
179             scale_param = T.minimum( 1,\
180                             T.sqrt(squared_incoming_weight_constraint/T.sum(\
181                                 T.sqr(updated_param), axis = 0)))
182 
183             updates.append((param, updated_param*scale_param ))
184             # rescales parameters if after the weight update the L2
185             #   penalty of the weights is greater than
186             #    squared_incoming_weight_constraint
187 
188         else: #includes convolutional layers and biases for all layers ->
189             updates.append((param, param - learning_rate*gparam + \
190                                 momentum*delta_param))
191 
192 
193     # NOTE: Learning rate update occur in the training loop
194     training_function = theano.function(inputs=[index],
195                             outputs=[classifier.DropoutNegativeLoglikelihood(y),
196                                         classifier.DropoutError(y)],
197                             updates=updates,
198                             givens={X:current_x[index*minibatch_size: \
199                                                 (index+1)*minibatch_size],
200                                     X_optx:current_x_optx[index*minibatch_size: \
201                                                 (index+1)*minibatch_size],
202                                     X_opty:current_x_opty[index*minibatch_size: \
```

```
                                                (index+1)*minibatch_size],
                                    y:current_y[index*minibatch_size: \
                                                (index+1)*minibatch_size]})

    #(3.4) TESTING FUNCTION
    testing_function = theano.function(inputs=[index],
                            outputs=[classifier.p_y_given_x,classifier.error(y),
                                        y, classifier.y_pred],
                            givens={X:current_x[index*minibatch_size: \
                                                (index+1)*minibatch_size],
                                    X_optx:current_x_optx[index*minibatch_size:\
                                                (index+1)*minibatch_size],
                                    X_opty:current_x_opty[index*minibatch_size:\
                                                (index+1)*minibatch_size],
                                    y:current_y[index*minibatch_size: \
                                                (index+1)*minibatch_size]})

    print 'Finished building models in %.1fs'%(time.clock()-model_build_start)

    # Recording parameters:
    start_training_time = time.clock()
    training_nlls = []
    training_errors = []
    test_cube_errors = []
    test_seq_errors = []
    test_vid_errors = []
    average_parameter_value = []
    stdev_parameter_value = []

    #(5) TRAIN LOOP
    print 'Beginning training at epoch: %i ' % epoch
    while epoch < epoch_patience:
        epoch += 1
        training_time_start = time.clock()

        #Randomize index for drawing cases from training data
        random.shuffle(randomized_index)

        #Reset temp parameters
        temp_training_nlls = []
        temp_training_errors = []

        #Train for an epoch
        is_a_final_set = False if x_train_set.shape[0]%minibatch_size == 0 \
```

```
                        else True
        for gpu_set_index in xrange(n_train_batches/minibatches_to_gpu \
                                + int(is_a_final_set)):
            final_set = True if gpu_set_index == \
                        (n_train_batches/minibatches_to_gpu) and
                        is_a_final_set else False
            #Load in training set onto GPU
            swap_aug_gpu_dataOF(current_y, current_x,
                        current_x_optx, current_x_opty,
                        y_train_labels, x_train_set,
                        block_size = minibatch_size,
                        random_seq_sample = True,
                        augment_data = True)

            #Run model through data loaded on GPU
            for minibatch_index in xrange(minibatches_to_gpu):

                #MOMENTUM: Update deltas
                if minibatch_index >= 1 or epoch >= 2+current_epoch:
                    [delta_param.set_value(np.asarray( \
                            curr_param.get_value(borrow=True)-prev_param,
                            dtype=theano.config.floatX), borrow=False) \
                            for curr_param, prev_param, delta_param in \
                     zip(classifier.params, prev_params, delta_params)]

                #MOMENTUM: Save current params
                prev_params = [param.get_value(borrow=False) for \
                                param in classifier.params]

                out = training_function(minibatch_index)
                temp_training_nlls += [out[0]]
                temp_training_errors += [out[1]]

        #Save recording parameters
        training_nlls += [np.mean(temp_training_nlls)]
        training_errors += [np.mean(temp_training_errors)]

        print 'Epoch # %i, time to train: %.1fs Training error: %f%% NLL: %f' %\
                    (epoch, (time.clock()-training_time_start),
                        training_errors[-1]*100.,training_nlls[-1])

        #Update learning rate
        learning_rate.set_value(\
            (learning_rate.get_value()*learning_rate_decay).astype(\
```

