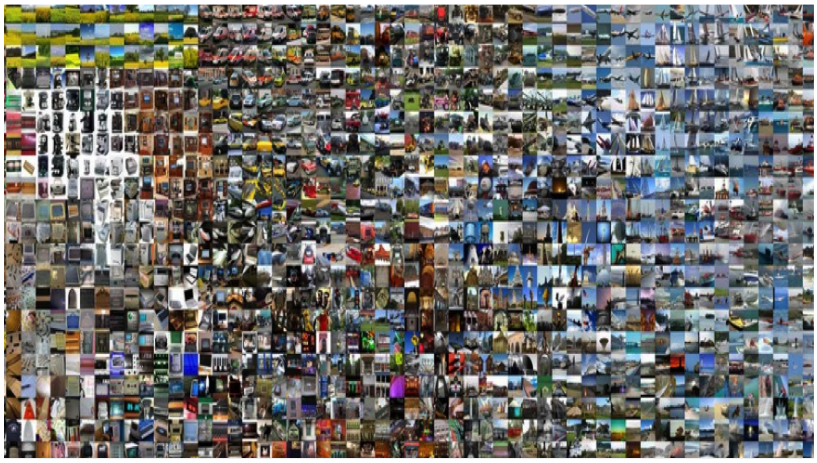
Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2016



Project Title:	**Deep Learning with Computer Vision**
Student:	**Hemang Rishi**
CID:	**00731747**
Course:	**4T**
Project Supervisor:	**Dr. T K Kim**
Second Marker:	**Dr D.F.M. Goodman**

Abstract

There has been a rise of using deep learning techniques to classify medium to high scale resolution objects on multiple state-of-the-art hardware. This project investigates research towards the opposite direction by designing an implementing deep learning techniques to classify low resolution images in limited hardware. It has uniquely adapted award-winning architectures for low scale use and produced a top-5 validation accuracy of 76%. It has also discovered the key parameters that optimize the models using a Tiny ImageNet data set in limited hardware and has made a step towards driving research to implement deep learning techniques on embedded systems.

Acknowledgements

I would like to thank Dr Rigas Kouskouridas and Dr T K Kim for their unconditional support and Ms. Esther Perea for her guidance.

Contents

Chapter 1

Introduction

1.1 Motivation

One of the first things a young human learns is how to recognize an object [1]. It becomes a almost natural process that occurs without any hesitation. However, computer vision algorithms have struggled to develop a system that is capable of recognizing objects with such ease in the real world. No robust solution exists despite the fact that that object recognition has been studied for many years. There are many reasons that account for this but the two most notable ones are: firstly, there are large numbers of objects in the world and secondly there are many external factors that increase the difficulty to classify these images [2]. These factors include the lighting of objects in a scene, object background clutter, occlusion (preventing the full image from being seen) and even colour invariance.

Figure 1.1: *A selection of challenges for Computer Vision systems to recognise objects: occlusion, lighting clutter of espresso cups [3]*

If a reliable object detection system could be designed, it would open the door for many developments such as image search algorithms, image retrieval and image reconstructions. All of such developments will have a use in numerous applications ranging from autonomous cars to aiding visually impaired individuals gain a better understanding of their environment [4]. Hence, there is still a very strong motivation to find a robust solution.

Traditional classification techniques use a time-labouring approach of hand-crafting features which are fed through linear classifiers that can be trained. However, they do not scale well to the larger representations of the visual world due to the invariances described [2]. Hence, a robust solution should involve learning and extracting features autonomously to produce a strong visual representation [5].

Recently, there have been attempted solutions aided by the recent advances of hardware - making learning more complicated classifiers much quicker [6]. These solutions, much like our brain, use a hierarchical approach which classifies images using multiple trainable layers that are initiated by analysing low level edge features and concluded by analysing high level abstractions [5]. They are deep learning architectures.

For vision, a specific set of deep architectures have been used called Convolutional Neural Networks (CNNs). They have found to have had much success in the recent ImageNet challenge [7]. The goal of the challenge is to take a subset of data from the real world and classify over a million images (with view point variation) into a thousand classes. In recent years, CNNs have been used to reduce the top-5 error rate to just 3.52% [8]surpassing human recognition accuracy [9]. However, not only has state of the art hardware been used [10], but medium to high scale resolution (256x256) images have been classified [7] over a training time of three to four weeks. These resolutions allow relevant regions of the image to be large enough to classify. In practice, these image resolutions are much lower: the interest regions are much smaller and thus may inhibit classification [11]. For instance, lots of video surveillance is captured using low resolution cameras (cost reasons) and there has been much work in trying to identify faces from these images with very small interest regions (16x16) [11]. Additionally, if CNNs were to ever be used in daily practice, they would be required to be implemented (and trained) on limited hardware such as embedded systems [12]. These systems, due to their limited computational resources, would be prevented from capturing or using high and medium resolution images for classification [13].

There has been recent research to use CNNs on limited (in terms of memory and speed) hardware given the numerous applications. [14] attempted to use FPGAs to implement a CNN architecture. [15] endeavoured to use embedded systems for facial recognition. [16] tried to optimise CNN's implementation on GPUs (Graphical Processing Units) of a mobile phone. Finally, over the last few months, an attempt has been made to train a CNN on a embeded GPUs using low precision numbers. [17]. There is clear trend that attempts are being made to make CNNs more accessible and using low resolution images will help in this pursuit.

Unitil recently, the problem of using low resolution images for classification has been largely overlooked. There have been endeavours, within the last year, to perform object classification on fine-grained (intra class lassifciation) data sets using low resolution images by [18] and [13]. However, there is still much to be done to make CNNs used further in daily life.

1.2 Aims and Objectives

The aim of this project is to explore object recognition using novel classification techniques found in the field of machine learning: deep neural networks or "deep learning". The advantage of using such techniques is the ability to learn object categories through training or 'experience': a technique which attempts to replicate part of the processes our brain undertakes to recognize objects.

More specifically, 200 classes of low resolution (64x64) images containing clutter, invariances and other real world challenges will classified. The data-set will include both coarse (differentiation between

animals and objects) and fine-grained (types of species of butterfly) classes. This makes the challenge more difficult that the previously solved CIFAR-10 (classifying 32x32 pixels into 10 classes).

Due to the fact that there is a drive to implement CNNs on lower memory systems, such as an embedded system [19], the project will aim to classify these objects on lower memory hardware than the one used by the Imagenet winners. This project aims not to use an embedded system but perform a small step towards lower memory classification by using a GPU a third of the memory size of the ImageNet winners. Additionally, this project is not focussed on optimising low level software but rather aimed at identifying the characteristics of the architectures that give the highest test accuracy for lower resolution images while using the lowest memory. For the sake of testing a number of different architectures, the network has to be trained under a pre-defined time.

By setting up the project in this manner, several questions arise which this project will attempt to answer:

- Can a CNN classify low resolution images efficiently in a relatively short time period with limited memory?
- If so, which architecture gives the greatest accuracies?
- What meaningful insights can be drawn by adjusting architectures and hyper-parameters for the limited hardware?

Chapter 2

Background

The aim of this section is to form a deeper understanding of the project specifications and evaluate prior work.

It will begin by considering what is meant by deep architectures and why use Convolution Neural Networks . Following this, current work with deep architectures and computer vision will be explored in order to understand the progress in this field. Finally, existing work on classification using the larger ImageNet database will be analyzed. This should allow for a careful formation of a design for implementation.

2.1 Why use Deep Architectures?

To have an intuitive understanding, this part will examine how existing classification works followed by the explanation of neural networks and finally how all of this links to explain why the deep architecture of the convolution neural networks is needed.

2.1.1 The Traditional Machine Learning Approach

A computer processes an image by examining its pixels. These pixels are represented by their intensities: a number ranging from 0 to 255 (an eight-bit number). For example, consider a black and white image. If a pixel is closer to the white spectrum it will be represented by a number close to 255 and likewise, a pixel closer to the black spectrum will be represented by a number close to 0. This analogy can be extended to colour images which have three channels each representing red, green and blue colours. Here, there are three numbers corresponding to three intensities [20].

Existing classification techniques aim to use these pixels to identify different characteristics of the image. For example, they may identify where the contours of an object lie in an image. However, they require some training data (many images) to perform this categorization.

Usually simple categorization models aim to learn parameters to construct a linear classifier. Consider an example where the aim is to classify images into ten different categories (cats, dogs etc.). Here, the method is to assign weights as the parameter, W, which compares each of the pixels in the image to each category, labelled x. This comparison occurs by multiplying the weights and the inputs

(with bias term, b). The output matrix can be labelled as a score and usually the highest number of the score is the category the image can belong to. However, the features to identify the categories have to be hand-picked (also known as supervised learning) and are linearly separable.

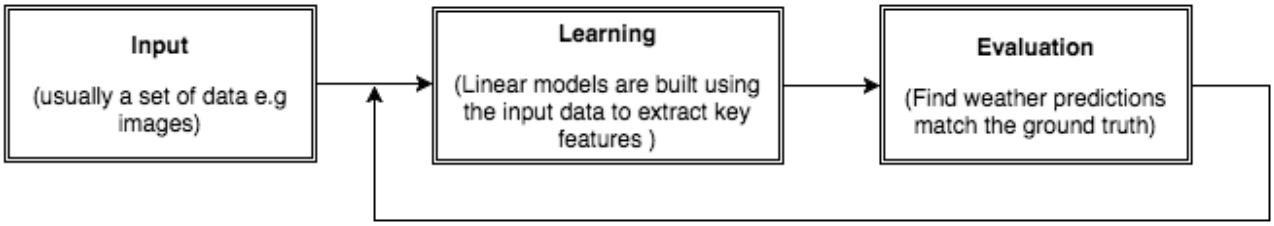


Figure 2.1: *The process of training a linear classify model*

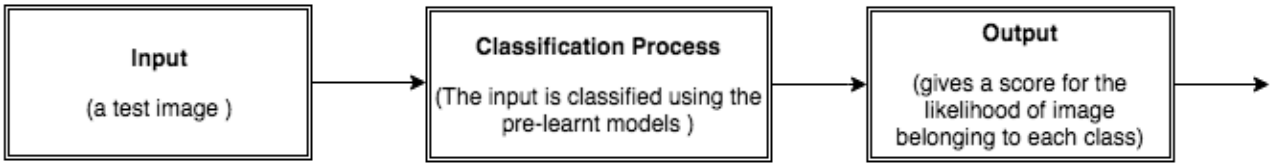


Figure 2.2: *The process of test a linear classifier model*

Training the classifier is an iterative process. In figure 2.1 this can be seen after the evaluation phase. After the discrepancy between the prediction (output of the score function) and ground truth is calculated, the parameters of the model are adjusted to minimize this error. As the error can be represented by a cost/loss function, optimization methods, such as the gradient descent method, are used for efficient minimization.

Gradient descent is commonly utilised and it specifically analyses the error function with respect to the weights. However this is done after every iteration of the training pipeline and so forms an iterative set of linear equations:

$$x_{n+1} = x_n + \alpha\delta \tag{2.1}$$

The core condition is that x_{n+1} must be less than x_n . This occurs due to the fact that , the direction, and α, the learning rate, must produce a descent direction towards the minimum. However, α must not be too large as it may miss the minima or too small as it will take a long time to converge. This is one of the most common methods to tune the weights for particular classification techniques.

Unfortunately, traditional linear classification techniques are considered too weak to accurately account for intra-class variations (differences types of cars) [21]. This is due to the fact there are many complex behaviours that vary from image to image such as the way light falls, different orientations and possible other objects. These are usually called factors of variation as they can vary separately and independently. To model these complex behaviours a highly varying mathematical function is required that can display a large number of variations and allow for separation of key features [22].

Hence there is a need for tunable non-linear classification techniques – known as neural networks.

2.1.2 Neural Networks

The way the brain processes information has been the inspiration behind neural networks. Computations in our brain are performed by 86 billion neurons of them all interconnected by synapses [23]. Dendrites feed information into the neuron (like an input). The computation is performed in the cell body (learning) and if activated it sends a signal to other neurons through the axon. Although this is a very basic description of a neuron's function, a similar architecture is used in conventional neural networks and it allows for non-linear classifiers to be trained.

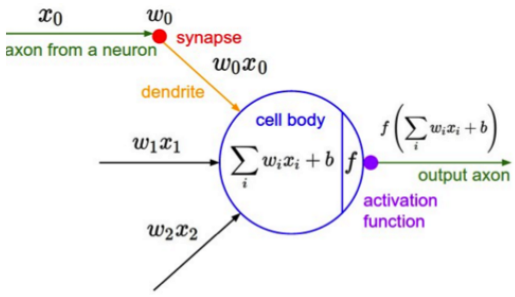


Figure 2.3: *showing two filters for vertical and horizontal lines and two resulting activation maps [21]*

As seen in 2.3, a single neuron uses a similar weighting technique seen in linear classifiers and uses an activation function that determines its binary representation of 1 or 0. Activation functions are the source of the non-linearity of a neuron and can vary depending on application but this will be further examined when discussing Convolutional Neural Networks.

The main differences between linear classifiers becomes more apparent when considering the neural network as a whole – the architecture:

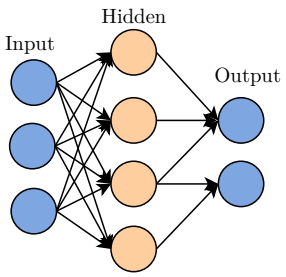


Figure 2.4: note the brown circles represent neurons and black lines are analogous to synapses

Each neuron is placed into layers and that are interconnected. The neurons are not intra-connected to each other. This allows for the convenient use of matrices when computations are fed forward. The layers between the input and output are classed as "hidden" and given the active functions will be non-linear - the output will be non-linear functions of the input. Subsequently, neural network can build a model that can adequately fit the statistical properties of the data.

The amount of hidden layers can vary. The more there are, the greater the number of representable functions. Deep architectures will have many hidden layers (10-20) [22] but shallower architectures will have much less. Additionally, the number of parameters class the size of a neural network. The more parameters, the more computation is required to tune them. They are optimally adjusted by similar, but more complex, iterations than in the simple gradient descent. A process called backpropogation [24] is used. Backpropagation aims to examine the local gradient at each layer by starting with the gradient of the output. The aim is to work backwards with the computations to understand to which layer the output is most sensitive and perform gradient descent to minimize the loss at that neuron:

$$\frac{\delta \mathrm{J}}{\delta \mathrm{W}} = (\frac{\delta(\frac{1}{2}(\mathrm{y_t} - \mathrm{y_o})^2}{\delta \mathrm{W}}) \tag{2.2}$$

Note J=cost function,W=the weights, $\mathrm{y_t}$=traning set data, $\mathrm{y_o}$=output of neural network

$$\frac{\delta \mathrm{J}}{\delta \mathrm{W}} = (\mathrm{y_t} - \mathrm{y_o})(\frac{\delta \mathrm{y_o}}{\delta \mathrm{W}}) \tag{2.3}$$

Here $\frac{\delta \mathrm{y_o}}{\delta \mathrm{W}}$ is a function of the output of the previous hidden layer before hence the equation becomes:

$$\frac{\delta \mathrm{J}}{\delta \mathrm{W}} = (\mathrm{y_t} - \mathrm{y_o})(\mathrm{z}^3)(\frac{\delta \mathrm{z}(3)}{\delta \mathrm{W}}) \tag{2.4}$$

$\mathrm{g}(\mathrm{z}^3)$= Output of activation function of previous layer, $\mathrm{z}^3 = \mathrm{W}(\mathrm{g}(\mathrm{z}^2))$

The equation expands until the first layer to discover the loss of the network. The aim is to minimise this function to reduce loss. However, there is an increasing number of computations for greater amount of hidden layers.

Furthermore, augmenting the number of layers will allow features to be further finely separated on the training set but there is a risk of over tuning the parameters (overfitting): it may give accurate results for a training set but may not so accurate on a test image. As a result, when overfitting occurs the neural network does not generalize well.

Thus neural networks use techniques such as regularization to minimize this risk. This involves reducing some weighting of parameters. Specifically, there is a penalization term added to the loss function to implement this:

$$\begin{aligned} \mathrm{J} &= \frac{1}{2\mathrm{m}}[\sum_{\mathrm{i}=1}^{\mathrm{m}}(\mathrm{y_t^i} - \mathrm{y_o^i})^2 + \lambda \mathrm{R}(\mathrm{W})] \\ \mathrm{R}(\mathrm{W}) &= \sum_{\mathrm{j}=1}^{\mathrm{n}} \sum_{\mathrm{l}=1}^{\mathrm{m}} (\mathrm{W_{j,l}^2}) \end{aligned} \tag{2.5}$$

Here the regualrization term is the square of the weights but can take different forms. This is investigated further when designing the system in section 4. Notice though the scalar multiple . This is another parameter needed to be tuned by the network. A common term to describe these parameters, that need to be adjusted during the time of training, is called hyper-parameters. Looking at the architecture as a whole, there are an arbitrary number associated with a neuron and so if deep architectures were to have many layers, the computation of tuning these parameters will become very

large. In addition, the number of inputs associated with a pictorial image is much larger than other applications. A 64x64 image can have up to 4096 inputs. This also increases the number of parameters needed. Subsequently, this augments not only the computational power but also the amount of memory required. Although neural networks were first formed in the 1980s, deep architectures were not considered until the late 1990s. LeCun, a pioneer in deep learning, introduced a variation of neural networks that alleviated that problem [25]: a Convolutional Neural Network (CNN).

A CNN is required to be a deep architecture. The main explaination behind this is found by considering how images need to be processed by an algorithm . To classify real world images, an algorithm needs to handle the factors of variation. Then it needs to extract the key features and, just like our brain, use them to build a more abstract representation of the environment. For example when recognizing a face there are a number of steps that lead to greater level of abstraction: firstly, the lines contours of the face and the features need to be identified (first layer) and secondly, after this the local shapes need to be understood (second layer), thirdly, the global shape needs to be identified (layer 3) and eventually these then need to placed into a category of a nose, mouth and eyes to identify the category of a face. The combination of layers introduces a hierarchal structure needed to recognize an object and allow for the factors of variation to be dealt with. It is found the brain works in a similar manner [22] using its V1, V2, V3 neurons.

This is difficult to achieve with a shallow architecture (1-3 layers). Additionally it is important that the algorithm learns these features (these features should not be handpicked by the designer) as it is difficult to have prior knowledge of all the factors of variation in a large set of images to build this higher level of abstraction [22]. Subsequently, deeper architectures such as that of convolutional networks have led to string of advances in object recognition.

2.2 What is a CNN?

As mentioned before, a CNN is a specialized neural network that uses a deep architecture and assumes an image as an input. This allows the architecture of the neural network to be designed in a specialized manner to optimize performance and allow for it to be suitable to use in real world applications. It specifically optimizes and so reduces the number of free parameters that need to be trained. To understand how this is achieved, the architecture needs to be considered.

The architecture of a CNN has a few key features: firstly, it has 3 dimensions – width height and depth (not the depth of the neural network). Secondly, in a certain layer, neurons can only be connected to a small region of the previous layer. This will decrease the size of the output to a single vector of scores of the class (arranged along the depth dimension).

Thirdly, it contains three distinct types of layers: a convolution layer, a pooling layer and a fully connected layer. These layers will be stacked and adapted in different ways to build a convolutional network. It is important to examine the functions of these types of layers to understand how it reduces the number of parameters:

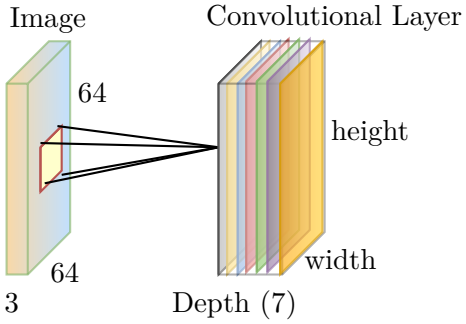


Figure 2.5: An example of a layer (convolutional) in a neural network. Note the increase in depth represents stacked filters

Convolutional Layer

This layer is recognized to be the core of a CNN and like all the other layers; it outputs a 3D vector (width x height x depth). The key differential is that the weights attached to the neurons are a set of learnable filters. These filters are then attached to a small spatial region along the width and height of the previous layer. The weights of this neuron are slid (convolved) across the width and height of the image. Taking a dot product of these weights with the image pixels forms the output which is then passed through an activation function which constructs activation (feature) maps. A collection of these maps is created each of which extracts and identifies a certain feature useful for the network as a whole to perform classification. These activation maps are further stacked along the depth dimension of the corresponding layer. A single increase in the depth dimension results in another activation map added.

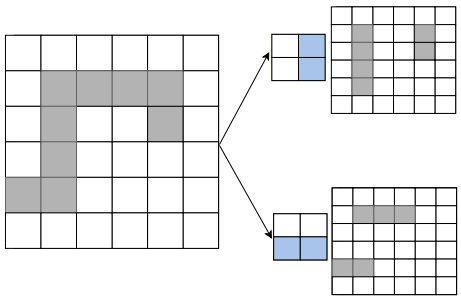


Figure 2.6: An example of a layer (convolutional) in a neural network. Note the increase in depth represents stacked filters

It is important to consider the types of activation function to generate an activation map. They are non-linear and they classify the output of the weights (if logistic) into a binary representation for the neurons (on or off). If a neuron is classed as “on”, some functions allow the neuron to take the value of its output. However, there is one particular activation function that has propelled CNNs forward: the Rectified Linear Unit function[26]. It does not saturate as easily as other activation functions as sigmoids and thus less likely to produce dead weights. The purpose of this section is to give a high level introduction into CNNs and hence the types of activation functions are further investigated in

the design section 4.

A convolutional layer requires several parameters which are associated with how it is connected to the previous layer. These help downsize the larger image into smaller activation maps:

Table 2.1: My caption

Parameter	What it controls
Receptive field (F)	Spatial area (width and height) of previous layer which convolution layer is connected (depth remains constant)
Depth Stride (S)	Controls the spacing of neurons connected to the input layer (may lead to receptive fields overlapping for low strides)
Zero Padding (P)	Pads the border of an input image within a convolutional network with a number of zeroes. Padded so that the spatial size of the output volume can be controlled.

The parameters are applied to a formula that determines the activation map of a convolutional layer:

$$\text{Spatial Size of Output} = \frac{\text{W} - \text{F} + \text{P}}{\text{S}} + 1 \tag{2.6}$$

Where W is the input volume, F is the receptive field, S is the stride and P is the zero padding layer. It is important to note that the spatial size of the output must be an integer and thus the amount of strides must be set correctly. This stems from the fact that the receptive fields have to cover exactly the whole image in a symmetric manner and the spacing is controlled by the stride. An example below demonstrates how this reduces the number of parameters:

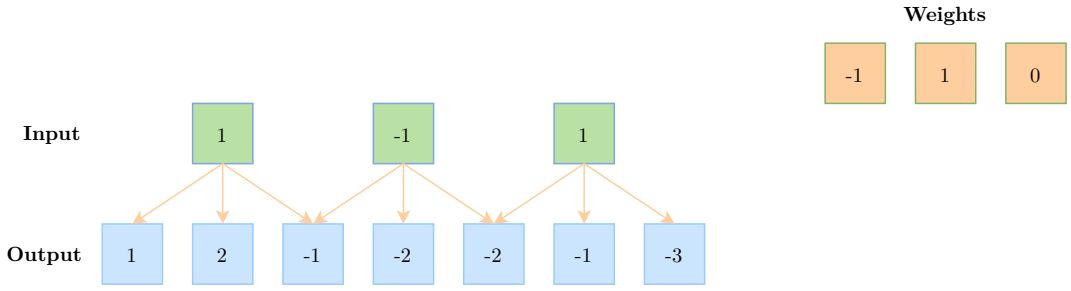


Figure 2.7: An example of size reduction using a convolutional layer. Here P=2, F=3, W=5 and S=2. Therefore the spatial output in the convolutional layer will be 3

The most essential matter to notice is that the weights are shared between each filter output. As a result only a single set of optimization needs to be performed for an activation map. This follows from the assumption that if a receptive field is placed on a particular spatial position on the input to compute and detect certain feature (to produce an activation map), it will be useful in different spatial positions across the image. For example, take a convolutional layer that has 300 weights spread across 300,000 neurons (typical amount for a depth image [21]). One would have 90,000,000 (300,000*300) parameters in the single convolutional layer. However if you use parameter sharing, each depth slice

on the CNN would only need a single parameter as it is performing a computation to identify a similar feature. Hence in the example, if the previous layer had a depth of 50, one would need simply (50*300 weights) 15,000 parameters spread across all the neurons (+50 bias terms). Therefore each neuron in the same depth slice will be sharing the same parameters. To incorporate this into the back-propagation algorithm, the sum of the gradients across each neuron would be taken from each depth slice returning a single weight per slice.

Pooling

This layer is usually inserted between convolution layers in a CNN. Its aim is to down sample each activation map. There are various techniques to do this such as using a MAX operation. Here it uses a filter of a certain size (pre-set parameter) - normally a 2x2 set of pixels. This size segment allows for appropriate downsizing without completely destroying the information contained in activation map. It finds the maximum intensity in that segment of four numbers and discards the rest: the width and height spatial size in reduced but depth is retained. Subsequently, this reduces the amount of parameters thereby the computation in the network. MAX pooling is found to be the most commonly used compared to the other techniques particularly in the larger Imagenet challenge [27].

Fully Connected Layer

This is the typical layer seen in neural networks where there is full set of pairwise connections to the previous layer. This is commonly seen in the penultimate layer of convolution networks. It is usually followed by a linear classifier (SVM or LogSoftMax) to interpret the class scores again this is something that is explored in detail in 4.

The layers and parameters described can be configured in different manners which can lead to varying performances. As of now, there is no clear way of assembling these layers together – the only essential condition is to construct multiple convolutional layers (followed by activation functions) with occasional pooling layers and a fully connected layer combined with a possible SoftMax or SVM function at the end used to classify the scores produced from the deep architecture. The number of layers is dependent on application. Additionally the parameters in the convolutional layers and the fully connected layers are the only ones needed to be trained as the pooling layers and activation functions remain fixed. However, the architecture of the model is found to impact the validation accuracies rather than the fine tune of hyper-parameters.

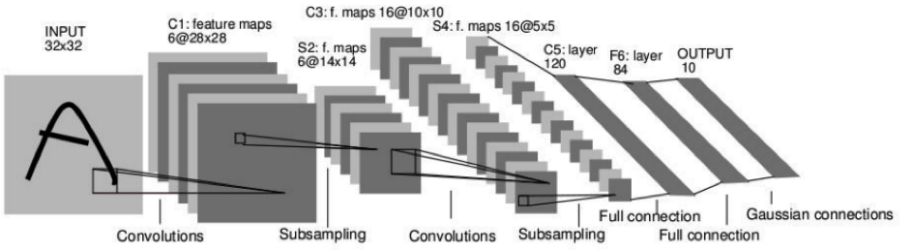


Figure 2.8: *One of the first architectures developed by LeCunn in 1998 [25]*

2.3 Computer Vision applications using CNNs

CNNs have made many image classification applications feasible and have sparked recent advances by the use of deep architectures in computer vision: particularly in the cases where traditional machine learning approaches have reached their limits. Such instances can be seen in object detection, hand tracking, face tracking and action recognition.

Hand tracking

The main problems of tracking hands include the number of degrees of freedom it contains, the number of orientations or positions a hand can hold and self-occlusions. Additionally, many hand gestures are naturally rapid in their movements [28]. This makes the initialization of the hand gestures important as its needed to control the variables involved. Existing classification techniques use two main methods: the discriminative and generative [29]. The discriminative methods can identify hand poses through classification techniques and can deal with rapid motions. However, it cannot handle environments with many variables nor the high dimensionality of hand gestures. This method uses a "K-th nearest neighbour" (comparison of one image to the set of training images) and therefore is limited by the number of gestures as it will be too computationally expensive to compare and capture all the variations the hand poses. A generative method presents the opposite problem to discriminative methods. It can handle the number of degrees of freedom the hand possess but is slow. Also, both methods find it difficult to classify hidden joints (not seen by the camera). This is where deep architectures are useful. A CNN can be used to combine these two models. It does not need hand tuned features and has recently been found to deliver state-of-the-art performances [29]- especially in detection of the hidden joints. It also helps to find gestures of a hand's non-rigid movements. It is found to perform better than shallower architectures hence amplifying the need of deeper ones.

Object Detection

State-of-the-art systems using machine learning models have propelled the object recognition field. One such example is with Deformable Part-based Model (DPM) [30]. It decomposes objects into graphical representations and builds part-based models by learning and extracting the key feature of the object (discriminative learning). However, the training data for this shallow architecture requires manual

feature decisions to produce robust recognition. Thus deeper architectures not only eliminate the need of these hand-designed features but can learn more complex models due to their large hierarchical representation. This is particularly noticed when applying them to images in the ImageNet Challenge. For the aforementioned reasons, here a CNN is the preferred architecture. However, CNNs tend to overfit the training data and thus there is an increasing need for regularization. As discussed in section 4, techniques have been developed that adds more bias to the data and allow for successful implementation.

Action Recognition

A key area of interest in recent research is the automatic understanding of how humans interact with different environments. To achieve such comprehension, human action recognition is pivotal. There have been attempts with traditional classification techniques that particularly aim at combining motion data with texture descriptors. These descriptors, such as Harris-3D [31] and Cuboid detector [32], look at the texture around manually engineered spatial-temporal points. They are found to have high performance if carefully engineered (finely tuned) for a specific context space but do not generalize well for different contexts. Again, the advantage of deep architectures is that such features do not need to be hand crafted and the hierarchical representation can handle more general representations. However, CNNs are designed to extract features form static images and therefore spatio-temporal features, which change over time, will not be taken into account. In [33], they successfully attempted to make the inputs into the convolutional neural networks 3D and combine it with a shallow Recurrent Neural Network (RNN) (such as Long Short Term Memory) to classify the output. The RNNs classify tasks that involve short time lags and so are apt for such purpose. Thus this shows the adaptability of deep architectures and in particular CNNs.

2.3.1 Face Recognition

Facial Recognition is a prominent research field in computer vision. Local regions of a facial image can be analyzed by traditional image descriptors (SIFT,HOG) followed by aggregation through some sort of pooling mechanism. Again, such descriptors need to be crafted from hand-designed features and they do not generalize well for data sets with 1.2 million images. Additionally, traditional classifiers cannot tune non-linear functions, needed to classify images, as well as a CNN. Prior work includes prominent attempts such as DeepFace [34]. It inputs pairs of faces to a CNN and computes the Euclidean distance between the two. Therefore when it is trained, the goal is to minimize the euclidean distance between faces displaying the same identity and maximize the distance for faces displaying different identities. In general their whole architecture used a collection of CNNs to classify the image. Such method successfully classifies 4 million images with the 4000 identities provided. Recently, Google [35] attempted to extend such technique to classify 200 million faces but used a set of three faces instead of two: where two of them would be similar identity and the third not similar. The goal was to make the distance between two similar faces smaller than that of the third “pivot” face. Having applied such technique to a number of CNNs, it was found to have one of the best performances to

date on large public data sets (LFW and LTF). This shows that convolutional neural networks can be grouped together to find the difference and similarities between many images.

2.3.2 Past attempts at the larger scale Imagenet challenge

Over the last couple of years there has been much work on the larger scale ImageNet challenge. Since these algorithms have classified higher -resolution images with such success, it was though of as a good starting point.

In 2012, Krizhevsky designed a deep architecture (AlexNet) using a CNN to provide a robust solution and error rates lower than ever seen before: the correct class was predicted 84.2% of the time within the first five predictions thereby producing a top 5 of 15.3% [6]. It has sparked more attempts: In 2014 two further prominent architectures were developed – the VGG network and Googlenet. They reduced the top-5 error rates to 7.5% [27] and 5.7% [36] respectively. In the tail-end of 2015, Microsoft produced an architecture which reduced the top-5 error rate to 3.57% - the best yet [8]. The architectures of these four main attempts will be discussed with a view of providing possible inspiration for adaptation and implementation for the lower resolution images.

Alexnet

In 2012 Krizhevsky and his fellow colleagues were the first to use a convolutional neural network architecture to solve this challenge. They produced the lowest ever error rates (in 2012). Hence their architecture has been the basis of many solutions of this challenge.

It was an extremely deep network of 60 million parameters and 650,000 neurons contained in 5 convolutional layers, 3 fully connected layers followed by a 1000-way Softmax layer at the end [6]. It used Regularized Linear Unit (ReLu) activation functions, which do not suffer from saturation, and so the input need not be normalized. However, from experimentation they found that normalization produced better generalization even with non-linearised activation functions. They used a specialised normalizing technique called “brightness optimization” which was different from the commonly used mean normalization. Additionally, the other elements of pre-processing conducted on the input including taking a centralized patch of size 224x224.

The biggest problem they faced with such a large architecture was overfitting. They used several techniques to combat this. Firstly, they used data augmentation. This included methods such as PCA (principle component analysis) (which reduces the number of dimensions in a data set), artificially enlarging the data set and overlap pooling. The data set was increased by a factor of 2048 by randomly cropping and flipping several images while overlap pooling was achieved by overlapping the regions of the sub-samples.

Secondly, they used a regularization technique that has become very popular: dropout. Dropout is a probabilistic method that is used to keep certain neurons active. Once these neurons have dropped out, the outputs do not contribute to the forward-pass of the network nor are they considered in the backward propagation. As a result, every time an image is trained on a CNN, a different architecture of that CNN is sampled. This allows neurons to develop independence as it does inter-depend on other neurons’ functions to recognize a feature. This increases the robustness of the large network.

Recent hardware advances made such use of the CNN possible. They were able to use high power Graphical Processing Units (GPUs). Two 3GB GPUs were used by carefully splitting the CNN in two and parallel computations. They argued they could possibly achieved a lower error rate with a larger network but were limited by the number of GPUs and the time to train the data (two-three weeks training time).