```
291                 theano.config.floatX))
292         dropout_learning_rate.set_value(\
293             (dropout_learning_rate.get_value()*\
294                 dropout_learning_rate_decay).astype(theano.config.floatX))
295 
296         #Check test error
297         if epoch%freq_validation_check == 0:
298 
299             # Get test error through vote on seq's
300             print 'Getting test error...'
301             start_test_check = time.clock()
302 
303             test_p_y_given_x = np.zeros(shape=(x_test_set.shape[0], \
304                                                num_of_output_classes))
305             temp_test_errors = np.zeros(shape=(x_test_set.shape[0]))
306 
307             final_set = False
308             for gpu_set_index in xrange(n_test_batches/minibatches_to_gpu):
309                 swap_gpu_dataOF(current_y, current_x,
310                                 current_x_optx, current_x_opty,
311                                 y_test_labels, x_test_set,
312                                 minibatch_size = minibatch_size,
313                                 randomized_index = None,
314                                 gpu_set_index = gpu_set_index,
315                                 minibatches_to_gpu = minibatches_to_gpu,
316                                 final_set = final_set)
317                 for i in xrange(minibatches_to_gpu):
318                     out = testing_function(i)
319                     test_p_y_given_x[gpu_set_index*minibatch_size:\
320                                      (gpu_set_index+1)*minibatch_size]=out[0]
321                     temp_test_errors[gpu_set_index*minibatch_size:\
322                                      (gpu_set_index+1)*minibatch_size]=out[1]
323 
324             #Final set
325             if x_test_set.shape[0]%minibatch_size != 0:
326                 final_set = True
327                 swap_gpu_dataOF(current_y, current_x,
328                     current_x_optx, current_x_opty,
329                     y_test_labels, x_test_set,
330                     minibatch_size = minibatch_size,
331                     randomized_index = None,
332                     gpu_set_index = gpu_set_index,
333                     minibatches_to_gpu = minibatches_to_gpu,
334                     final_set = final_set)
```

```
335
336                 for i in xrange(minibatches_to_gpu):
337                     out = testing_function(i)
338                     if minibatches_to_gpu > 1:
339                         sys.exit("DEPRECATED minibatches_to_gpu must be 1")
340                     test_p_y_given_x[-minibatch_size:] = out[0]
341                     temp_test_errors[-minibatch_size:] = out[1]
342
343             #TEST ERROR: CUBE LEVEL
344             test_cube_error_rate = np.mean(temp_test_errors)
345             test_cube_errors.append(test_cube_error_rate)
346
347             #TEST ERROR: SEQUENCE LEVEL
348             test_seq_actual_y = np.zeros(shape = (len(uni_seqId_test_set)),\
349                                          dtype = 'int' )
350             test_seq_aggregate_p_y_given_x = np.zeros( shape = \
351                 (len(uni_seqId_test_set), num_of_output_classes), \
352                 dtype = 'float')
353
354             #sum p_y_given_x for matching seqIDs
355             for uni_seq_ind in xrange(len(uni_seqId_test_set)):
356                 for case_ind in xrange(test_p_y_given_x.shape[0]):
357                     if seqId_test_set[case_ind] == \
358                                          uni_seqId_test_set[uni_seq_ind]:
359                         test_seq_aggregate_p_y_given_x[uni_seq_ind,:] += \
360                                          test_p_y_given_x[case_ind,:]
361                         #aggregate p_y_given_x for the same sequence
362                         test_seq_actual_y[uni_seq_ind] = y_test_set[case_ind,0]
363
364             test_seq_predicted_y = np.argmax(test_seq_aggregate_p_y_given_x, \
365                                              axis=1)
366             test_seq_error_rate = 1 - np.mean(np.equal(test_seq_predicted_y, \
367                                                        test_seq_actual_y))
368             test_seq_errors.append(test_seq_error_rate)
369
370             #TEST ERROR: VIDEO LEVEL
371             test_vid_actual_y = np.zeros(shape = (len(uni_vidId_test_set)), \
372                                          dtype = 'int' )
373             test_vid_aggregate_p_y_given_x = np.zeros( shape = \
374                         (len(uni_vidId_test_set), num_of_output_classes), \
375                         dtype = 'float')
376
377             #sum p_y_given_x for matching seqIDs
378             for uni_vid_ind in xrange(len(uni_vidId_test_set)):
```

```
379                 for case_ind in xrange(test_p_y_given_x.shape[0]):
380                     if vidId_test_set[case_ind] == \
381                             uni_vidId_test_set[uni_vid_ind]:
382                         test_vid_aggregate_p_y_given_x[uni_vid_ind,:] += \
383                                                 test_p_y_given_x[case_ind,:]
384                         #aggregate p_y_given_x for the same sequence
385                         test_vid_actual_y[uni_vid_ind] = y_test_set[case_ind, 0]
386 
387             test_vid_predicted_y = np.argmax(test_vid_aggregate_p_y_given_x,\
388                                                 axis=1)
389             test_vid_error_rate = 1 - np.mean(np.equal(test_vid_predicted_y,\
390                                                     test_vid_actual_y))
391             test_vid_errors.append(test_vid_error_rate)
392 
393             print 'TEST ERROR - Cube: %f %% \tSequence: %f %% \tVideo: %f %%'%\
394                     (test_cube_error_rate*100, test_seq_error_rate*100, \
395                         test_vid_error_rate*100)
396 
397 
398     print '... Training complete in %i s' % (time.clock()-start_training_time)
399 
400     #(7) SAVE RESULTS
401     localtime = time.localtime(time.time())
402     year = localtime.tm_year
403     month = localtime.tm_mon
404     day = localtime.tm_mday
405     hour = localtime.tm_hour
406 
407     yyyymmddhh = str(year) + str(month) + str(day) +str(hour)
408 
409     #Best parameters
410     best_params_filename = output_save_path+yyyymmddhh+"_"+ \
411                             dataset_code+"_"+os.path.basename(__file__)[:-3]+\
412                             "_best_parameters.pkl"
413     save_data(best_params, best_params_filename)
414 
415     last_params = [param.get_value(borrow=False) for param in classifier.params]
416     last_params_filename = output_save_path+yyyymmddhh+"_"+dataset_code+"_"+\
417                             os.path.basename(__file__)[:-3] + \
418                             "_last_parameters.pkl"
419 
420     save_data(last_params, last_params_filename)
```

B.7 Data Augmentation Methods

To increase the speed of augmentation, the `fast_warp` method by Dieleman (2014) was used.