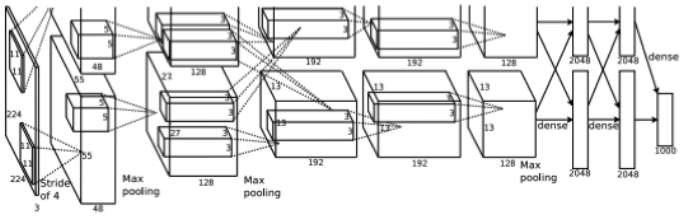
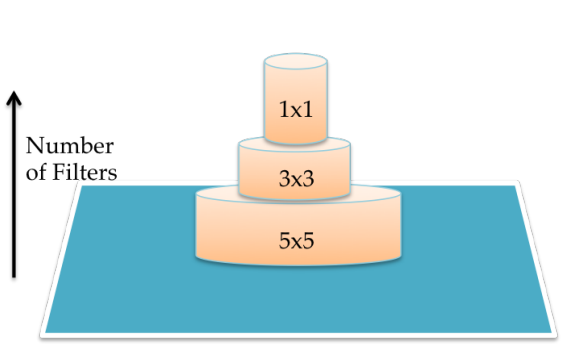


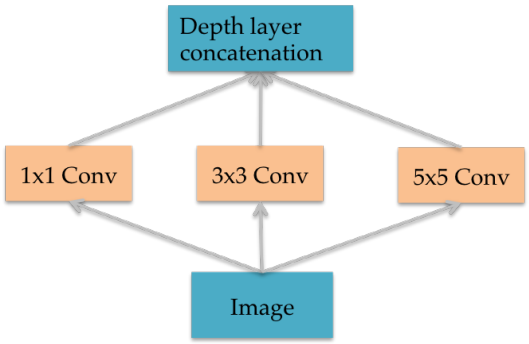
Figure 2.9: *the 2012 ImageNet winning architecture split into two [6]*

GoogLeNet

Two years later, the GoogLeNet architecture won the ImageNet challenge. It presented two key areas of development from the original Alexnet. Firstly, it utilized the idea that images can be classified by analyzing its correlation statistics. Specifically, this idea originated from a theory found by Arora et. al [37]. The theory states that each layer can be formed from the correlation statistics of the previous layer. This idea can be applied to images by taking correlations that tend to be in clusters, preforming convolution and aggregating them. The clusters can be local (1x1 pixel) or more spread out (3x3 pixels, 5x5 pixels). This process was contained in an inception module:



(a) *The clusters of correlations statistics extracted by local convolutions*



(b) *An Inception module represents the aggregation of these statistics*

Figure 2.10: *Breaking down the inception module: diagram left shows how the clusters are extracted from the image and on the right how this would be represented in a neural network*

However, it was found that this increased the dimensions and so created a computational bottleneck. Therefore, 1x1 convolutions were applied before the 3x3 and 5x5 convolution layers. This idea created a mini convolution network ("network in a network" (NIN)) architecture and was inspired by [38]. The purpose of this preceding layer was two fold. Firstly, the 1x1 convolution layer acted like a

max pool layer but for the depth dimension. This made it possible to reduce the dimensionality to any pre-specified value and therefore reduce the computational complexity that otherwise may limit the depth of the network. For example, the 5x5 convolution layers were the biggest factor in the increase in computational complexity. By using a proceeding 1x1 [36] convolutional layer the dimensionality could be reduced to appropriate levels whilst still maintaining the original resolution of the image. Secondly, it added another layer of non-linearity before another non-linear function – thereby allowing for more complex classification. This inception module thus allowed for localization of objects in a region of the image and then permitted classification using a local CNN. Subsequently, this technique presented a dramatic change from the conventional stacking of depth layers common in CNNs. The second key development was the reduction in the number of parameters. Usually increasing the depth led to a proportional increase in this number but this network but this model was the opposite: It reduced the number of parameters by nearly a factor of 12 (to 5 million) compared to the AlexNet model and, at the same time, was found to present a higher top-5 accuracy [36]. It also comprised of 27 layers of which 22 are trainable but no fully connected layer at the end. Instead, it used spatial average pooling to reduce the number of dimensions before the negative loss function. This was the main reason why the parameters were kept to a minimum. Hence given the memory limitations described in project brief, this may prove to be a useful architecture to adapt and implement. Additionally, the inception model may lead to some results which may help localize the objects in the very low resolution images.

VGG Net

VGG Net was the runner up of the 2014 ImageNet challenge. It focussed on using smaller receptive fields (3x3) in the convolution filters but used increased amount of depth to compensate. The architecture showed significant improvement as demonstrated in the lower top-5 error rate of 7.3%.

Once again, it replicated much of AlexNet's pre-processing techniques by cropping the image to 224x224. However, the mean value of each RGB component was subtracted from the training images. Also, similar random cropping for augmentation was used. It was then entered into the architecture which contained 3x3 convolution layers. The receptive field size was chosen as it was believed to be the smallest size to identify directional features in the image (left/right, up/down and centre). In between certain layers of the 3x3 convolution layers there are 2x2 layers of max-pooling. Furthermore at the end of the architecture there are two fully connected layers that reduce the spatial size to a one-dimensional vector representing the number of classes of images. The final classes are classified by a negative log-likelihood function. At test time though, they used slightly different techniques: they experimented with scaling jitter (adjusting the size of images as long as the shortest sides were above 224). This was found to increase their accuracy by 0.2%. Furthermore, they used a form of multi-crop evaluation (ten crops see **??**) to form an average of accuracies of the test images which further increased accuracies by 0.4

They tested several models with such architecture but varied the number of 3x3 convolutions. Specifically they varied the depth from 11 to 19 layers. In addition, they kept the number feature maps limited to 64,128,256,512 which increased in order with respect to the depth of the network as

seen in figure **??**.

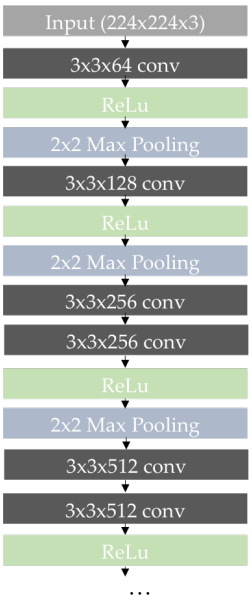


Figure 2.11: *A sub-sample of the VGG net*

Note The key advantage of this model is that the architecture can become extremely deep without exceeding the number of parameters of a shallow network with a larger receptive field and still produce high validation accuracies. The reason for this can be accounted by the fact that every 7x7 is equivalent to three 3x3 layers (appendix) each with three non-linear rectification layers which allows network to increase its discrimination.

$$\begin{array}{cc} \textbf{Number of weights for 7x7 filter} & \textbf{Number of weights for 3x3 filter} \\ \mathrm{C} \times (7 \times 7 \times \mathrm{C}) = 49\mathrm{C}^2 & 3 \times \mathrm{C} \times (3 \times 3 \times \mathrm{C}) = 27\mathrm{C}^2 \end{array} \tag{2.7}$$

Additionally, as noted previously in equation 4.15, there are fewer parameters making it more computationally efficient. It is important to consider the hardware used to implement the above architecture. It included using 4 Titan Black (12GB) GPU systems [**?**] and took 2-3 weeks to train a single network using 74 epochs and 5000 iterations per epoch. However, this was tested across more than a million images. Given the feature of small number of parameters and the simplicity of the model, its a worthwhile model to implement and adapt towards the Tiny ImageNet training set.

ResNet

The most recent architecture to have won the ImageNet challenge was the ResNet [8]. It was published in December 2015 and presented a further development of the VGG Net. The main focus was to eliminate certain negative phenomena seen when increasing the depth of a network. They categorized these effects into two cases: firstly, increasing the depth of a CNN can lead to vanishing and exploding gradients as they may not be initialized properly hence produce lower validation accuracies. Secondly, even if the network is initialized properly (via batch bormalization (discussed later)), the accuracy may

still decrease with increasing depth. They stated the most likely reason for this is that the accuracy becomes saturated and starts to degrade. They ruled, with certainty, that this degradation was not due to overfitting and adding more layers would lead to lower training accuracies. This was verified from an experiment where identity layers (let the outputs pass through) were inserted in between convolutional layers and the training accuracy decreased.

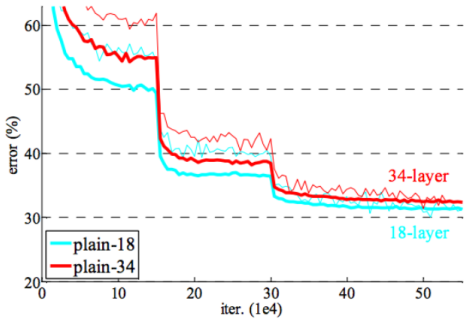
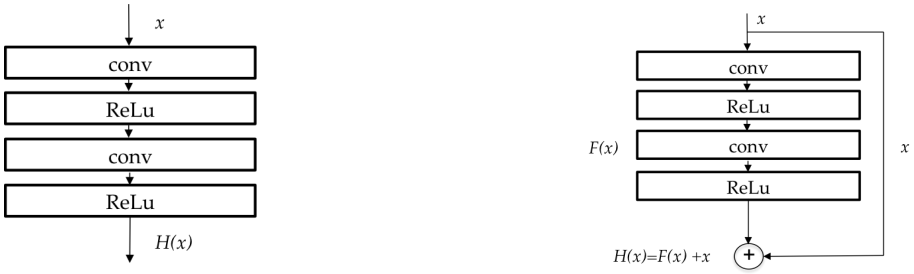


Figure 2.12: *An example of degradation whre the 34 layer network gives higher error rate than the 18 layer (found by creators of Resnet [8])*

Therefore, they believed the problem was centred around the way that the network was optimized. Subsequently, they aimed to solve this problem by using residual learning. Using this technique, a top-5 error rate of 3.57% was achieved.

Residual learning involves not relying on stacked layers to achieve the desired final mappings (of matching images to the correct class) but use a more intricate form. To illustrate this, consider a stacked network, the desired map of correct class labels is H(x) but with residual maps the stacked network must fit a different mapping of F(x) = H(x)–(x), where x is the original image [8]. The key hypothesis was the fact this forming mapping was easier to optimize than the stacked convolution filters.



(a) *A stacked (plain) module mapped onto H(x)* (b) *A ResNet module with different mapping*

Figure 2.13: *diagram showing how different mappings are performed*

Consider the gradient flows of the diagram of the plain module 2.13a. During back propagation, it has to go through all the weights in each and every layer. However, with a residual network, it contains an addition term 2.13b. This distributes the gradient to all its children equally, and so the skip connections allow the gradients from the last layers to feed into very early convolutional layers error close to the images. The function of the middle layers would be to add signals in between. It was found

this process made the optimization more efficient and also, given that the skip layers contained no parameters, there was little increase in computation. Note they tested this not only on the imageNet dataset, but across the smaller CIFAR-10 proving that this problem of optimization is not specific to a single dataset. Additionally, they too implemented a network-in-network architecture: they used 1x1 convolutions before and after 3x3 convolutions for the imageNet challenge and used ensemble models to produce the low error rate .

Note the best performing ResNet model used 152 layers with skip connections every two layers. It also interestingly down sampled the image from (after using similar pre-processing techniques to the AlexNet) 224x224 to 56x56 at the very beginning of the network. This network therefore may prove worthwhile to test if a certain amount of depth can be achieved considering the memory limitation of the hardware. The ResNet used 8 GPUs to solve the imageNet challenge for a 152 layer deep network.

Chapter 3

Tools for Implementation

3.1 Software

There are several softwares that are used specifically for creating and training deep learning architectures. The three most commonly used are Torch [39], Caffe [40], Theano [41]. Torch is based on LuaJiT, which is an portable scripting language that has automatic memory management making it one of the fastest scripting languages that can be compiled [42]. Whereas Theano is a python based language that was originally developed to efficiently evaluate multi-dimensional arrays. Finally Caffe is based on a C++ library with elements of Python and MATLAB. By analysing [43] and the documentation for each platform, several pros and cons of using each software can be established.

	Base Langauge	**Execution Speed**	**Memory footprint**	**Multi -GPU**	**Other comments**
Torch	LuaJiT	Competitive with Theano	Lua has smaller memory footprint than python	yes	Optimization and other criteria can be defined easily Supports the use of threads. Easy C interface
Caffe	Mixture of C++, Matlab, Python	Slower than Theano & Torch	Not as small as Lua	yes	Plenty documentation and library of models available
Theano	Python	Competitive with torch	Not as small as Lua	no	Optimized for mathematical tensor calculations

Table 3.1: Comparison of software

Given Torch has one of the smallest memory footprints and is based on LuaKit (one of the fastest

compilers), It was decided that it would be used as the implementation software for this project.

3.2 Hardware

The CNNs need to be trained on certain hardware and there are two readily available solutions: Central Processing Units (CPUs) and Graphical Processing Unit (GPUs). GoogLeNet used a large number of CPUs to distribute their operations efficiently. In particular, they built their own optimized software, the Distbelief [44], that distributed their loads across very large number of CPUs. They also used an additional server that kept count of the parameters. However, all of the networks discussed in the previous section used GPUs. There were a few main reasons for this:

Firstly, GPUs have a greater computational throughput than CPUs. The latter is fast at computing operations sequentially but has a small number of cores (processing units). While the former has many cores and has highly parallelized operations. However, there are less instruction cycles per each core to ensure maximum throughput [45].

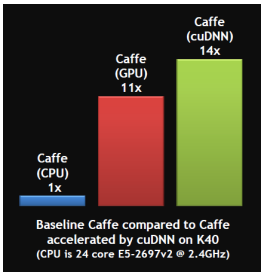


Figure 3.1: *The comparison of speed increase using cuDNN and a CPU. Although Caffe was used as a demonstration, it is expected torch has similar characteristics [46]*

Secondly, GPUs are optimized to preform matrix multiplications and in particular convolutions. They usually have a high level library that performs the conversion of the convolution matrix into a more optimized form specifically for the GPU. For example NVIDIA GPUs have their own custom software that contains a library, cuDNN, which implements highly tuned convolution routines [46].

Additionally, there are embedded GPUs that are becoming increasingly available given the demand of many commercial applications such as virtual reality. Once recently released one has a Jetson TX1 [47], has a 4GB GPU on board but only 256 cores. This makes it much slower to train a neural network.

However, there are two main issues with GPUs. When transferring data from the CPU to the GPU there is a chance a memory bottleneck could occur and GPUs are very expensive to ascertain. Four GPUs were used to train the VGG nets, NVIDIA Titan Black (12GB), and each cost nearly $1000 each [10]. An embedded TX1 costs $566 [47] to obtain. There are cheaper options that can be useful for a common user for shorter-term usage. Amazon Web Services offer a smaller NVIDIA Grid K520 (4GB) (g2 instance) [48], for rent of $0.65 an hour (with student credit of $100). This is one of the cheapest on the market. Note only certain laptops and computers have GPUs built in so this is not a viable solution. Since the NVIDIA Grid K520 had comparable memory size to the most advanced

embedded system, it was worthwhile to use to train the neural networks. Furthermore, given that this project is a step towards implementing onto embedded systems, only one GPUs used.

To handle the GPU–CPU bottleneck, there are several measures that can be taken. First of all, the data set can be split into smaller batches for training (see section 4). The CPU can fetch data from the memory while the GPU runs a complete backward and forward pass of the batch of data. This can be accomplished by using threads (operations running in parallel). One thread can fetch data from memory while the other thread ships already loaded data to the to the GPU. Additionally, the memory from a hard disk is slow to read, hence ideally the data should either be stored on the cache of the CPU and loaded in a contiguous format or pre-packaged onto a database. In torch, there is an available data-loader that performs these operations: it stores images onto a cache and applies threads to load the data into the GPU [49]. This project will aim to use an adaptation of this code.

Chapter 4

Designing the model

This section is aimed at explaining the design process of the algorithms with the prior knowledge, from 2 of the operation of CNNs and other systems used to solve the larger scale imageNet challenge.

The algorithms had to classify 200 classes of low resolution images (64x64). In order to initiate the design processes the data-set was examined. Following this, the design phase was split into three steps. Firstly, various pre-processing measures were examined in order to gain the maximum efficiency out of the model. Secondly, three models were designed inspired from VGG Net, Googlenet and ResNets. Finally, various post-processing steps were considered. A plan for implementation was then formed by combining these three steps.

4.1 The dataset

The dataset was obtained from [3]. It was split into three sets: training, validation and test sets. The test set was formed by taking 20,000 images from the 100,000 training images (100 images per class) as their were no labels provided with the test set of the data set. As a result, the remaining validation set size (provided with the data set) was 10,000 images and the training set size was 80,000 images.The dataset was split in this manner in order to test and tune the hyper-parameters with the validation set and confirm this generalization of the model with the test set.

Addititonally, the images, as mentioned before in section **??**, were of low resolution (64x64x3) as seen in **??** and were a scaled down version of the ILSVRC-2012 [7] ImageNet challenge. This allowed for less training time, less memory and expanded the types of hardware (GPUs) that could be used within the project specification 3. Although bounding boxes were supplied to localize the object, they were not tasked to be used in the algorithms: the object had to be recognized with clutter discounting the pre-determined localization.

Classification for low scale resolution images have been performed on the CIFAR-10 training set with over 90% accuracy [50]. However, this data-set included 190 more classes and more images with occlusion and clutter making the task of designing the model even more challenging.

It can be seen that some of the images are extremely difficult to recognise due to the low resolution and are subject to occlusion and clutter. However, there are some less coarse images as presented by

Figure 4.1: *A small sample of the data set - images are subject to occlusion, clutter and some even extremely hard to recognise by the human eye*

the duck and frog. Thus there is some noise present in the images and was something to consider when designing the system.

4.2 Pre-processing

Common pre-processing techniques for machine-learning include standardization [51]: firstly, the images have the mean of a sample set of training images subtracted across every feature. This ensures the data in zero-centred along ever dimension. Secondly the images are then normalized (divided by the standard deviation of a sample of training images) to ensure the pixel values are between 1 and -1:

$$\mathbf{z} = \frac{\mathbf{x} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} \tag{4.1}$$

Where the mean is estimated by

$$\boldsymbol{\mu} = \frac{1}{\mathrm{N}} \sum_{\mathrm{k}=1}^{\mathrm{N}} (\mathrm{x_k}) \tag{4.2}$$

and the standard deviation is found from:

$$\boldsymbol{\mu} = \sqrt{\frac{1}{\mathrm{N}} \sum_{\mathrm{k}=1}^{\mathrm{N}} (\mathrm{x_k} - \mu)^2} \tag{4.3}$$

An efficient implementation of Equation 4.1 is usually performed across the three colour channels (red, green and blue).This was preferred to subtracting a mean image as that would involve storing a larger array and performing more operations. It was also something that worked well in the VGGet architecture. Furthermore. to save computation, it was computed across a 1000 samples. Normalization represented a precautionary process when used for feature scaling - the images are roughly of the same relative scales (as the intensity values are nearly always between 0-255), these two processes removes any absolute uncertainty of unbalanced scales of features.

Although scaling may be a precautionary process for standardizing features. It is necessary when

it comes to updating the weights in the network. Since the weights are proportional to the error and the input and if all the inputs are positive - the weights will be positive. Therefore, all the updates in the network will have the same direction of change for certain inputs. This introduces a oscillating effect for the gradients to converge to the local minimum [52], slowing down learning and therefore introducing inefficiencies. By normalizing the data-set,there will be negative and positive weights and therefore it is found this speeds up convergence as there will reduce oscillations and result in a faster convergence to the the local minimum. Hence, for these two reasons normalization was included in the initial design of the models.

Additionally as seen by the AlexNet [6] and Googlenet [36], there were other pre-processing techniques used before training. These were mainly centred around the idea of augmenting the data set. As described in the background section 2 the network would be susceptible to overfitting particularly as it increases in depth. An augmented training set is a method one can use in pre-processing to reduce this effect. The technique involves increasing the training set by introducing affine transforms of the image and storing these as another sample. An indepth survey [53] found that affine transforms of cropping and flipping together can increase validation accuracy upto 3-4%. Figure 4.2 represents a method used with the ResNet[**?**].

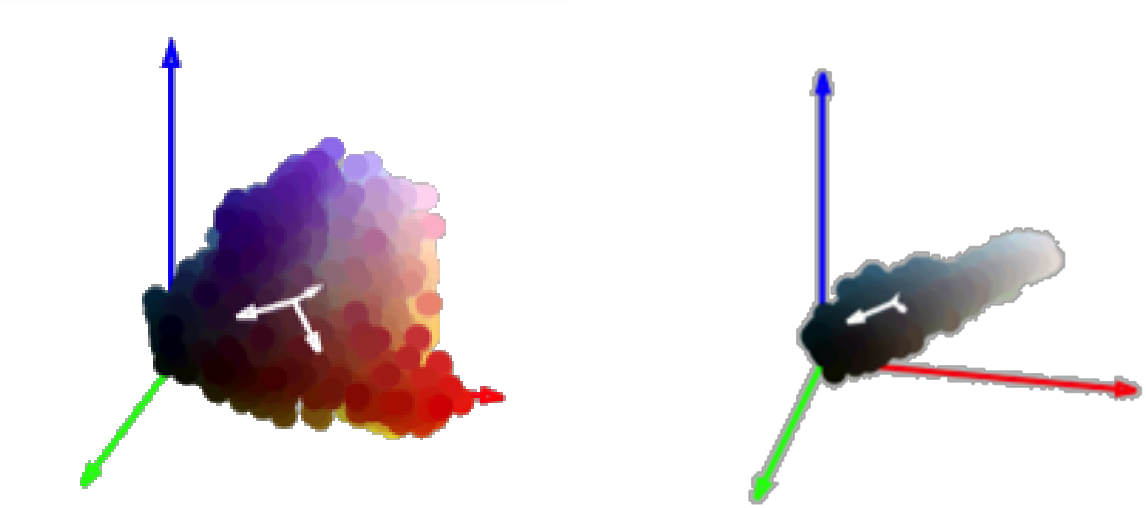
(a) original version

(b) flipped version

Figure 4.2: An example of cropping on an image from the Tiny imageNet training set. cropping was kept constant size of 56x56 and occurred at the following positions (from left to right): centre, bottom right corner, bottom left corner, top left corner, top right corner

Alex Krizhevsky et al. used a simple random cropping and flipping of the centre patch to augment the training data by a factor of 2048 in the AlexNet before passing it through the neural network. At test time, the above ten crop method was used to find the average prediction. Although the cropping occurrence was random when augmenting the train set, the size of each image remain fixed after cropping: It decreased from 256x256 to 224x224. In addition the GoogLeNet and VGG Net augmented the training set in the same manner as the AlexNet. It was found to bring an additional 1% accuracy increase of the validation set. With this initial design and while implementing the pre-processing steps, data augmentation by random cropping and flipping was used. The crop ratio was set to be the same as the one used in the AlexNet and VGG Net $\frac{224x224}{256x256} = \frac{56x56}{64x64}$.

Another augmentation technique the AlexNet, VGG Net and GoogLeNet used was applying colour jitter or RGB colour shift. It offset the colour shifts in the principle component directions.



(a) *The colour distribution and principle components of the butterfly image(above)*

(b) *the colour distribution and principle components of the duck image(above)*

Figure 4.3: *The white arrows represent the principle components of each colour (red, green and blue). The images were obtained from the training set*

One of way of achieving this was performing principle component analysis along each dimension of the depth of the image (colour domain) and adding spherical gaussian noise centred at 0 with a 0.1 standard deviation along these components . The Gaussian noise was scaled in with a way such that the standard deviation was multiplied with the square root of the respective eigenvalue values (as this approximated with the brightness channel of the image) and implemented along the principle components [54]. This technique does not distort the image as such perturbation only affects the brightness among the colour components that have the greatest variance.

An example of the colour distribution and the respective principle components can be seen in 4.3. Here two images from the dataset were taken (the butterfly from 4.2 and the duck from 4.1). Small amount of Gaussian noise was then be added and produce the images in figure 4.4.

It can be discerned from **??** that applying Gaussian noise with standard deviation greater than 1 leads to a more noticeable colour change in the image. Hence, this standard deviation value was used in the initial design of the algorithm. Note another augmentation technique could involve adding Gaussian noise in a non-aligned manner. However, this was considered after processing the initial design.

Thus, the pre-processing techniques for the initial design were inspired from previous models: it included augmenting the data in the two manners described above and standardizing each image. This was particularly useful to consider as the discrepancy between the number of training images in the image-net challenge and tiny image-net challenge is quite large: a factor of ten. If augmentation was not taken into account the network would most likely over fit the data-set as determined in [6] and inhibit the depth of the network that can be implemented. As a result, the data-set was randomly augmented with a uniform distribution. Note it was recommended to test the effects separately

Figure 4.4: *Examples of the images to show how much effect the Gaussian noise will have on the images. For each object: original image (top left), image with the 0.2 Gaussian noise applied (top right),image with the 0.5 Gaussian noise applied (bottom left), image with the 1 Gaussian noise applied (bottom right)*

(except for standardization) to understand how much they would increase the validation accuracy and complexity of the system.

4.3 Designing the model

When building the model, it was important not just to identify apt structures to implement,with regards to the depth and number of feature maps, but also incorporate possible techniques to aid attaining the highest possible validation accuracies. Models examined in the previous section all incorporate such methods to decrease the error rates. These techniques allowed for the overfitting to be reduced and for loss functions and weight initializations to be optimized.This subsection will explain the process of determining which structure of model was built and also which techniques should be incorporated if certain characteristics (such as overfitting) should appear when testing the model. This is essential due to the novelty of this field; there have been many publications from the time that the VGG Net and GoogLeNet was released to present that may be worthwhile to incorporate into the design.

Initializing Weights

An important consideration was weight initlization. Although this was something to be considered before initiating training, there is technique that can be incorporated into the architecture of the model. It essenttial consideration as this has been a challenge for deep neural networks and thus has been a key area of research. The pertaining reason is the weights cannot be initialized to the logical zero. This number is logical because it is the expected mean of the input after normalization and the weights are updated by the relationship δ input where δ is the scalar error [52]. Therefore, if half the inputs are negative then half the weights would be negative. However, if all the neurons were initialized to zero it would mean that all the neurons would return the same output and so during back

propagation all the gradients across the neurons would be the same. Following on, all the parameter updates would be identical across all the neurons and so the network would not correctly converge. Therefore, there must be some form of asymmetry in the initialization values.

One of way of doing this is randomizing numbers whose values would be close to zero and not identical. This would allow for each weight to have distinct updates. Note that the random initializations can be taken from a multi-dimension Gaussian or a uniform distributions.

Another challenge to overcome is the fact that the variance [55] of the output grows in proportion to the number of inputs. This can be derived in the following manner:

$$\begin{aligned} \mathrm{Var}(\mathrm{out}) &= \sum_{\mathrm{k}=1}^{\mathrm{n}} \mathrm{Var}(\mathrm{x_k w_k}) \\ &= \sum_{\mathrm{k}=1}^{\mathrm{n}} [\mathbb{E}(\mathrm{x_k})]^2 \mathrm{Var}(\mathrm{w_k}) + [\mathbb{E}(\mathrm{w_k})]^2 \mathrm{Var}(\mathrm{x_k}) + \mathrm{Var}(\mathrm{x_k})\mathrm{Var}(\mathrm{w_k}) \end{aligned} \tag{4.4}$$

assuming the weights and inputs are zero mean the equation reduces to:

$$\sum_{\mathrm{k}=1}^{\mathrm{n}} \mathrm{Var}(\mathrm{x_k})\mathrm{Var}(\mathrm{w_k}) \tag{4.5}$$

which scales down to:

$$\mathrm{n}\mathrm{Var}(\mathrm{x_k})\mathrm{Var}(\mathrm{w_k}) \tag{4.6}$$

Thus to ensure the output variance does not grow and affect convergence, the weights can be scaled with respect to $\frac{1}{\sqrt{\mathrm{n}}}$, when drawn from a distribution, so that it cancels out the input in the variance. The implementaion ensures there will be less chance of saturations and vanishing gradients when it comes to the activation functions. Thus this is a standard implementation and was first used by leCun when attempting to make the backpropogation more efficient [52]. Although this was specifically designed for sigmoid function activations, it has been used in a number of implementations such as [6]. Recently, there has been published work that has expanded on this initialization specifically catering for ReLu regularizations such as the MRSA intialiation [9].

However, Batch Normalization [56] has seen a prominent recent growth in use - as seen particularly in the ResNet. This technique is integrated into the model. It ensures that every layer has an input of the Gaussian distribution of variance 1. This means that the network does not have to continuously readjust for when the distribution of the weights in the preceding layer changes. It works by normalizing and transforming each and every feature or activations when a minibatch of inputs are passed through. the equations below found in the orginal paper explains why it presents an extremely desirable property to use.

$$\hat{\mathrm{x_i}} = \frac{\mathrm{x_i} - \mu_{\mathrm{B}}}{\sqrt{\sigma_{\mathrm{B}}^2 + \eta}} \tag{4.7}$$

equation 4.7 represents a simple normalization where B is the mini batch size and its done across the network. Despite this normalization producing a unit gaussian, it may be good posterior for a sigmoid activation but not so much for a ReLu activation. The following transform combats this and

it is this property that makes it so desirable to use (again the equation was obtained from []).

$$y^k = \gamma \hat{x_i} + B \tag{4.8}$$

Note this is an affine transform and γ and B are learn able transforms - it allows the network to chose how the apt normalization for the proceeding activation. The reason the parameters are tunable is that the normalization and transform it performs, still allows it to be differentiable. Therefore back-propagation can still be effective in the network. The main advantage though is that it reduces the need for careful parameter selection and in some cases it has proved to act as a regulizer - reducing the need for optimization techniques.

Thus in conclusion, the initial design of the models incorporated Batch Normalization. However, LeCun's initialization was first tested to understand the benefits that Batch Normalization could bring.

Activation Functions

As mention in section 2, the activation function is placed at the end of convolution layer and it is activated on recognition of certain features to create feature maps. There are a few common activation functions;

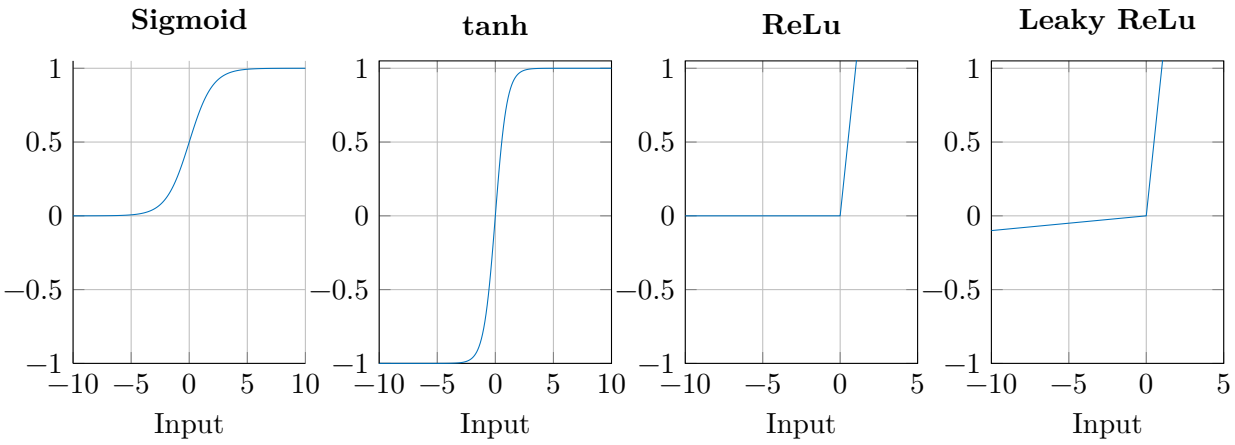


Figure 4.5: *Activation Functions*

Sigmoid Functions

This function has a mathematical expression $\frac{1}{1+e^{-x}}$. It was one of the most commonly functions used in neural networks up to 2012 [6]. It takes a real valued function and compresses it between 0 and 1. However, there are some disadvantages with this. If the output of the activation function is large or very small, it is likely to saturate the neurons. More importantly, the gradient at the saturation points are very small. Since gradients are multiplied during backpropagation, this function will stop gradients being transmitted to preceding layers if it is saturated. Thereby inhibiting the network to optimize correctly. The second disadvantage is that this function leads the outputs to be non-zero centred. This is a problem for the aforementioned reason that the weights will not optimize efficiently due the derivative of the weights being fully positive or negative creating an oscillating effect and

slowing convergence. Finally, the exponential term can be expensive to compute but relative to the large computations of the convolution layers this a small disadvantage.

tanh(x)

The tanh(x) was suggested by LeCun et al [5]. It improved on the sigmoid by allowing the data to be zero-centred but there was still a problem with saturated gradients.

Rectified Linear Unit (ReLu)

Hence Krizhevsky et al. [6] suggested to use a Rectified Linear Unit function. It;s mathematical expression is $f = \max(0, x)$. The key property is that it does not saturate for large values and as a result it speeds up convergence of gradient descent by factor of 6. It also more computationally efficient compared to the tanh and sigmoid function. However, it can be found for negative weights, it still saturates and kill gradients, the output is not zero centred and the gradient is not defined at 0 (but this case is rare)

Leaky Rectified Linear Unit A variation of this is Leaky Rectified Units [57] which have a negative slope for negative weights. This is een in the mathematical expression $f = \max(0.01x, x)$. This means the function does not saturate.. It also holds the same advantages as a Rectified Linear Unit.

There have been futher developments in the last year of further activation function such as the Parametric Rectified Linear Units [9] which allows the gradient slope of Leaky ReLu to be a hyperparameter ($f = \max(\alpha x, x)$. However, for the initial design of the CNNs, ReLu's were used as they have been implemented with success in the architectures considered in 2.

Loss Functions

The last layer of any CNN architecture is a linear classifier that quantifies the loss for each class. The two most common classifiers used in CNNs include Support Vector Machines (SVM) and Softmax [2]. There is not too much difference found in perfomance if one or the other is used as the last layer [58]. The core difference between the two classifiers can be found my considering their mathematical expressions:

$$\text{SVM Loss} = \sum_{t \neq y} \max(1 + W_t x - W_y x, 0) \tag{4.9}$$

$W_y x$= correct label score, $W_t x$=score of other labels

The SVM (hinge loss) aims to find a the loss of the network. It works in such that if the score of the correct label exceeds the score of any other label in the set by an arbitrary margin (here its 1), then a zero will be returned. If not, the losses for those classes that do not fit the margin will be summed. The process is repeated for every class and after which the mean will be taken to produce an average loss. Note, the average losses are not unique: different combination of class losses can give the same average loss.

The softmax classifier interprets the scores as unormalized log probabilities. As result, this classier first exponentiates the probability of the true class and divides by the sum of the exponentials of all

the other classes in relation to it:

$$P(Y{=}\mathrm{k}/X{=}\mathrm{x_i}) = \frac{\exp^{\mathrm{W_k x}}}{\sum_{\mathrm{t}} \exp^{\mathrm{W_t x}}} \tag{4.10}$$

Therefore to produce the loss function, the negative log of the expression 4.11 would have to be taken. The negative term ensures that the correct class has the lowest loss:

$$\mathrm{loss} = -\log(P(Y{=}\mathrm{k}/X{=}\mathrm{x_i})) \tag{4.11}$$

Thus both these classifiers produce a loss that needs to be minimized. The Softmax classifier will produce a loss for a particular class even if its close to zero whereas the SVM will produce a zero loss providing its within a specific margin. Also, the Softmax classifier or (the negative log likelihood function) loss is more intuitive and given there is not a big performance difference between the two classifiers combined with the fact most of the winning imageNet architectures have incorporated it, this classifier was used in the initial designs.

Optimization and mini-batches

The loss function quantifies how the correct class is predicted. However, a method needs to be used to optimise the weights to ensure the minimum loss. The traditional technique is to use gradient descent.

$$\mathrm{W(t+1)} = \mathrm{W(t)} - \eta \frac{\delta \mathrm{E}}{\delta \mathrm{W}} \tag{4.12}$$

Here η represents the learning rate, W represents the weights after iteration time t, and E represents the error. The gradient in this formula is found by the backpropgation method discussed in 2. This method can only calculate the average gradient after a complete run through of the data set: this technique is called batch learning more specifically. Given the cost surface has many dimensions and is non convex, there may be many flat regions and so there is no guarantee that the weights converge in the right manner: the convergence may not be fast or may not converge to a good solution.

A technique considered by leCun et al [52] suggest to use stochastic gradient descent. This involves taking a single sample from that large data set and estimate the true gradient. This operation would be performed for a number of iterations until a full pass of the data-set is achieved. The reason this can be done is the fact that much of the data set has many correlated or even identical samples. Therefore, if the gradient of a whole set is computed it is likely to return a value close to if the gradient was taken over a small subset of gradients from single samples. This redundancy through correlated data can make batch learning much slower and stochastic gradient learning much faster.

Secondly, it is found that better results can sometimes be produced by stochastic learning.This is because this process tends to be noisy when updates are performed and, because there are many local minima, it is unlikely to remain in one local optimum solution. A batch solution is likely to find a local optimum that may not be the global optimum and remain in it. This way it makes it more likely that the stochastic gradient descent would find the local minima. On the other hand, this noise may stall full convergence as the weights may fluctuate around the local minima instead of converge

towards it. One way to reduce the noise is to increase the stochastic single sample to a collection of samples or a "mini-batch size". There is not a set mini-batch size and it depends on the memory of GPU and timing limitations. The most common sizes are 32,128 and 256 and they are still found to converge much faster than full batches [59]. This is also more memory efficient than using the full batch - particularly when second order methods are used like the conjugate or newton method - which do not guarantee global minimum and involve storing the hessians.