```
import numpy as np
import skimage.transform
import cPickle
import time
import sys
import matplotlib.pyplot as plt

#LOAD DATA
def load_data(dataset):
    return cPickle.load(open(dataset, 'r'))

#INDIVIDUAL FRAME TRANSFORMATION METHOD:
def build_frame_transformation(translation=[0,0], zoom=1., shear=0,
                                rotation=0):

    #build transformation function
    transformation = skimage.transform.AffineTransform(\
                        scale=(1/zoom, 1/zoom), rotation=np.deg2rad(rotation),\
                        shear = np.deg2rad(shear), translation=(translation[0], \
                        translation[1]))

    return transformation

#RANDOM GENERATION OF AUGMENTATION PARAMETERS
def generate_augmentation_parameters(translation_range=[0,3],
                                        zoom_range=[0.97,1.02]):
    #generate transformation values
    x_translate = np.random.randint(*translation_range)
    y_translate = np.random.randint(*translation_range)
    zoom = np.random.uniform(*zoom_range)
    shear = 0
    rotation = 0
    return [x_translate, y_translate], zoom, shear, rotation

def fast_warp(img, tf, output_shape=(9,1,120,160), mode='reflect'):
    '''
    This wrapper function is about five times faster than
      skimage.transform.warp, for our use case.
```

```
    Source: http://benanne.github.io/2014/04/05/galaxy-zoo.html
    '''
    m = tf._matrix
    img_wf = np.empty(output_shape, dtype='float32')
    for k in xrange(output_shape[0]):
        img_wf[...] = skimage.transform._warps_cy._warp_fast(\
                        img, m, output_shape=(output_shape[2],\
                                              output_shape[3]), mode=mode)
    return img_wf

#APPLY AUGMENTATIONS TO CUBE
def apply_aug_to_cube(cube, num_frames =  9 , height = 60, width = 80,
                        output_shape = (60,80)):
    #loop through frames applying the same transformation to each
    augment_seq = np.zeros(shape=(num_frames, height, width), dtype="float32")
    augment_params = generate_augmentation_parameters()
    augment_func = build_frame_transformation(*augment_params)

    flip = True if np.random.randint(2) > 0 else False

    for i, frame in enumerate(cube):
        #augment each frame in the same manner
        #aug_frame = augment_func(frame)
        aug_frame =skimage.transform._warps_cy._warp_fast(frame-np.min(frame),
                        augment_func._matrix, output_shape=(height, width),
                        mode='reflect')
        if flip:
            aug_frame = np.fliplr(aug_frame)

        #downsample
        ds_frame = skimage.transform.resize(image=aug_frame,
                        output_shape = output_shape)

        augment_seq[i, :, :] = ds_frame
    return augment_seq

#SEQUENCE MANIPULATION
def ds_fps(seq, ds_factor=2):
    #downsample fps by ds_factor
    return [seq[np.arange(seq.shape[0])%ds_factor==i,:,:] \
                for i in xrange(ds_factor)]

#DIVIDE SEQUENCE INTO 9-FRAME CUBES
def divide_into_cubes(seq, cube_length=9):
```

```
    if seq.shape[0] < cube_length:
        sys.exit('Sequence too short for cube length: %i' % seq.shape[0])
    #return list of cubes of length cube_length
    return [seq[ind*cube_length:(ind+1)*cube_length] \
                for ind in xrange(seq.shape[0]/cube_length)]

#GENERATE MINI-BATCH DATA
def generate_block_input_data(X_data, y_data, block_size = 128,
            max_frame_crop = 5, mean_centre = False, augment_data = True):
    #make block of data ready for the net
    #where block size is appropriate for gpu load in
    X_data = np.array(X_data) if type(X_data) != np.ndarray else X_data
    y_data = np.array(y_data) if type(y_data) != np.ndarray else y_data

    block_data_x = []
    block_data_y = []

    cube_count = 0
    #cube data and augment frames
    for ind, seq in enumerate(X_data):
        #randomly crop [0,5] frames from sequence
        seq = seq[np.random.randint(max_frame_crop):]
        #ds_fps and cube
        cubes = np.concatenate([divide_into_cubes(ds_seq) \
                                    for ds_seq in ds_fps(seq)])
        #frame augmentation for each cube
        for ind_cube, cube in enumerate(cubes):
            #augment cube
            if augment_data:
                block_data_x.append(apply_aug_to_cube(cube))
                block_data_y.append(y_data[ind])
            else:
                block_data_x.append(cube)
                block_data_y.append(y_data[ind])

            cube_count += 1

            if cube_count == block_size: break #end conditions
        if cube_count == block_size: break #end conditions

    if mean_centre:
        block_data_x = np.asarray(block_data_x)
        average_case = np.mean(block_data_x, axis = 0)
        retval = []
```

```
        for i in xrange(block_data_x.shape[0]):
            retval.append([block_data_x[i] - average_case])

        return np.asarray(retval), np.asarray(block_data_y)
    else:
        return np.asarray(block_data_x), np.asarray(block_data_y)

#CUBE ALL DATA
def cube_all_seq(X_data, y_data, mean_centre = False, max_frame_crop = 5,
                    augment_data = False):

    X_data = np.array(X_data) if type(X_data) != np.ndarray else X_data
    y_data = np.array(y_data) if type(y_data) != np.ndarray else y_data

    block_data_x = []
    block_data_y = []

    cube_count = 0
    #cube data and augment frames
    for ind, seq in enumerate(X_data):
        #randomly crop [0,5] frames from sequence
        seq = seq[np.random.randint(max_frame_crop):]
        #ds_fps and cube
        cubes = np.concatenate([divide_into_cubes(ds_seq) \
                                    for ds_seq in ds_fps(seq)])
        #frame augmentation for each cube
        for ind_cube, cube in enumerate(cubes):
            #augment cube
            if augment_data:
                block_data_x.append(apply_aug_to_cube(cube))
                block_data_y.append(y_data[ind])
            else:
                block_data_x.append(cube)
                block_data_y.append(y_data[ind])

            cube_count += 1

    if mean_centre:
        block_data_x = np.asarray(block_data_x)
        average_case = np.mean(block_data_x, axis = 0)
        retval = []
        for i in xrange(block_data_x.shape[0]):
            retval.append([block_data_x[i] - average_case])

```

```
171         return np.asarray(retval), np.asarray(block_data_y)
172     else:
173         return np.asarray(block_data_x), np.asarray(block_data_y)
```

B.8 Utility Methods

A collection of utility methods including methods to load and swap memory on the GPU memory, optical flow calculation using OpenCV (Bradski, 2000) and activation functions are included in code in this section. The algorithm developed by Farnebäck (2003) implemented in the OpenCV is used to calculate optical flow efficiently in the hardwired layer. As the data was too large to be loaded entirely onto the GPU's memory, it was necessary to swap the data on and off the device.

```
1  import theano, time, cPickle, random, cv2
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import theano.tensor as T
5  from data_augmentation import generate_block_input_data
6
7  def soft_sign(x):
8      return x/(1+T.abs_(x))
9
10 def mod_tanh(x):
11     return 1.7159*T.tanh((2./3.)*x)
12
13 def ReLU(x):
14     return T.maximum(0., x)
15
16 def load_data(filename):
17     return cPickle.load(open(filename,'r'))
18
19 def save_data(data, filename):
20     cPickle.dump(data , open(filename, 'w'))
21
22 #LOAD INITIAL DATA ONTO GPU
23 def load_dataOF_gpu(y_data, x_data, minibatch_size, randomized_index = None,
24                     gpu_set_index=0, minibatches_to_gpu=1):
25
26     if randomized_index == None: #assume no randomization then
27         randomized_index = range(x_data.shape[0])
28
```

```
    #JIT opt calculation
    x_optx, x_opty = opticalFlow( \
                        x_data[ randomized_index[ \
                          (minibatch_size*minibatches_to_gpu)*gpu_set_index: \
                          (minibatch_size*minibatches_to_gpu)*(gpu_set_index+1)]])

    #convert y_data to ndarray type
    y_data = np.asarray(y_data)

    #load onto GPU
    shared_x = theano.shared( \
                    x_data[randomized_index[\
                        (minibatch_size*minibatches_to_gpu)*gpu_set_index: \
                        (minibatch_size*minibatches_to_gpu)*\
                        (gpu_set_index+1)]].astype(theano.config.floatX),
                        borrow = True)

    shared_x_optx = theano.shared(np.asarray(x_optx, dtype=theano.config.floatX),
                        borrow = True)
    shared_x_opty = theano.shared(np.asarray(x_opty, dtype=theano.config.floatX),
                        borrow = True)
    shared_y = theano.shared( \
                    y_data[randomized_index[\
                        (minibatch_size*minibatches_to_gpu)*gpu_set_index: \
                        (minibatch_size*minibatches_to_gpu)*(gpu_set_index+1)]].astype \
                            (theano.config.floatX), borrow = True)

    return theano.tensor.cast(shared_y,'int32'), shared_x, shared_x_optx, shared_x_opty