If gradient descent is performed with mini-batches, it is recommended by LeCunn to shuffle the data so that no one class appears more than another and it ensures the model trains well on all sorts of images on the dataset. There are further techniques used to speed up the convergence of stochastic gradient descent. One of these is using the idea of momentum [60]:

$$\begin{aligned} \mathrm{v}_{t+1} &= \mu \mathrm{v}_t - \eta \frac{\delta \mathrm{E}}{\delta \mathrm{W}} \\ \mathrm{W}_{t+1} &= \mathrm{W}_t + \mathrm{v}_{t+1} \end{aligned} \tag{4.13}$$

This technique involves a velocity vector that accumulates values in the descent direction. It will have the highest magnitudes through the initial iterations but reduce closer to the minimum in the later iterations. The reduction is controlled by a hyper parameter μ which is analogous to the coefficient of friction. In all network architectures that won the imageNet challenge, SGD with momentum of 0.9 was used - a common value found in most implementations.

There has been recent further development of this technique with regards to the Nesetrov Momentum. This adds a term to derivative to influence the descent direction or gradient direction [60] when updating the velocity:

$$\begin{aligned} \mathrm{v}_{t+1} &= \mu \mathrm{v}_t - \eta (\frac{\delta \mathrm{E}}{\delta \mathrm{W}} + \mu \mathrm{v}_t) \\ \mathrm{W}_{t+1} &= \mathrm{W}_t + \mathrm{v}_{t+1} \end{aligned} \tag{4.14}$$

By adding this term to the derivative, the starting point for the next iteration is being influenced. It is found this technique further speeds convergence but it is not so widely used as compared to the SGD with momentum. There are per-parameter updates such as Adagrad [61] which uses an adaptive learning rate. However, this involves storing the backpropogation gradients over many iterations. Additionally, it performs a huge first step towards the minimum but suddenly reduces its step size to almost zero. As a result, it may work for some problems and not other: it may not generalize well.

SGD with momentum was used in the initial design but with possibility of trialling Nesterov momentum after some tests. This would speed up convergence and thus possibly aid increase in accuracies within the time limit. Additionally, the batch size of 256 was chosen. The batch size has to be multiple powers of two to take advantage of the computational speed ups of GPUs [10]. It was also small enough as not to use too much memory on the GPUs as seen in 4.3.2. This batch size was also used in the winning imageNet architectures,

4.3.1 Measures to handle overfitting

Ovefitting occurs when the weights do not generalize well: hence the training accuracies may be very high but the validation and test accuracies maybe very low. Overfitting in CNNs is common hence a number of techniques were developed to handle this. The aim of them was to reduce the sensitivity of certain weights or penalize the loss function. One which was discussed in section 2 is L2 regularization or Weight Decay. This is added to the loss function as penalty. The equation is repeated as a reminder from section 2,

$$\begin{aligned} J = \frac{1}{2m}[\sum_{i=1}^{m}(y_t^i - y_o^i)^2 + \lambda R(W)] \\ R(W) = \sum_{j=1}^{n}\sum_{l=1}^{m}(W_{j,l}^2) \end{aligned} \tag{4.15}$$

Here λ is a hyper-parameter that is needed to be set. AlexNet and the VGG Net have used 5 $\times 10^{-4}$. Hence, if overfitting does occur, this regularization term will be used.

Another common form of regularization is Dropout [62]. Dropout randomly selects, with a probability, p, only a certain set of neurons to train during a run through of the network. As a result, some weights become independent from weights of other nodes and thus this helps to generalize better:

(a) *dropout probability 0.5 on hidden layer during train time 1*

(b) *dropout probability 0.5 on hidden layer during train time 2*

Figure 4.6: *Showing Dropout of probability of 0.5 on hidden layers for two different forward propogations*

Lastly, augmenting the data set has found to have had success with AlexNet [6]. It adds noise to the training data thereby reducing the chances of producing very precise weighs. This was discussed extensively in pre-processing section and is one of the techniques that can be used.

4.3.2 Model Architectures

Three model architectures were initially designed. They were largely scaled down versions of the originally inspired designs. This was done for two reasons: firstly, there were memory and timing limits and secondly, it allowed for examination of different features and further expansion of the model during test time. To establish if the memory and timing limits were not being violated, estimates were made for both using common knowledge for the former and using pre-determined benchmarks from [43]. These benchmarks were established on much higher scale and expensive hardware (Titan Black GPU) hence an arbitrary increment in timings were taken.

Adapted-VGG Net

The first model to be considered was the VGG Net. This represented an advancement of the AlexNet but was a simpler architecture than GoogLeNet and ResNet. Additionally,the smaller receptive field sizes meant that the layers could be stacked to increase depth without increasing the number of parameters to the same degree as the AlexNet. It presented an ideal starting point considering the project restrictions. The structure of the model is presented below in 4.7.

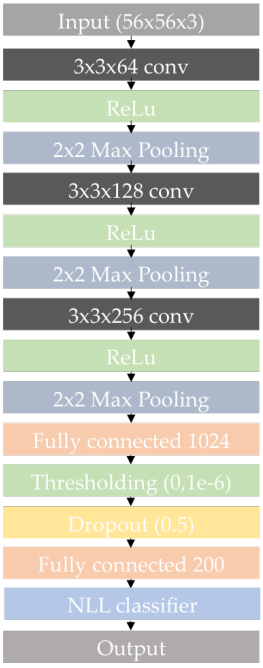


Figure 4.7: *Adapted VGG Net model: a scaled down version to enable the model to operate in the restricted memory environment. Note the 3x3 Convolutions*

Layer	**Size**	**Estimated Mem.**	**Weights**
input	56*56*3	10k	0
1st Layer conv	56*56*64	200k	36,864
Max Pooling	28*28*64	50k	0
2nd Layer conv	28*28*128	100k	147,456
Max Pooling	14*14*128	25k	0
3rd Layer Conv	14*14*256	50k	589,894
Max Pooling	7*7*256	12.5k	0
Fully Connected Layer 1	4096	4k	51,380,224
Fully Connected Layer 2	200	0.2k	819,200
Total		**451.7k values/image**	**52.97M Parameters**

Table 4.1: Amount of memory this model uses

The estimated memory in table 4.3 is given in terms of number of values. There are 4 bytes per value (assuming a floating representation is used). The bytes per image per a single forward pass of

a network is 451.7k $\times$ 4 = 1.8068MB/image. Note this needs to be multiplied by a factor of 2 for the backward pass of the network to calculate the error. Therefore the amount of memory for an image is 1.8068MB/image $\times$ 2 = 3.62MB/image. If a batch size of 256 images were taken, this would amount to 927MB on the GPU at one instance. Applying the same analysis to the number of parameters of each layer would result in using 202MB. However, does not take into account the parameters used in activation functions, back-propagation or miscellaneous memory. After some initial experimentation and analysing the timing difference from [43], the memory used from the total number of parameters would be 3 $\times$ 202MB. Consequently, an estimated total **1.5GB** of memory would have be utilized on the GPU with this model. However, this value did not include augmentation of the dataset but showed that there was 2.5GB available for implementing techniques like this.

In terms of timing, consider the following benchmarks from [43]. A 3x3 convolution (with stride 1) takes 11ms on a forward pass of the "cudnn.spatialconvolution" command with an input of 13x13 image and a batch size of 128 images to produce 384 feature maps. The backpropogation takes three times the time of the forward pass independent of receptive field, number of feature maps and input. Therefore, in total 44ms were taken per each batch run through. Additionally the longest time measured in the benchmarks was 400ms for a 9x9 convolution operated on a 64x64 input and a 128 depth This design has a much lower filter sizes and even with decreased computational capability, it was unlikely that each convolution would take more than 400ms - still allowing time for 40 epochs to be run. However, this was something that was needed to be kept track of at test time.

With regards to the model itself, the following should be noted with reference to the layers:

1. Each 3x3 convolution layer the image was zero padded by a single pixel around the border. This enabled the spatial resolution to stay the same:

$$\text{output width} = \frac{(\text{input width} - \text{Zero padding})}{\text{stride}} + 1 = \frac{(64-1)}{1} + 1 = 64 \tag{4.16}$$

$$\text{output height} = \frac{(\text{input height} - \text{Zero padding})}{\text{stride}} + 1 = \frac{64-1}{1} + 1 = 64 \tag{4.17}$$

   Additionally, the stride was set to 1 which allowed the convolution layer to densely sample the preceding layer. This was a characteristic determined by the original VGG Net.

2. The model in figure 4.7 was inspired from the 11 layer VGG Net with the last layer removed. The reason for this was two fold. Firstly, this ensured that there was enough memory space for further augmentation. Secondly, the number of images being trained was the tenth of the size of the original ImageNet and they are of four times lower resolution. Thus a layer of 512 feature maps may lead to overfitting the data . However, given the lack of present research into selecting the number of feature maps, this was something that was investigated in the testing phase with this model.

3. The input to the preceding fully connected layers was converted to a single dimensional vector and fed into the fully connected layers to downsize the dimensions to the number of classes. This was done in one less stage than that of the original VGG Net as the input vector into the fully connected layers were half the size. Additionally, doing it in two stages may have increased the

number of parameters that can be used but allowed for the use of drop out to reduce the effect of overfitting. The classification was then performed by the softmax layer.

This initial design left enough memory space for a number of things to be tested. One of which was batch normalization (discussed in the following section 4.3). The research behind batch normalization was published in 2015 but VGG Net was published a year earlier. Given the improvements found with the use of this technique, it was worthwhile to consider implementing this into the structure. Also, different types of augmentation techniques could be implemented before increasing the depth of the network to understand how the affected the accuracies.

Additionally, there was very little information inside the publication of the VGG Net indicating how the number of feature maps was chosen. Understanding, how this effects the final accuracies may prove to also be worthwhile.

Adapted-GoogLeNet

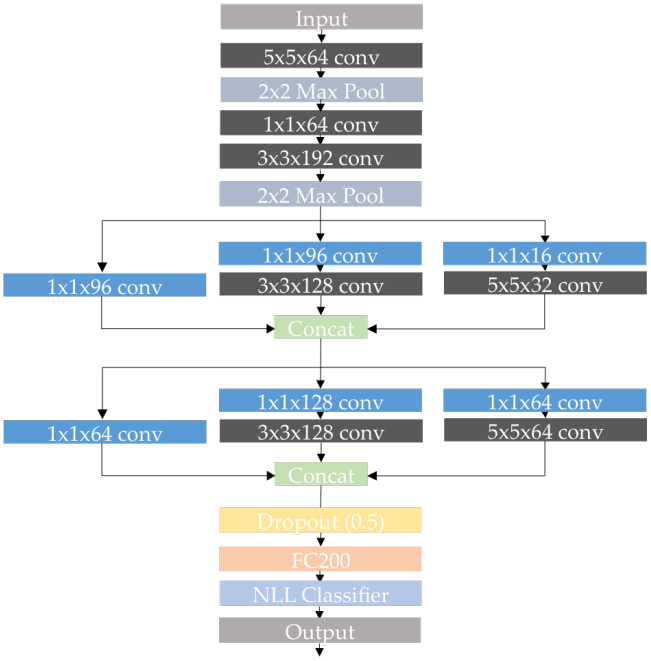


Figure 4.8: *Initial design of GoogleNet. Note the activation functions were removed for clarity. Every convolution layer is followed by a ReLu*

The total memory storage becomes (as explained from the VGG Net):

$$\begin{gathered} 4 \times 582.2\text{k} \approx 2.47\text{MB}/\textit{image} \\ \text{with backward pass included} \\ 2 \times 2.06\text{MB} \approx 4.94\text{MB}/\textit{image} \end{gathered} \tag{4.18}$$

If the same analysis were applied to the number of parameters, this would result in:

Layer	**Size**	**Estimated Mem.**	**Weights**
input	56*56*3	10k	0
1st Layer conv	56*56*64	200k	4,800
Max Pooling	28*28*64	50k	0
2nd Layer conv	28*28*64	50k	4,096
3rd Layer conv	28*28*192	150k	110,592
Max Pooling	14*14*192	38k	0
1st layer incept1.	(14*14*96 ×2 + 14*14*16)	40k	39,936
2nd layer incept1.	(14*14*128 + 14*14*32)	31k	123,392
1st layer incept2.	(14*14*64 ×2 + 14*14*128)	25k	147,456
2nd layer incept2.	(14*14*128 + 14*14*64)	38k	249,856
Fully Connected Layer 2	200	0.2k	10,035,200
Total		**632.2k values/image**	**10.72M**

Table 4.2: Amount of memory this model uses

$$4 \times 10.72\text{million} = 40.89\text{MB} \tag{4.19}$$

taking into account backprop and miscellaneous memory using same assumption as VGG Net

$$3 \times 40.89\text{MB} \approx 123\text{MB} \tag{4.20}$$

Thus for a batch size of 256, there would around **1.4GB** of memory. This estimation did not take into account the amount of memory used for concatenation layers and it would something that would be better understood at test time. Additionally, there was no augmentation included but the essential factor was that the memory available was still high enough to account for these factors while allowing for the expansion of the model. Also, on closer inspection, there is five times less parameters used than the VGG Net but its compensated by the increase of memory needed per image.

With regards to the timing, its difficult to estimate without implementing the model. However, by analysing the benchmarks [43] for the full 22 layer GoogLeNet a full backward and forward pass would take 1.02 seconds (using the 128 images,cuDNN library and a Titan Black GPU). If this were run for 390 iteration for 40 epoch, this would still take less than 11 hours (9 hours) of training. Although these were different environments, it somewhat still presented an upper limit of the timing which ensured this model would not exceed.

The model was changed in comparison to the original architecture in the following ways:

1. For each convolution layer, the image was zero padded by a single pixel around the border. This enabled the spatial resolution to stay the same while the down sampling was left to the 2x2 Max Pooling layers. Additionally, the stride was set to 1 which was characteristic of the original architecture.

2. This adaptation focused on extracting the first three layers of the GoogleNet. To begin with, instead of taking a 7x7 convolution, a 5x5 convolution was taken. The reason for this is that

the images were a quarter of the resolution on the original imageNet images. Hence, a 7x7 convolution would sample much more information of the Tiny ImageNet image in comparison. Hence, it was thought it was needed to downsize the filter to a 5x5. The same reason applied to using 2x2 Max pooling instead of 3x3 max pooling with a stride of 2: the information in the image is packed much more densely hence a stride would not be taken. Finally, the max pooling inception layer was removed. It was found only to increase accuracy by 0.6% after 22 layers [36] and it was identified as not a critical component in the overall architecture. Hence to ensure further expansion could be made at test time, this branch was removed - leaving the inception layer seen in the figure 4.9.

3. One of the core advantages of using a a GoogLeNet is the significant reduction of parameters. This happens because instead of using a fully connected layer, a average spatial pooling followed by a 1x1 convolution is used. However, for the initial design, a single fully connected layer was used. The reason for this was that it allowed for a basis comparison while still keeping the number of parameters much lower than the VGG Net. A key insight to draw is how much accuracy would this architecture compromise by using this method?.

Again, the initial design left enough memory space for a number of things to be tested: including batch normalization which was published a year after release of the Googlenet. It would be implemented into the structure in the same manner as the VGG Net - after every convolution layer a batch normalization would be added. Augmentation could also be used. It would also be interesting not just to examine what affect of spatial average pooling has on the shallow architecture but also by increasing the number inception layers while adjusting the number of feature maps: for instance, why were 192 spatial maps necessary in the third convolution layers?

These questions were endeavoured to be answered by iterating the design in test time.

Adapted ResNet

The reasons for designing the model in this manner are:

1. The model was designed to be very similar to the adapted VGG Net. The explanation for this was two fold: it gave a basis of comparison to see if there were improvements between the VGG model and the improved ResNet. Secondly, in the paper [8] adapted the original VGG model and expanded on this to produce the low error rates.

2. There was no clear explanation in the paper [8] as to how to connect identity layer to a layer with a different dimension. Therefore, a 1x1 convolution was used in the identity short-cut to adjust the dimensions as necessary. Additionally, an extra convolutional layer was added at the beginning of the network. This was a characteristic of the smaller architecture that was implemented in the paper for the CIFAR-10 dataset. Given this architecture is unlikely to extend to 152 layers and be comparable to the CIFAR-10 one, it was implemented in this network.

3. The original ResNet used average pooling before a single fully connected layer. A slightly larger average pool was taken here and used before the fully connected layer. This is because there

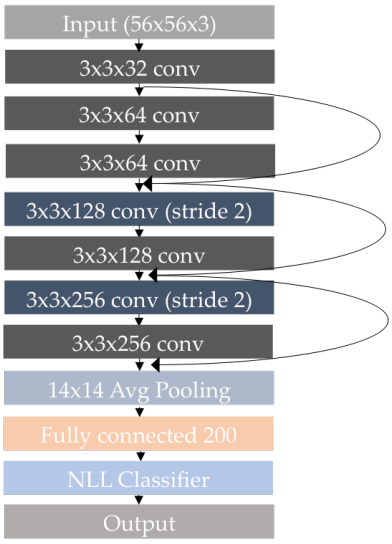


Figure 4.9: *Initial design of ResNet. Its an iteration of the VGG Net with a few alterations. Note the ReLu classifiers were removed from the diagram for simplicity. Also, 1x1 convolutions were inserted in the short cuts to ensure the dimensions matched when concatenating*

was one less residual module to perform down sampling with a stride of two in order to keep to a similar VGG adaptation.

The table below evaluates the memory used for the model:

Layer	**Size**	**Estimated Mem.**	**Weights**
input	56*56*3	10k	0
1st Layer conv	56*56*32	100k	864
2nd Layer conv	56*56*64	200k	18,432
3rd Layer conv	56*56*64	200k	36,864
Identity conv	56*56*64	200k	192
4th Layer conv	28*28*128	100k	73,728
5th Layer conv	28*28*128	100k	147,456
Identity conv	28*28*128	100k	8192
6th Layer conv	14*14*256	50k	294,912
7th layer conv	14*14*256	50k	589,824
identity conv	14*14*256	50k	32,768
Avg. Pooling	1*1*256	0.25k	0
Fully Connected Layer 2	200	0.2k	512,200
Total		**1150.45k values/image**	**2.04M Parameters**

Table 4.3: Amount of memory used by model

$$4 \times 1150.45\text{k} \approx 4.71\text{MB}/\textit{image}$$
$$\text{with backward pass included} \tag{4.21}$$
$$2 \times 2.06\text{MB} \approx 9.424\text{MB}/\textit{image}$$

If the same analysis were applied to the number of parameters, this would result in:

$$4 \times 2.04\text{million} = 26.47\text{MB} \tag{4.22}$$

taking into account backpropogation and miscellaneous memory using same assumption as VGG Net

$$3 \times 26.47\text{MB} = 79.4\text{MB} \tag{4.23}$$

Therefore for a batch size of 256, approximately 2.48GB of memory will be used. This still left 1.5GB to expand or change the model. The biggest decrease was the parameters: only 2.5% number of the parameters are used compared to the adapted VGG Net model. Additionally, with regards to the timing: its was likely to be comparable or less than the VGG Net. This was because its a very similar model and uses less parameters. However, this was a factor that had to be kept track of at test time.