#SWAP DATA ON GPU
# Necessary as data is larger than GPU memory
def swap_aug_gpu_dataOF(current_y, current_x, current_x_optx, current_x_opty,
                        y_data, x_data, block_size, random_seq_sample = True,
                        augment_data = True):

    if random_seq_sample == True: #assume no randomization then
        randomized_index = range(x_data.shape[0])
        random.shuffle(randomized_index) #inplace shuffling
    else:
        randomized_index = range(x_data.shape[0])

    #perform augmentation using a randomized ordering of the sequences
    are_nan = True
    while are_nan:
```

```
73          aug_x_data, aug_y_data = generate_block_input_data( \
74                                                  X_data = x_data[randomized_index],
75                                                  y_data = y_data[randomized_index],
76                                                  block_size = block_size,
77                                                  max_frame_crop = 5,
78                                                  mean_centre = False,
79                                                  augment_data = True)
80          if np.sum(np.isnan(aug_x_data)) == 0 and \
81              np.sum(np.isnan(aug_y_data)) == 0 :
82              are_nan = False
83          else:
84              print "****NAN found in data Augmentation process****"
85
86      #JIT opt calculation
87      x_optx, x_opty = opticalFlow(aug_x_data)
88
89      #load onto GPU
90      current_x.set_value(aug_x_data.astype(theano.config.floatX),borrow = True)
91      current_x_optx.set_value(np.asarray(x_optx, dtype=theano.config.floatX),\
92                                  borrow = True)
93      current_x_opty.set_value(np.asarray(x_opty, dtype=theano.config.floatX),\
94                                  borrow = True)
95
96      if theano.config.device != 'cpu':
97          current_y.owner.inputs[0].owner.inputs[0].set_value(aug_y_data.astype\
98                                  (theano.config.floatX), borrow = True)
99      else:
100         current_y.owner.inputs[0].set_value(aug_y_data.astype \
101                                 (theano.config.floatX), borrow = True)
102
103 #OPTICAL FLOW CALCULATED JUST-IN-TIME USING OPENCV2
104 def opticalFlow(clip_values):
105     ret_list = [[],[]]
106
107     for clip in clip_values:
108     empty_flow = np.zeros_like(clip[0])
109         prev_frame = clip[0]
110         temp = []
111         temp_flow_x = []
112         temp_flow_y = []
113         for ind in xrange(1,len(clip)):
114             curr_frame = clip[ind]
115             temp = cv2.calcOpticalFlowFarneback(prev=prev_frame,
116                     next=curr_frame, flow = empty_flow, pyr_scale=0.5,
```

```
117                     levels=1, winsize=5, iterations=5, poly_n = 5,
118                     poly_sigma=1.5, flags=0)
119             prev_frame[:] = curr_frame
120             temp_flow_x += [temp[:,:,1]]
121             temp_flow_y += [temp[:,:,0]]
122 
123         ret_list[0].append(np.asarray(temp_flow_x))
124         ret_list[1].append(np.asarray(temp_flow_y))
125 
126     return ret_list
```

Bibliography

Almeida, L., Langlois, T., Amaral, J. D., and Redol, R. A. (1997). On-line step size adaptation. Technical report, INESC. 9 Rua Alves Redol, 1000.

Almeida, L. B., Langlois, T., Amaral, J. D., and Plakhov, A. (1998). Parameter adaptation in stochastic optimization. In Saad, D., editor, *On-line Learning in Neural Networks*, pages 111–134. Cambridge University Press, New York, NY, USA.

Baccouche, M., Mamalet, F., Wolf, C., Garcia, C., and Baskurt, A. (2011). Sequential deep learning for human action recognition. In Salah, A. and Lepri, B., editors, *Human Behavior Understanding*, volume 7065 of *Lecture Notes in Computer Science*, pages 29–39. Springer Berlin Heidelberg.

Banerji, M., Lahav, O., Lintott, C. J., Abdalla, F. B., Schawinski, K., Bamford, S. P., Andreescu, D., Murray, P., Raddick, M. J., Slosar, A., et al. (2010). Galaxy Zoo: reproducing galaxy morphologies via machine learning. *Monthly Notices of the Royal Astronomical Society*, 406(1):342–353.

Bengio, Y., Courville, A., and Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation.

Berndt, E. K., Hall, B. H., and Hall, R. E. (1974). Estimation and inference in nonlinear structural models. In *Annals of Economic and Social Measurement*, volume 3, pages 103–116. NBER.

Bishop, C. M. et al. (2006). *Pattern recognition and machine learning*, volume 1. Springer New York.

Bordes, A., Bottou, L., and Gallinari, P. (2009). SGD-QN: Careful Quasi-Newton Stochastic Gradient Descent. *The Journal of Machine Learning Research*, 10:1737–1754.

Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In Lechevallier, Y. and Saporta, G., editors, *Proceedings of COMPSTAT'2010*, pages 177–186. Physica-Verlag HD.

Bradski, G. (2000). The OpenCV library. *Dr. Dobb's Journal of Software Tools*.

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

Breiman, L., Friedman, J., Stone, C. J., and Olshen, R. A. (1984). *Classification and regression trees*. CRC press.

Caruana, R. and Niculescu-Mizil, A. (2006). An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 161–168, New York, NY, USA. ACM.

Cireşan, D., Meier, U., Masci, J., and Schmidhuber, J. (2012a). Multi-column deep neural network for traffic sign classification. *Neural Networks*, 32(0):333 – 338.

Cireşan, D., Meier, U., and Schmidhuber, J. (2012b). Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3642–3649.

Cireşan, D. C., Giusti, A., Gambardella, L. M., and Schmidhuber, J. (2013). Mitosis detection in breast cancer histology images with deep neural networks. In Mori, K., Sakuma, I., Sato, Y., Barillot, C., and Navab, N., editors, *Medical Image Computing and Computer-Assisted Intervention (MICCAI) 2013*, volume 8150 of *Lecture Notes in Computer Science*, pages 411–418. Springer Berlin Heidelberg.

Cireşan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. (2010). Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 22(12):3207–3220.

Cireşan, D. C., Meier, U., Masci, J., Gambardella, L. M., and Schmidhuber, J. (2011). High-performance neural networks for visual object classification. *CoRR*, abs/1102.0183.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3):273–297.

Dalal, N. and Triggs, B. (2005). Histograms of oriented gradients for human detection. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2005.*, volume 1, pages 886–893.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition, 2009.*, pages 248–255. IEEE.

Dieleman, S. (2014). My solution for the Galaxy Zoo challenge. http://benanne.github.io/2014/04/05/galaxy-zoo.html. [Online, accessed 23-May-2014].

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159.

Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P., and Bengio, S. (2010). Why does unsupervised pre-training help deep learning? *The Journal of Machine Learning Research*, 11:625–660.

Farnebäck, G. (2003). Two-frame motion estimation based on polynomial expansion. In Bigun, J. and Gustavsson, T., editors, *Image Analysis*, volume 2749 of *Lecture Notes in Computer Science*, pages 363–370. Springer Berlin Heidelberg.

Fukushima, K. (1988). Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks*, 1(2):119–130.

Fukushima, K., Miyake, S., and Ito, T. (1983). Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-13(5):826–834.

Gao, Z., Chen, M.-Y., Hauptmann, A. G., and Cai, A. (2010). Comparing evaluation protocols on the KTH dataset. In Salah, A., Gevers, T., Sebe, N., and Vinciarelli, A., editors, *Human Behavior Understanding*, volume 6219 of *Lecture Notes in Computer Science*, pages 88–100. Springer Berlin Heidelberg.

Geman, S., Bienenstock, E., and Doursat, R. (1992). Neural networks and the bias-variance dilemma. *Neural computation*, 4(1):1–58.

Geronimo, D., Lopez, A. M., Sappa, A. D., and Graf, T. (2010). Survey of pedestrian detection for advanced driver assistance systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(7):1239–1258.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feed-forward neural networks. In *International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, pages 249–256.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier networks. In *14th International Conference on Artificial Intelligence and Statistics*, volume 15, pages 315–323.

Goodfellow, I. J., Bulatov, Y., Ibarz, J., Arnoud, S., and Shet, V. (2014). Multi-digit number recognition from street view imagery using deep convolutional neural networks. *CoRR*, abs/1312.6082.

Graves, A., Mohamed, A.-R., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6645–6649. IEEE.