Having designed the initial model, there were several insights hoping to be drawn at test time: Do the short-cut identities improve accuracies with greater depth? Is a 14x14 average pooling too large a filter to use compared to the 7x7 and 8x8? How greater depth could one go without violating the limits?

4.3.3 Hyper-parameters

Before training the models, several hyper-parameters needed to be determined. Cross-validation was expensive in terms of time to perform (due to the extremely large data set) hence the validation set was kept separate to the training set to determine these hyper parameters.The most important of which was the learning rate. Many of the learning rates for the original architectures were included in the papers. These learning rates could not be applied and used here, hence a systematic method had to be used to determine them before initiating training. One approach was to use a course to fine search to find the ranges of the hyper-parameters then use random search in between those ranges. A good starting point was to look at the learning rate of each original network and begin by testing a multiplicative factor of ten either side of this value. Then values would be halved till a suitable range of learning rates were found.Note it was not required to be performed across a large number of epochs but just a single one.

The next hyper-parameter to decide was how the learning rate would decay. This learning rate must diminish over time particularly when it reaches near the minima: it must not oscillate around the minimum point. Hence there are a couple of techniques to perform this: step, exponential and decay by the number of iterations. The most simple of which is step decay and it guarantees a decrease without having to consider the number of iterations or compute an exponential. Hence, an arbitrary value of 5% decay per iteration was taken. This would be trialled during test time and would be decreased or increased if need be.

The rest of the hyper parameters had been decided throughout the design section and are summarized in table 4.4.

Hyper parameters	Value/Range	Method of Determination
Learning rate	0.01-0.1	depends on model - course to fine search with random grid search used
Learning rate decay	0.05	To be changed, if needed after first test
Momentum	0.9	determined from previous architectures
Batch Size	256	a factor of two, small enough to allow plenty of memory space GPU
Number of iterations	390	enough to at least to cover the data-set. (100,000/256) was taken
Epochs	40	An arbitary number that would be changed at test time dependent on convergence

Table 4.4: Hyper-parameters

4.4 Post Processing

Whilst Pre-processing is concerned with adding noise during train time of the CNN, post processing is concerned with finding a way to reduce the noise when validating or testuing the models. One of the most common techniques is to use ten (as described in pre-processing) or scale crops on the test image. Here, the test image would be cropped ten times and each would be passed into the network and the result would be averaged. Scale cropping involves increasing the resolution of the images randomly and average over the predictions. It was found to work well for 256x256 images [27] but given the lower resolution 64x64 images used in this data set - it may not be appropriate as it may increase the blur further. Hence since ten crops was a common technique and a way to reduce noise, it was added into the initial design.

4.5 Implementation

The training of the architecture would be implemented in the following manner during train time:

Validation of the training set would occur after every iteration of the training set, hence a smaller batch would be taken randomly and the percentage of top-1 and top-5 accuracies would be calculated.

A 11 hour pre-defined training limit was used as this would represent circa 7 days of training on an embedded system (which is a commonly known standard amount of training time on normal GPUs), the maximum train time allowed was 722s per epoch for all the studied architectures.

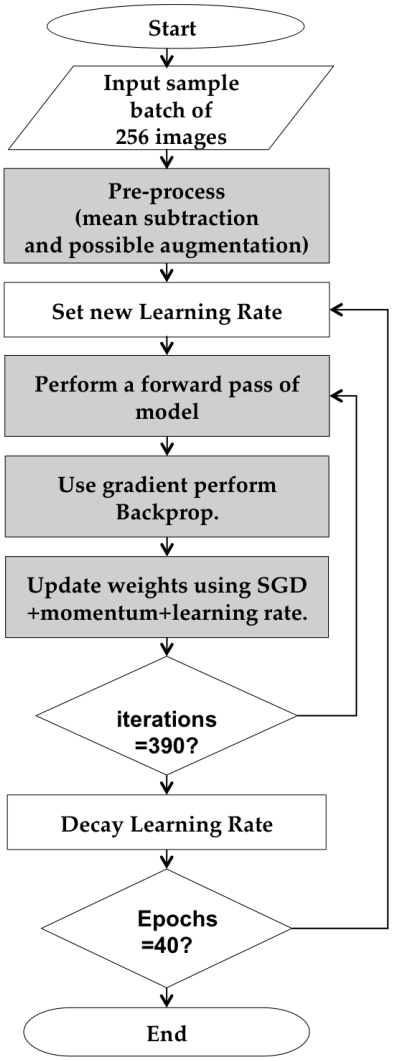


Figure 4.10: *The flowchart represents the methodology of training the network*

4.6 Summary

Thus in this section three models, based on the VGG Net, Googlenet and Residual networks, were designed and several key parameters to be found were identified. Additionally, there were some interesting questions that arose from such designs. Below, the testing points were summarized:

VGG Net First of all, the hyper-parameters were needed to be found. This specifically included the learning rate. SGD optimization with momentum of 0.9 along with the mini-batch size of 256 had already been decided to be used as per the original VGG Net and Alexnet. The initial model designed (without batch normalization) would then be tested with this set of hyper-parameters using the augmentation technique of random cropping and flipping. The model's validation and training accuracies and loss would then need to be determined with the post-processing techniques discussed. If overfitting were to occur, L2 regularization, dropout and further augmentation techniques such as the colour jittering via PCA would need to be evaluated. Then batch normalization would be implemented to fully understand the effect it has on the network. Finally, since the VGG Net's main

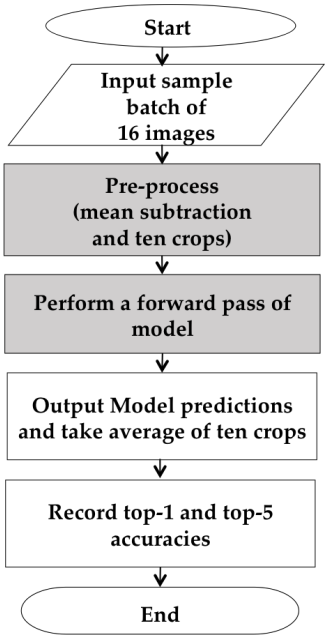


Figure 4.11: *The flowchart represents how validation accuracies would be obtained*

feature is the ease of implementing large depths, this too must be explored within the memory and timing limits. Additionally, since the effect of the number of feature maps was not fully understood, feature map sizes would be reduced to understand the how the validation accuracies change. This would allows the model to be trained using less memory.

Googlenet The test regime would be similar to that of the VGG Net: the hyper-parameters would be tuned and the model designed would be tested. If overfitting occurred different techniques would be implemented as in the VGG Net and batch normalization would then be used along with other augmentation techniques. The key feature of this model is the "network-in-network" architecture and aggregation of statistics from many mini models. The questions that needed to be answered in testing is what effects does this architecture have: would reducing the number of features in the network and adding more of these layers increase validation accuracies? Additionally, what effect would adding spatial averaging pooling have?

Residual Networks This is very much like the VGG Net. The initial designed was planned to be implemented but depending on the outcomes of the testing of the VGG Net, this could change. The learning rate would need to be optimised and two key insights would be explored Do the short-cut identities improve accuracies with greater depth? Is a 14x14 average pooling too large a filter to use compared to the 7x7 and 8x8? How greater depth could one go without violating the limits?

Chapter 5

Results

5.1 VGG Net

Hyper-parameter tuning

The Learning rate was found by running through the network for one iteration (390 epochs) and the validation and train accuracies were measured. As explained in the previous section, a course to fine approach was used. By examination of previous work (VGG Net, Alexnet and Googlenet), which found the learning rate of 0.01, the range was extended taken from this 0.001 to 0.9

Learning Rate	**Training accuracy**	**Validation accuracy**	**Cost**
0.09	1.14	1.12	5.29
0.045	1.34	1.10	5.25
0.025	1.14	2.40	5.24
0.02	1.02	2.66	5.22
0.017	2.13	2.17	5.22
0.015	1.23	2.46	5.22
0.012	1.12	2.00	5.22
0.01	1.47	1.93	5.22
0.001	0.39	0.7	5.29

Table 5.1: Table showing course-fine approach to learning rate

It can be seen from the table above that the optimum learning rate lies between 0.15 and 0.20 as the validation accuracies are the highest at this point and has the least cost(or loss). Thus a finer search was performed between 0.12 and 0.22 for 10 iterations:

There is some fluctuations maybe due to the initial starting points (the randomisation was kept constant for each epoch). Though, on closer inspection, around 0.015 both the validation and training sets have the highest accuracies and the lowest loss. The learning decay ray was kept constant at 0.05 and would be changed after the first complete training set if needed.

Learning Rate	Training accuracy	Validation accuracy	Cost
0.o2106	1.15	2.45	5.23
0.02015	1.26	2.55	5.22
0.01832	1.63	2.41	5.22
0.01747	1.72	2.52	5.22
0.01685	1.41	2.56	5.22
0.01478	2.34	2.62	5.22
0.01358	1.82	2.17	5.22
0.01327	1.23	2.46	5.22
0.01298	1.74	2.00	5.22

Table 5.2: Table showing random search of learning rate

Testing the model

Using the hyper-parameters found from above and not initially including regularization and any post-processing, the model was tested. This helped identify which of the key attributes aided an increase of the accuracy with the smaller data-set. The proceeding figures show the results.

Analyzing the basic model with weight decay

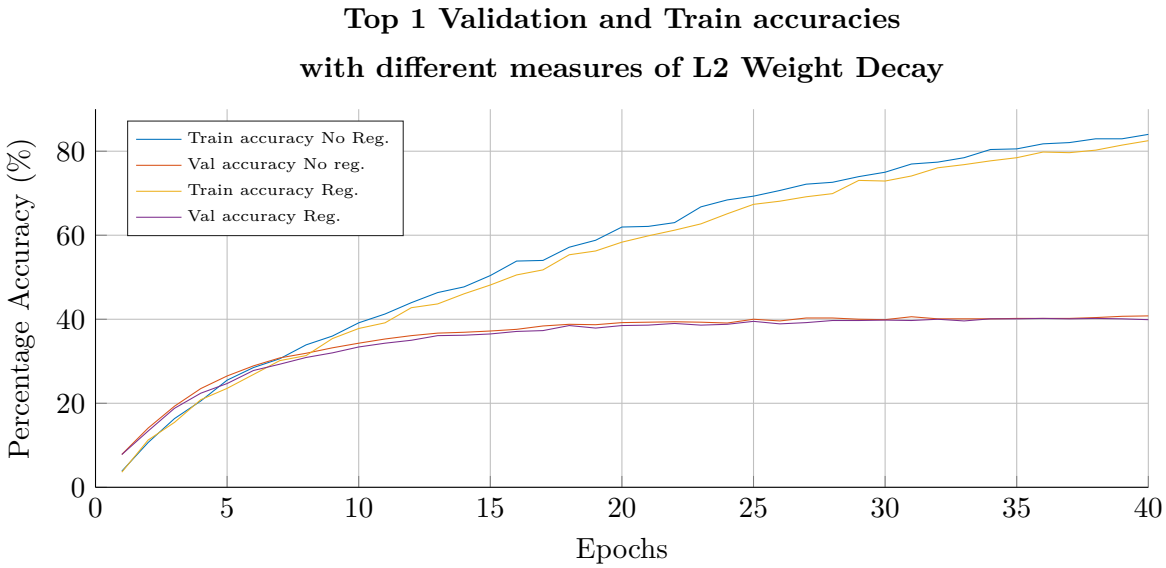


Figure 5.1: *PSD of* x(n) *for various values of* α

It can be seen that the maximum validation accuracy reached with this model is 40.2% and the maximum training accuracy is 84%. Additionally, the validation accuracy converges after twenty epochs while the training accuracy has not fully converged after forty epochs. Thus, this result can

	Top 5 Train. Acc.	Top 5 Val. Acc.	$\frac{\text{Time}}{\text{TrainEpoch}}$	Mem. GPU
No Reg.	97.2%	65.1%	178s	1605MiB
With Reg	97.2%	65.2%	178s	1605MiB

Table 5.3: My caption

lead to several conclusions. Firstly, given the gap between the training accuracy and the validation accuracy, there may be some overfitting present as well some possibility of some error in initializations. This was confirmed a low test error of 32%. Therefore, the first thing that was explored was weight decay with L2 regularization. The original VGG Net used a weight decay of $5 * 10^{-4}$ (and so too the Alexnet), thus due to the extent of overfitting this value was doubled and applied to the model. It produced a result that reduced the training accuracy by 2% to 82% and increased the validation accuracy by 0.6%. Whilst there were some improvements, this did not reduce the 41.2% gap and the training and validation accuracies. Thus it was deemed necessary to trial other overfitting techniques to understand if this would reduce large discrepancy before accounting for other measures such as initialization.

Furthermore, another conclusion that can be drawn with regards to the complexity. The weight decay does not add further memory usage but does increase the time for every train epoch by roughly 0.006s. It also has very little impact when predicting the top 5 most probable classes of each image (increases the validation accuracy by 0.01%). The memory of the GPU is also close to the estimate of the theoretical calculated value. The additional memory from the calculated value could be accounted for by other variables inside the code.

When analysing the training loss, It can be seen that regularization is having an intended effect of penalizing heavy loss. Note, as a sanity check, the loss has an initial value of 5 which is as intended as the probability of each class should be equal at the start and the theoretical loss should be $-\ln(1/200)$, 5.29. The training loss is close to zero for both cases: 0.654 when regularization occurs and 0.553 without it. The validation loss which has a minimum of 4.1824. By comparison, this confirms there is an element of overfitting further room for improvement in the model and also that the weight decay is the right value.

From the above results, it is clear that weight decay does not have a big impact even if a value is taken that is double the normal used in deeper models (which have greater susceptibility to overfitting). Due to the slight improvements though, it was included in the rest of other tests carried out. Note, there was no cause to increase the depth of the model as it would most likely overfit the data set further.

Further Augmentation

One of the key attributes of the first test was the amount of memory spare of the 4GB GPU and time available. It was already established in [6] that augmentation through the random cropping helped increased the accuracy by 1% and reduce overfitting. Also, the biggest difference in the data set used between past networks and the one designed was the number of training images inputed. Random

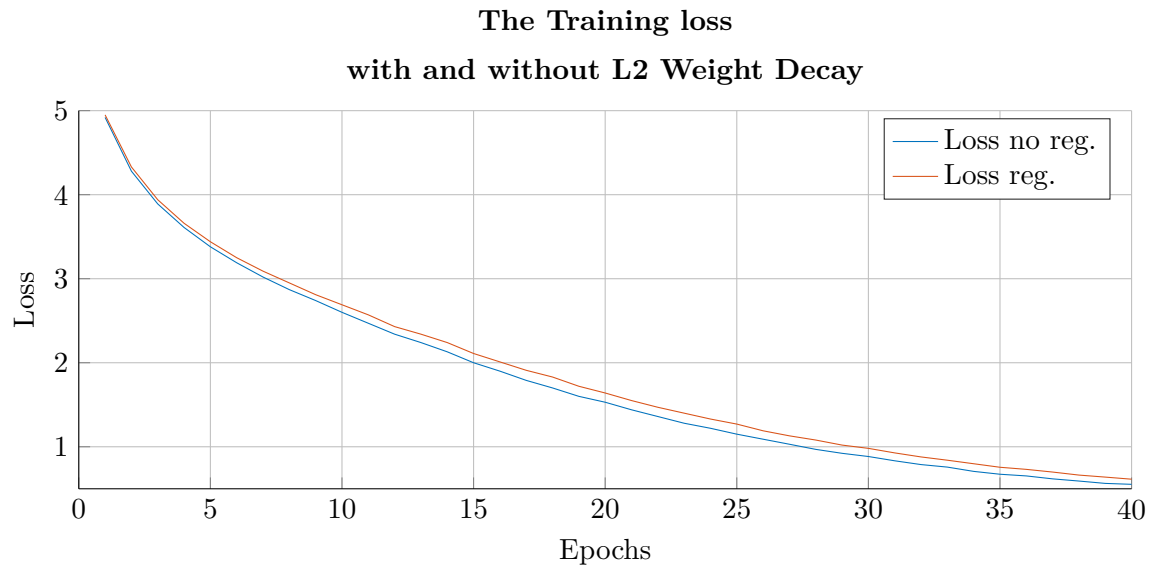


Figure 5.2: *Training loss with and without regularization. The slight increase when using regularization can be noticed*

crops were performed to augment the data set (produce roughly four times as many images). The number sampled out of this increased data-set was set by the batch size and number of iterations in an epoch. Ten crops processing was also applied at each validation epoch.

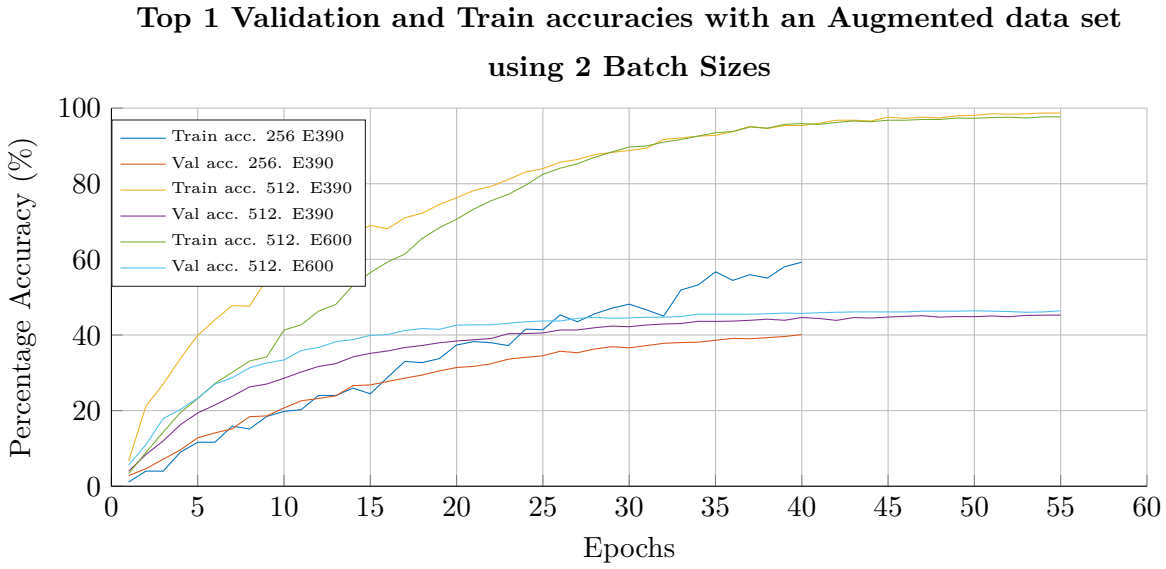


Figure 5.3: *Training and validation over batch sizes with increased augmentation of data*

By using a mini-batch of 256 with this augmented data-set, two factors can be concluded. The batch size no-longer provides a good gradient estimate to the larger single batch of the whole data set. This can be deduced from fluctuations in the training set accuracy representing the presence of extra amount of noise. Secondly, there is no convergence at 40 epochs. Therefore, two measures

	Top 5 Train. Acc.	Top 5 Val. Acc.	$\frac{\text{Time}}{\text{TrainEpoch}}$	Mem. GPU
256 E390	82.80%	65.6%	117.84s	1605MiB
512 E390	91.30%	70.18%	346.32s	2523MiB
512 E600	99.90%	71.20%	532.80s	2523MiB

Table 5.4: Memory, complexity, time per epoch percentage accuracy of the correct prediction appearing in the top 5 predictions

were taken to counteract this. Firstly, The number of epochs and the batch size was increased. The batch sizes usually have to be an order of two (for taking advantage of the parallel nature of GPUs and making memory allocation more efficient). Thus the maximum batch size can be 512 without exceeding memory allocation. Hence this batch size was chosen to augment the data set.

Thus it can be deduced from the graph that there are a fewer fluctuations in training accuracy than the batch size of 256. Additionally, there is much higher validation accuracy - 45.26% compared to the 40% of which ten crops testing produced a 1.6% increase in accuracy (on average) for each the test. If the number of iterations per epoch were increased to 600, a larger sample subset will be taken from the augmentation. However, this results in just 1.16% increase in validation accuracy for 53% increase in time taken to complete an epoch. To ensure that there is an equal probability of each class appearing in that sample subset, histograms were plotted that replicated the random procedure of selecting images to form the mini-batch for one whole epoch. It can be seen that there is an almost equal probability that equal amount of images from each class will be picked across 600 iterations in an epoch. Note the time per single epoch was increased but there was still a hundred second margin to attempt different techniques.

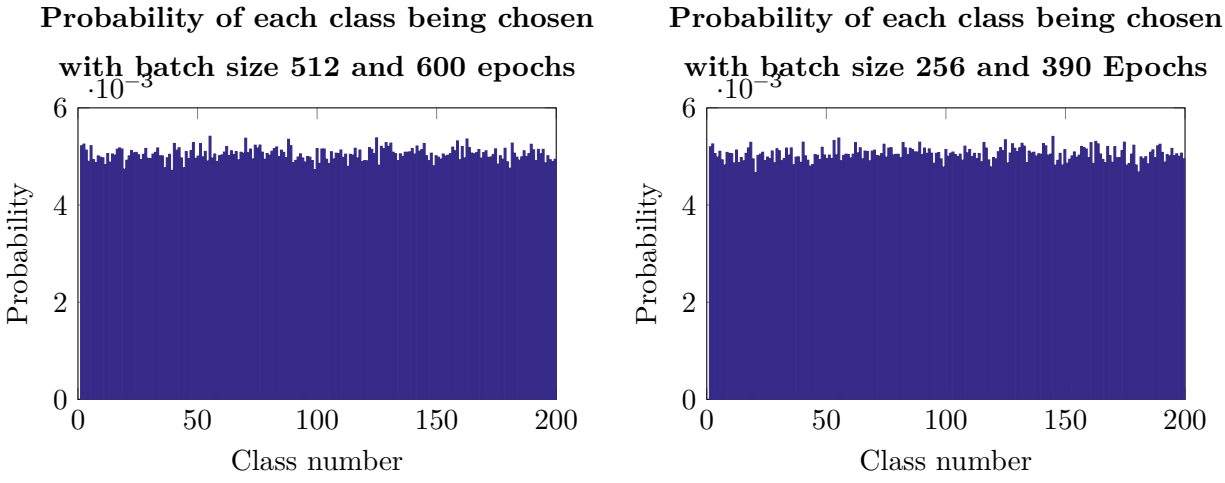


Figure 5.4: *Histograms showing the probability of occurrence of samples from each class across predefined number of iterations and epochs.*

After examination of 5.3 it can be deduced that there is an increased training accuracy but the gap between the validation and training accuracy is around 43.3% which is comparable to the previous trials. This was confirmed by the lower 38% test accuracy. Hence, although the validation accuracy

has increased, the training accuracy has increased in proportion. This re-affirms that there is still a problem with overfitting and possibly other factors. This is validated by examining the training loss reaching extremely close to 0 (0.007) once converged. Note the minimum validation loss has also decreased to 2.64 (256 batch size), 2.90 (512 Batch size 390 iterations) and 2.84 (512 batch size and 600 iterations).

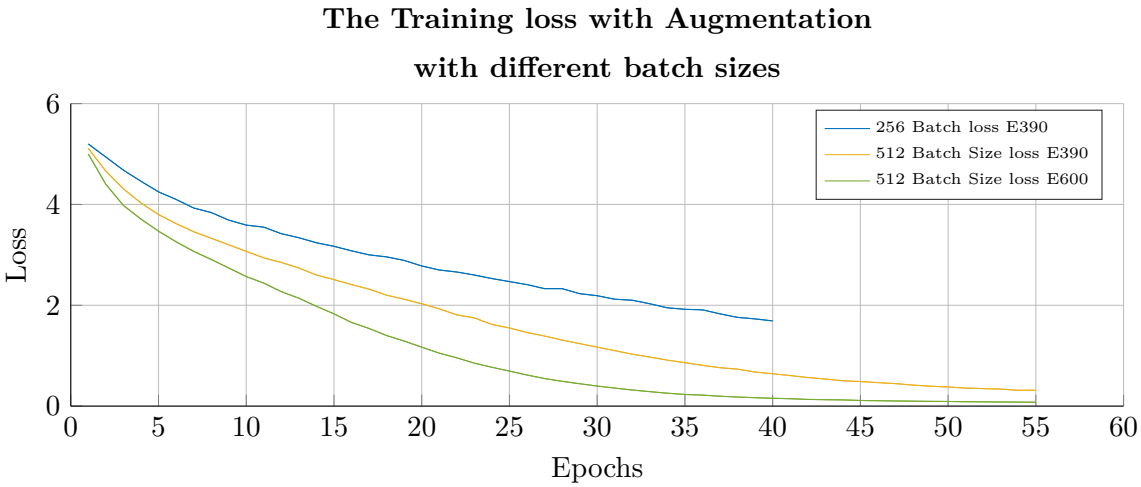


Figure 5.5: *Training Loss of the relative batch sizes of the augmented training set*

Thus for the rest of the design, 600 iterations and a batch size of 512 was kept constant as this produced the highest validation accuracies.

Using Dropout and Batch Normalization

Dropout was then introduced to further reduce overfitting. The plan was to sequentially add Dropout after each regularization layer starting with the layer containing the greatest number of feature maps (the last layer). The graph, 5.6, displays the first iteration of the design (where drop-out of value 0.5 was added to the last layer).

The core deduction is that the training accuracy is reduced by 1.8% and thereby overfitting is reduced. The validation accuracies have not changed and thus the gap between training and validation is still around 40%. It has been noted by [62] as one of the most effective regularization techniques. Additionally, a dropout value of 0.5 is being used this means only half of each feature map is active at one point of time during training. If the probabilistic value of dropout was increased there is a strong likelihood that validation accuracies would decrease and this would still not account for the large gap between validation and training accuracies. Hence, there must be an issue which is not directly related to overfitting. Instead of running drop-out for every layer, Batch Normalization was attempted next to help address the issue of initialization.

Figure 5.7 displays the result of using batch normalization. Here an immediate decrease in reduction in size between the validation and train accuracies can be noticed (a 22.5% gap) and the validation accuracies have increased too. This indicates that some fine-tuning was needed with regards to the weight initializations. As mentioned in [56], using batch normalization allows for less dependency on

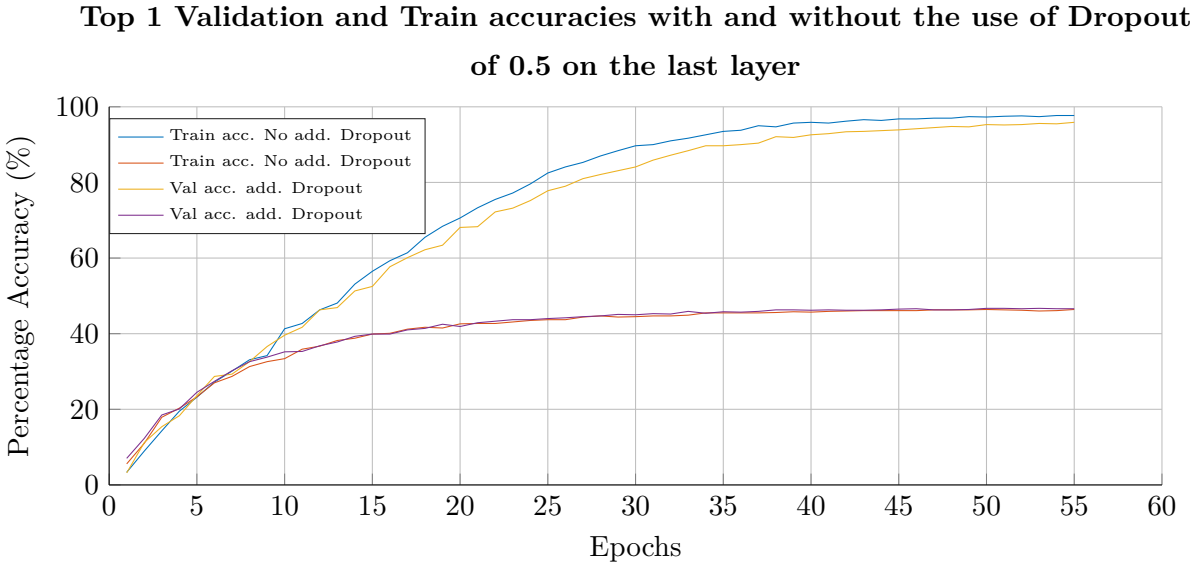


Figure 5.6: *Addition of Dropout on last layer (256 feature maps)*

the learning rates. Thus a more aggressive learning rate was trialled. Although the validation accuracy increased with higher learning rates (48.6% 49.4% and 50.1% for a learning rate of 0.015,0.017 and 0.025 respectively), the network was not able to generalize as well as the lower learning rates. This can be discerned from the increasing gap between the validation and training accuracy. Subsequently, increasing the learning rate leads to overfitting.

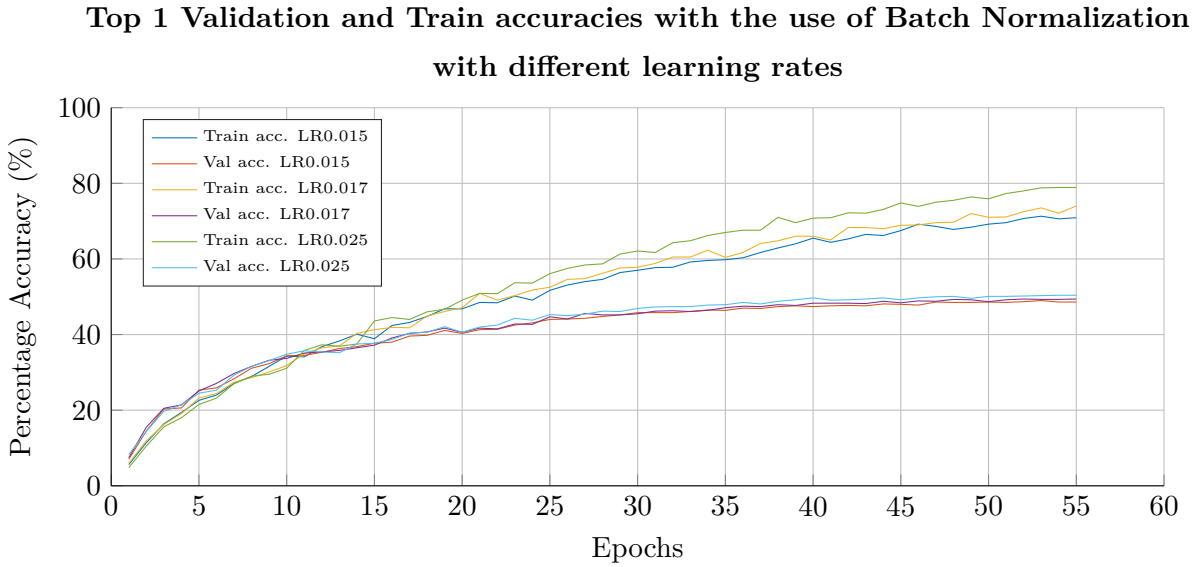


Figure 5.7: *Batch Normalization applied before ever ReLu layer and different learning rates applied*

Additionally, it was noticed that batch normalization stabilised the network at the expense of extra memory usage and 9% extra time taken to train an epoch . It would have been interesting an experiment using Dropout and Batch Normalization together, however there was not enough memory

	Top 5 Train. Acc.	Top 5 Val. Acc.	$\frac{\text{Time}}{\text{TrainEpoch}}$	Mem. GPU
Dropout	99.8%	70.9%	544.8s	2837MiB
BN LR0.015	91.6%	74.1%	576s	3883MiB
BN LR0.017	92.7%	74.1%	576s	3883MiB
BN LR0.025	95.2%	73.0%	576s	3883MiB

Table 5.5: Memory, complexity and percentage accuracy of the correct prediction appearing in the top 5 predictions

space available. In conclusion, batch normalization addressed several issues within the network and was used for the rest of the trials that followed with a Learning rate of 0.15.

What effect do number of feature maps at each layer in the network?

When designing the network, it was not clear as to how many feature maps should be set for each layer. This section aims to analyse what effect the number of feature maps has on the model. Due to memory limitations, the number of feature maps cannot be increased beyond the initially designed model. Hence feature maps of the order of magnitude below were examined:

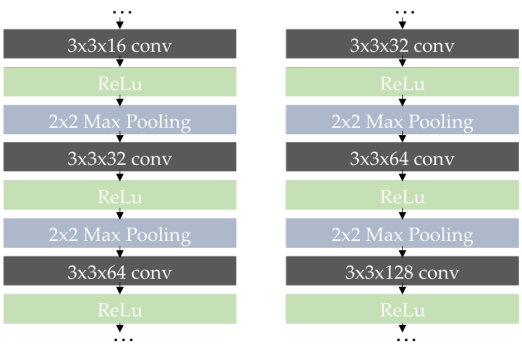


Figure 5.8: *Showing how the convolution models were scaled down. Right design labelled as model B and left design as model A*

Figure 5.9 shows that as the number of feature maps increase, the accuracy increases too. However, from these three tests a key trend can be noticed. As the number of feature maps per layer increases, the increase in accuracy decreases. Take for example the original feature map layer and model B: there is only a 0.6% in accuracy increase between the two (model B has top accuracy of 47.7% compared to the original network of 48.6%) . When comparing model A and model B, there is a 2% increase in accuracy (45.7% to 47.7%). Once again, it would be interesting to compare the memory and complexity associated with each layer. Table 5.6 presents these results.

It can be seen that Model A is 39% quicker to run than the original model and uses nearly half the memory. This presents an interesting insight:for only a slight decrease in validation accuracy there is a major decrease in memory usage. This may allow for a deeper network to be experimented using this select number of feature maps until the memory limit is reached.