Harvey, D., Lintott, C., Kitching, T., Marshall, P., and Willett, K. (2013). Kaggle - Galaxy Zoo competition. http://www.kaggle.com/c/galaxy-zoo-the-galaxy-challenge. [Online, accessed 20-May-2014].

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The elements of statistical learning: data mining, inference and prediction*, chapter Neural Networks, pages 389–415. Springer, 2nd edition.

Hinton, G. E. (2012). A practical guide to training restricted Boltzmann machines. In Montavon, G., Orr, G., and Müller, K.-R., editors, *Neural Networks: Tricks of the Trade*, volume 7700 of *Lecture Notes in Computer Science*, pages 599–619. Springer Berlin Heidelberg.

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

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580.

Horn, D., Naftaly, U., and Intrator, N. (2012). Large ensemble averaging. In Montavon, G., Orr, G., and Müller, K.-R., editors, *Neural Networks: Tricks of the Trade*, volume 7700 of *Lecture Notes in Computer Science*, pages 131–137. Springer Berlin Heidelberg.

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):0090–95.

Hyvärinen, A. and Hoyer, P. (2000). Emergence of phase-and shift-invariant features by decomposition of natural images into independent feature subspaces. *Neural computation*, 12(7):1705–1720.

Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation. *Neural networks*, 1(4):295–307.

Jarrett, K., Kavukcuoglu, K., Ranzato, M., and Le Cun, Y. (2009). What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th International Conference on Computer Vision*, pages 2146–2153. IEEE.

Jhuang, H., Serre, T., Wolf, L., and Poggio, T. (2007). A biologically inspired system for action recognition. In *IEEE 11th International Conference on Computer Vision*, pages 1–8. IEEE.

Ji, S., Xu, W., Yang, M., and Yu, K. (2010). 3D convolutional neural networks for human action recognition. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 495–502.

Ji, S., Xu, W., Yang, M., and Yu, K. (2013). 3D convolutional neural networks for human action recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(1):221–231.

Jones, E., Oliphant, T., Peterson, P., et al. (2001–2014). SciPy: Open source scientific tools for Python. http://www.scipy.org/. [Online, accessed 11-November-2013].

Keys, R. (1981). Cubic convolution interpolation for digital image processing. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 29(6):1153–1160.

Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Master's thesis, University of Toronto, Computer Science Department. Chapter 4.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In Pereira, F., Burges, C., Bottou, L., and Weinberger, K., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

Kshirsagar, V., Baviskar, M., and Gaikwad, M. (2011). Face recognition using eigenfaces. In *2011 3rd International Conference on Computer Research and Development (ICCRD)*, volume 2, pages 302 –306.

Le, Q. V., Zou, W. Y., Yeung, S. Y., and Ng, A. Y. (2011). Learning hierarchical invariant spatio-temporal features for action recognition with independent subspace analysis. In *2011 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3361–3368. IEEE.

Le Cun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. E., and Jackel, L. D. (1990a). Handwritten digit recognition with a back-propagation network. In Touretzky, D., editor, *Advances in Neural Information Processing Systems 2*, pages 396–404. Morgan-Kaufmann.

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

Le Cun, Y., Denker, J. S., Solla, S. A., Howard, R. E., and Jackel, L. D. (1990b). Optimal brain damage. *Advances in neural information processing systems*, 2(1):1990.

Le Cun, Y., Huang, F. J., and Bottou, L. (2004). Learning methods for generic object recognition with invariance to pose and lighting. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition.*, volume 2, pages 97–104. IEEE.

Le Cun, Y. A., Bottou, L., Orr, G., and Müller, K.-R. (2012). Efficient backprop. In Montavon, G., Orr, G., and Müller, K.-R., editors, *Neural networks: Tricks of the trade*, volume 7700 of *Lecture Notes in Computer Science*, pages 9–48. Springer Berlin Heidelberg.

LeCun, Y., Jackel, L. D., Bottou, L., Brunot, A., Cortes, C., Denker, J. S., Drucker, H., Guyon, I., Muller, U. A., Sackinger, E., Simard, P., and Vapnik, V. (1995). Comparison of learning algorithms for handwritten digit recognition. In Fogelman, F. and Gallinari, P., editors, *International Conference on Artificial Neural Networks*, pages 53–60, Paris. EC2 & Cie.

Lee, H., Pham, P., Largman, Y., and Ng, A. Y. (2009). Unsupervised feature learning for audio classification using convolutional deep belief networks. In Bengio, Y., Schuurmans, D., Lafferty, J., Williams, C., and Culotta, A., editors, *Advances in neural information processing systems*, pages 1096–1104. Curran Associates, Inc.

LeNet tutorial (2013). Convolutional neural networks (LeNet). http://deeplearning.net/tutorial/lenet.html. [Online, accessed 21-10-2013].

Leverington, D. (2009). A basic introduction to feedforward backpropagation neural networks. http://www.webpages.ttu.edu/dleverin/neural_network/neural_networks.html. [Online, accessed: 2014-01-29].

Linusson, H. (2013). Multi-output random forests. Master's thesis, University of Borås/School of Business and IT.

Lowe, D. G. (1999). Object recognition from local scale-invariant features. In *The proceedings of the seventh IEEE international conference on Computer vision, 1999.*, volume 2, pages 1150–1157. IEEE.

Mnih, V. and Hinton, G. E. (2010). Learning to detect roads in high-resolution aerial images. In Daniilidis, K., Maragos, P., and Paragios, N., editors, *Computer Vision–ECCV 2010*, volume 6316 of *Lecture Notes in Computer Science*, pages 210–223. Springer Berlin Heidelberg.

Mukherjee, D. P., Potapovich, Y., Levner, I., and Zhang, H. (2009). Ore image segmentation by learning image and shape features. *Pattern Recognition Letters*, 30(6):615–622.

Naftaly, U., Intrator, N., and Horn, D. (1997). Optimal ensemble averaging of neural networks. *Network: Computation in Neural Systems*, 8(3):283–296.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, volume 2011, page 4.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with CUDA. *Queue*, 6(2):40–53.

Nievas, E. B., Suarez, O. D., García, G. B., and Sukthankar, R. (2011). Violence detection in video using computer vision techniques. In Real, P., Diaz-Pernil, D., Molina-Abril, H., Berciano, A., and Kropatsch, W., editors, *Computer Analysis of Images and Patterns*, volume 6855 of *Lecture Notes in Computer Science*, pages 332–339. Springer Berlin Heidelberg.

Oliphant, T. E. (2007). Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20.

Rougier, C., Meunier, J., St-Arnaud, A., and Rousseau, J. (2011). Robust video surveillance for fall detection based on human shape deformation. *IEEE Transactions on Circuits and Systems for Video Technology*, 21(5):611–622.

Roux, N. L. and Fitzgibbon, A. W. (2010). A fast natural Newton method. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 623–630.

Scharr, H. (2000). *Optimal Operators in Digital Image Processing*. PhD thesis, Ruperto Carola University, Heidelberg, Germany.

Schaul, T. and Le Cun, Y. (2013). Adaptive learning rates and parallelization for stochastic, sparse, non-smooth gradients. *CoRR*, abs/1301.3764.

Schaul, T., Zhang, S., and Le Cun, Y. (2013). No more pesky learning rates. In *Proceedings of The 30th International Conference on Machine Learning*, pages 343–351.

Schindler, K. and Van Gool, L. (2008). Action snippets: How many frames does human action recognition require? In *IEEE Conference on Computer Vision and Pattern Recognition, 2008.*, pages 1–8. IEEE.

Schmidhuber, J. (2014). Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828.

Schraudolph, N. N., Yu, J., and Günter, S. (2007). A stochastic quasi-newton method for online convex optimization. In Meila, M. and Shen, X., editors, *Proceedings of 11th International Conference on Artificial Intelligence and Statistics*, volume 2, pages 436–443, San Juan, Puerto Rico. JMLR.

Schuldt, C., Laptev, I., and Caputo, B. (2004). Recognizing human actions: a local SVM approach. In *Proceedings of the 17th International Conference on Pattern Recognition, 2004.*, volume 3, pages 32–36. IEEE.

Segal, M. R. (2004). Machine learning benchmarks and random forest regression. Technical report, Center for Bioinformatics & Molecular Biostatistics, University of California, San Francisco, CA, USA.

Senior, A., Heigold, G., Ranzato, M., and Yang, K. (2013). An empirical study of learning rates in deep neural networks for speech recognition. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6724–6728. IEEE.

Simard, P. Y., Steinkraus, D., and Platt, J. C. (2003). Best practices for convolutional neural networks applied to visual document analysis. In *Proceedings of the seventh international conference on document analysis and recognition*, volume 2, pages 958–962.

Smeaton, A. F., Over, P., and Kraaij, W. (2006). Evaluation campaigns and TRECVid. In *Proceedings of the 8th ACM international workshop on Multimedia information retrieval*, pages 321–330. ACM.

Socher, R., Lin, C. C., Manning, C., and Ng, A. Y. (2011). Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 129–136.

Sutton, R. S. (1992). Adapting bias by gradient descent: An incremental version of Delta-Bar-Delta. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 171–176. AAAI Press.

Tibshirani, R. (1996). Bias, Variance, and Prediction Error for Classification Rules. Technical report, Department of Statistics, University of Toronto.

Turk, M. and Pentland, A. (1991). Face recognition using eigenfaces. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 1991.*, pages 586–591.

van der Walt, S., Schönberger, J. L., Nunez-Iglesias, J., Boulogne, F., Warner, J. D., Yager, N., Gouillart, E., and Yu, T. (2014). scikit-image: Image processing in Python. Technical report, PeerJ PrePrints.

Wagner, A., Wright, J., Ganesh, A., Zhou, Z., Mobahi, H., and Ma, Y. (2012). Toward a practical face recognition system: Robust alignment and illumination by sparse representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(2):372–386.

Wan, L., Zeiler, M., Zhang, S., Cun, Y. L., and Fergus, R. (2013). Regularization of neural networks using DropConnect. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1058–1066.

Wang, H., Ullah, M. M., Klaser, A., Laptev, I., Schmid, C., et al. (2009). Evaluation of local spatio-temporal features for action recognition. In *BMVC 2009-British Machine Vision Conference*.

Weinland, D., Ronfard, R., and Boyer, E. (2011). A survey of vision-based methods for action representation, segmentation and recognition. *Computer Vision and Image Understanding*, 115(2):224–241.

Willett, K. W., Lintott, C. J., Bamford, S. P., Masters, K. L., Simmons, B. D., Casteels, K. R., Edmondson, E. M., Fortson, L. F., Kaviraj, S., Keel, W. C., et al. (2013). Galaxy Zoo 2: Detailed morphological classifications for 304 122 galaxies from the Sloan Digital Sky Survey. *Monthly Notices of the Royal Astronomical Society*, 435(4):2835–2860.

Xu, W. (2011). Towards optimal one pass large scale learning with averaged stochastic gradient descent. *CoRR*, abs/1107.2490.

Yang, M., Ji, S., Xu, W., Wang, J., Lv, F., Yu, K., Gong, Y., Dikmen, M., Lin, D. J., and Huang, T. S. (2009). Detecting human actions in surveillance videos. In *Proceedings of the TrecVID Video Evaluation Workshop*.

Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In Fleet, D., Pajdla, T., Schiele, B., and Tuytelaars, T., editors, *Computer Vision – ECCV 2014*, volume 8689 of *Lecture Notes in Computer Science*, pages 818–833. Springer International Publishing.