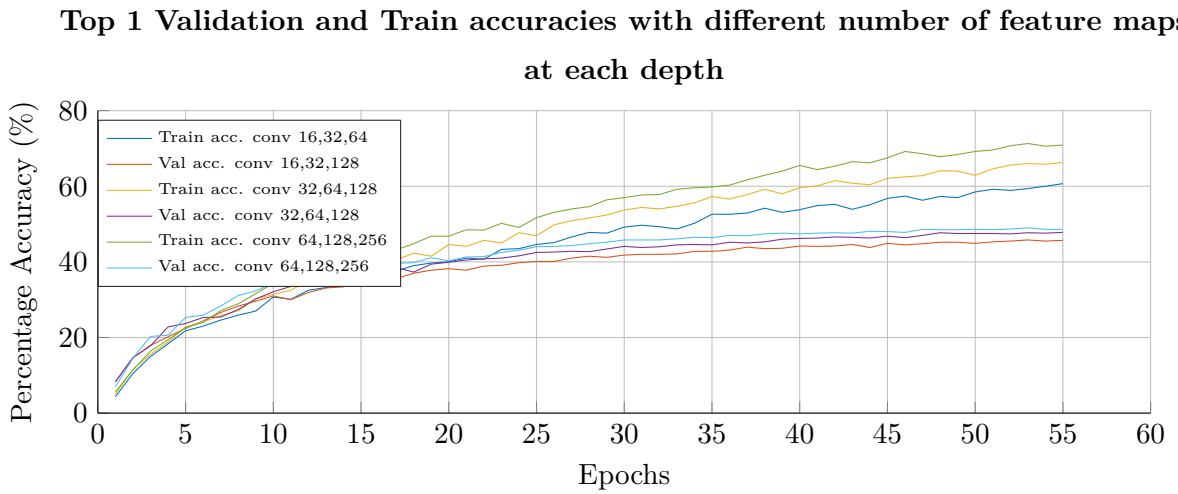
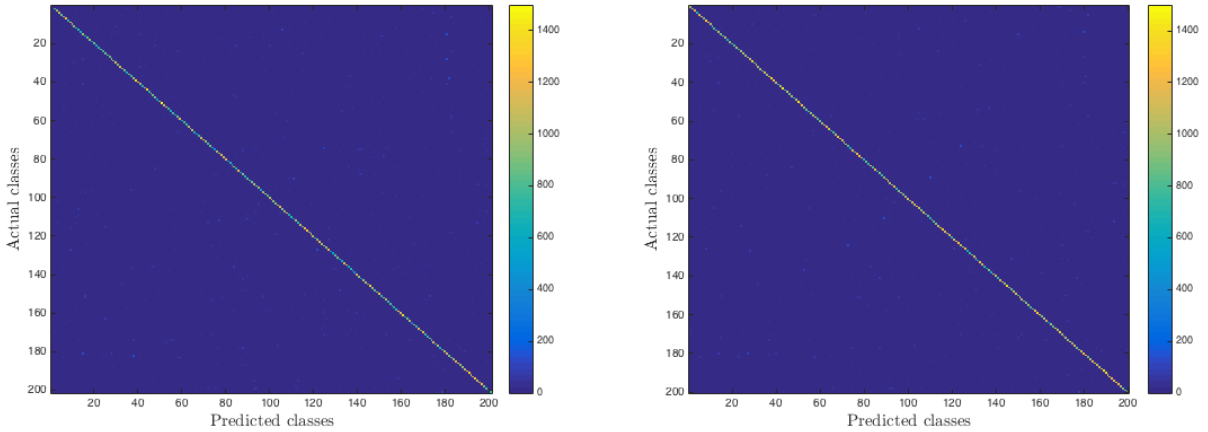


Figure 5.9: *Training and validation accuracies for different number of feature maps at each layer*

	Top 5 Train. Acc.	**Top 5 Val. Acc.**	$\frac{\textbf{Time}}{\textbf{TrainEpoch}}$	**Mem. GPU**
Model A	85.4%	70.8%	235.2s	1914MiB
Model B	88.4%	72.1%	345.6s	2571MiB
Orig. model	91.6%	74.1%	576s	3883MiB

Table 5.6: Memory, complexity and percentage accuracy of the correct prediction appearing in the top 5 predictions

To understand how the feature maps affect classification, it may be worthwhile to delve into specific classes. A commonly used methodology of comparing how many images of each class are predicted correctly is a confusion matrix. The more predictions that appear on the diagonal of the matrix, the better the model. Two confusion matrices for model A and B were taken as a sample.

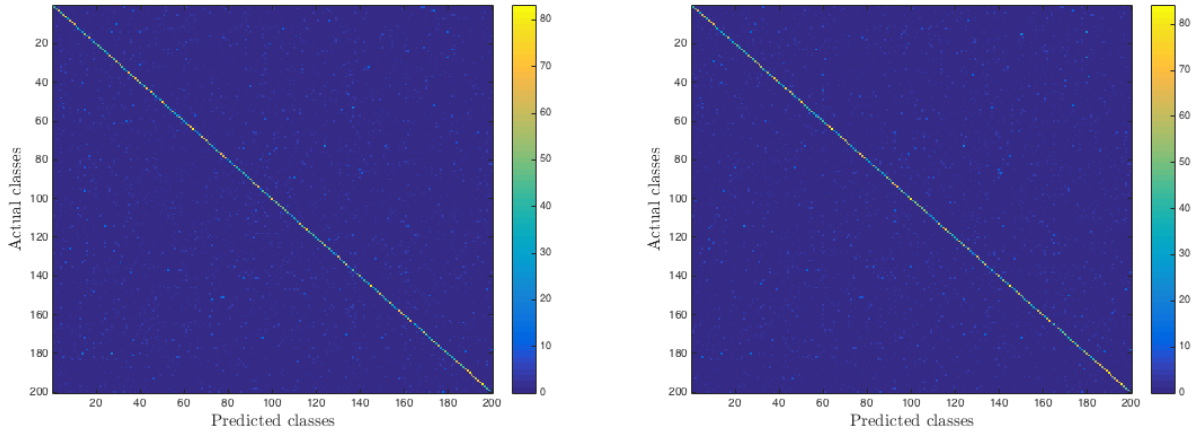


(a) *Model A Training confusion matrix* (b) *Model B Training Confusion Matrix*

Figure 5.10: *Confusion of training data set. Note a string diagonal with little misclassification*

The training confusion matrix presents a strong diagonal, indicating most classes are being classi-

fied accurately. By examining the colours, there seems to be slightly more classifications above 80% on model A than model B (more images are classified correctly).



(a) *Model A validation Confusion Matrix* (b) *Model B validation confusion matrix*

Figure 5.11: *Validation Confusion matrix for Model A and Model B. Note the increased in accurate predictions (blue dots) around the diagonal*

In figure 7.9 few more predictions outside the diagonal can be seen (observed by the increment of noise in the confusion matrix outside the diagonal). However, it is difficult to interpret which classes are less well predicted and perform a comparison between the two models based on the confusion matrix. This was also the case of the confusion matrix of the original model hence for conciseness, it was included in the appendix .1. Thus to gain a deeper understanding as how the neural network is behaving, a more manual breakdown of the classification of the classes was taken. The tables below include the failure cases and success cases for each model during test time.

class	**accuracy(%)**
Wooden spoon	5
Plunger	7
Syringe	10
chain	13
Umbrella	13

(a) *bottom 5 prediction labels with Model A*

class	**accuracy(%)**
dugong	78
bullet train	78
goldfish	80
school bus	82
trolley-bus	83

(b) *Top 5 prediction labels with Model A*

The biggest failure cases are images of wooden spoons. On closer inspection, the network is confused between wooden spoons, drumsticks (8%) and hour glasses (7%). It is unclear which filters the network uses to identify classes which has led to it being active area of research. Extremely recent and intricate attempts have been made by [63] to perform a backward pass through the network and breakdown each convolution layer to understand which features are being extracted. However, this does not given insight into how individual feature maps interact to extract such features. Attempts were made to replicate this algorithm without much success. The next alternative was to visualize the classes as seen in the first row of 5.12b. From this it can be seen that the textures presented by the

example of the wooden spoon and the drumstick are quite similar. Also, perhaps the shapes(edges) of the hourglass figure and the end of the wooden spoons are nearly identical and hence being confused. Note four out of the bottom 5 cases have straight vertical edges to them and given the low resolution of the images, it may have been hard to distinguish between them. This perhaps can be confirmed by analysing the top 5 predicted classes. By visual inspection in 5.12a, the best case images are the ones where the object constitutes a major segment of the image. For instance, the trolley buses usually have less clutter present in the images: it is much smaller (in terms of pixel scales) smaller relative to the size of the bus. Additionally, the goldfish has a very distinct texture differentiating itself quite easily from other classes. The standard deviation of class predictions was found to be 17.24. Thus there was quite spread on classes being analysed correctly however, this needs to be analysed with respect to the model B.

The tables below shows that Model B has similar failure cases to model A. Except the Umbrella produces the worst case failure case. The nearest class to the umbrella is an ice lolly(6% classification) and a sombrero (5%). By visual inspection from the second row in 5.12b some reasons can be discerned as why this may bet the case. The sombrero has a very similar shape to an umbrella and is held at the same position. Hence its becomes quite difficult to distinguish the two classes and only low level features can be used appropriately for this (such as the finer details). However, with low resolution images and limited depth, this becomes extremely difficult to recognise. The top 5 predictions, not

class	**accuracy(%)**
umbrella	4
wooden spoon	6
plunger	9
syringe	12
bannister	12

(a) *bottom 5 prediction labels with Model B*

class	**accuracy(%)**
dugong	80
bullet train	80
monarch butterfly	82
school bus	82
goldfish	84

(b) *Top 5 prediction labels with Model B*

only includes classes above 80% accuracy but also includes the monarch butterfly. Since there are several species of butterflies in the data-set (a fine-grained class), this indicates that textures may well be more distinguished with larger feature maps. However the trend where the image is the centre piece of the image is classified more accurately (larger Region of interest). The statistics of the class as whole show that 73 classes are predicted accuracy greater than 50%.

class	**accuracy(%)**
plunger	5
wooden spoon	10
umbrella	11
syringe	13
bucket	14

(a) *bottom 5 prediction labels with the original model*

class	**accuracy(%)**
sulphur butterfly	81
bullet train	81
obelisk	82
dugong	86
monarch butterfly	87

(b) *Top 5 prediction labels with original model*

The tables above show the original model failure cases and best cases. The bottom five cases have

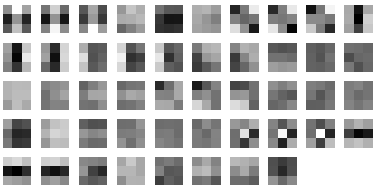
improved accuracies from model A. Also, neighbouring classes for the plunger include the a bathtub (7% recognition accuracy) and a bottle (7% recognition accuracy). Again, on further visual inspection in 5.12b, it shows that a plunger is found near a bathtub. This is quite common across the training set too. Hence this is not as much an object recognition problem but a localisation problem. The bottle and the plunger images confusion is harder to understand but again may be due to edges and colour. Note in the top 5 cases, the two species of butterfly are present. Both species have different textures but similar shape. This implies that with more feature maps, it is more likely that textures will be distinguished. Note only 96 classes are classified with above 50% accuracy and there is a standard deviation of 17.03 which is lower than model A and Model B.

(a) *most accurately predicted classes across three models tested*

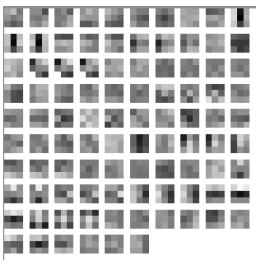
(b) *faliure cases of Model A(top row),B(middle row) and Orginal (bottom row)*

Figure 5.12: *Success and failure cases across the three models*

To gain some understanding of what is happening inside the model, the weights and the output of an image of the first layer can be visualized. Although its very difficult to correspond these weights to the actual image, a deductions can be made. There are 48 (16 $\times$3) 96 (32 $\times$3) weights for model A and B respectively in the first layer and they are represented in such a way that white parts represent higher values and black parts as low values. It can be seen that every filter is not completely black, hence there are no dead filters which means the network is being initialised properly. This was also the case for the original model (appendix .3).



(a) *Weights of 1st layer of model A*



(b) *Weights of 1st layer model B*

Figure 5.13: *Weights of the first layer of Model A and B*

However, a more useful analysis is considering the feature map of each network. To analyse this,

a success (goldfish from 5.12a) and a failure cases (wooden spoon from 5.12b) were passed into the network and after each convolution layer, a sub sample of a feature map of was taken as shown in figure 5.14. These are the outputted feature maps of this images for model A. From this, it can be deduced, as expected, each feature map performs the same process on the image. For instance in the first layer, it looks like the 12th feature map performs image enhancement and the the 7th feature map performs some sort high pass filtering. In the second layer, the images are slightly more blurred, but the goldfish seems to have more recognisable after being processed by feature map 2 and 12. The wooden spoon image is slightly more blurred and darker in appearance. After the 3rd layer of feature maps, the goldfish is nearly indistinguishable and the wooden spoon cannot me observed at all. Furthermore, an interesting feature is that fact the goldfish is shape is extracted completely in the first convolution layer where as the network is finding it very difficult to extract the wooden spoon. This may account for the low accuracy rates seen.

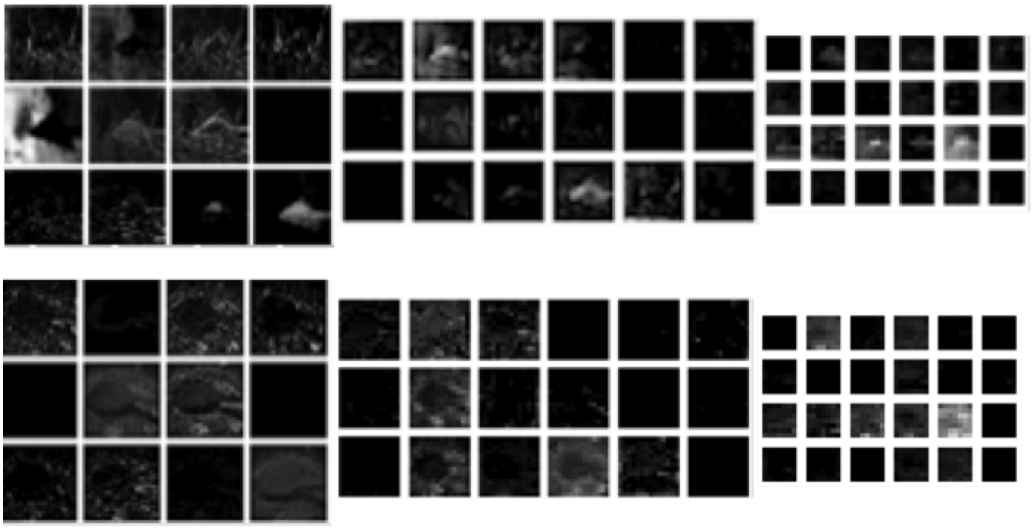


Figure 5.14: *Output of feature maps from model A with a goldfish (success) and wooden spoon (image)*

Note as mention above, it can be observed from figure 7.4 and 7.5 (appendix) that increasing the number of feature maps allows textures to be analysed. This verifies why the two different species butterflies were able to be recognised.

Thus in conclusion, the number of feature maps correspond to the number of features that can be extracted from an image: Increasing the number of feature maps is the same as augmenting the number of filters per layer. This allows the network to recognise finer grained images. However, beyond a certain point (as observed by the model B and the original model) there starts to less accuracy increase hence possibly displaying convergence. However, given the lack of available memory to test models with increased number of feature maps than the original, it was difficult to verify this conclusion. However, a key question can arise from this: If you decrease the number of feature maps but increase depth of the network, does this increase the validation accuracy?

Increasing the depth

The only way that depth can be increased while keeping the structure of the network would be to decrease the number of feature maps from the original design. It was found from the previous experiment that decreasing the number of feature maps by half leads to a small decrease in accuracy but large increase in memory availability. As a result, the models shown in 5.15 were used to understand what affects depth had on increasing the accuracy.

(a) *Subsection of the deeper model - D1* (b) *Subsection of the deeper model - D2*

Figure 5.15: *Subsection of the two deeper models tested (D1 and D2) - with extra Convolutional layers*

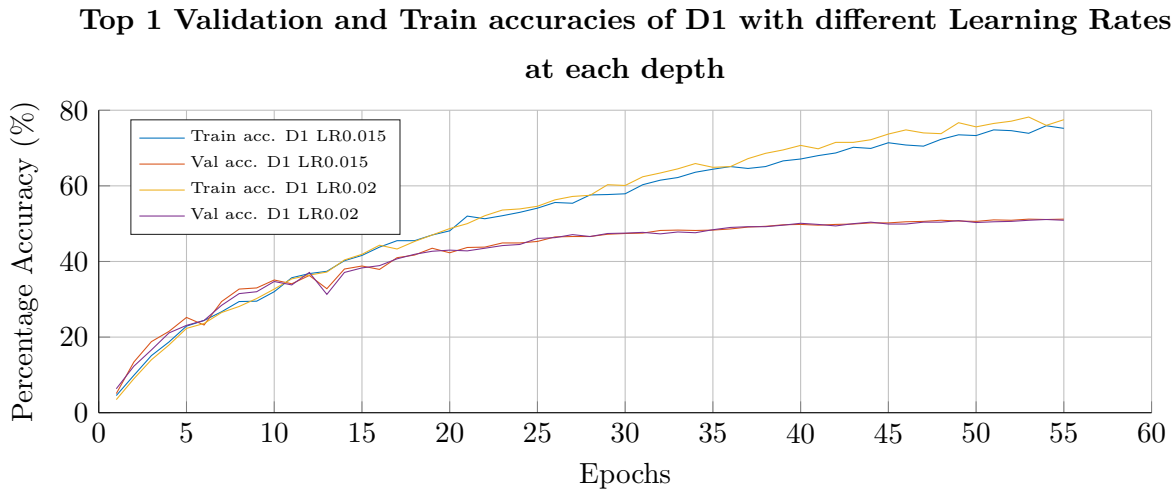


Figure 5.16: *Training and validation accuracies for D1*

from 5.16 The validation and train accuracy graphs can be seen for D1 with different learning rates. There is a increase in train accuracy for a learning rate of 0.02 while there is no validation accuracy increase - in fact, there is a 0.1% decrease in top validation accuracy. Moreover, this model presents the best top 1 accuracy (51.2% and 51.1%) and the best top 5 percentage (76.0%) accuracy

of all the VGG Net models tested. This makes the error rate of the top 5 accuracies reduce to just 24% which is 7% difference to the 2012 AlexNet which was trained over 2-3 weeks. Additionally, this was performed with just 8 hours and 24 minutes of training. The test accuracy for this model was a slightly lower 47.3%. It presented similar failure cases to the previous experiment but the lowest class accuracy was 8%. With this result, it was attempted to build an even deeper network D2.

	Top 5 Train. Acc.	Top 5 Val. Acc.	$\frac{\text{Time}}{\text{TrainEpoch}}$	Mem. GPU
D1 LR0.015	94.8%	76.0%	547.8s	2838MiB
D1 LR0.020	93.6%	75.8%	547.8s	2838MiB
D2 LR0.015	92.7%	74.7%	651.2s	3483MiB
D2 LR0.020	88.8%	73.4%	651.2s	3483MiB

Table 5.10: Memory, complexity, timing and percentage accuracy of the correct prediction appearing in the top 5 predictions

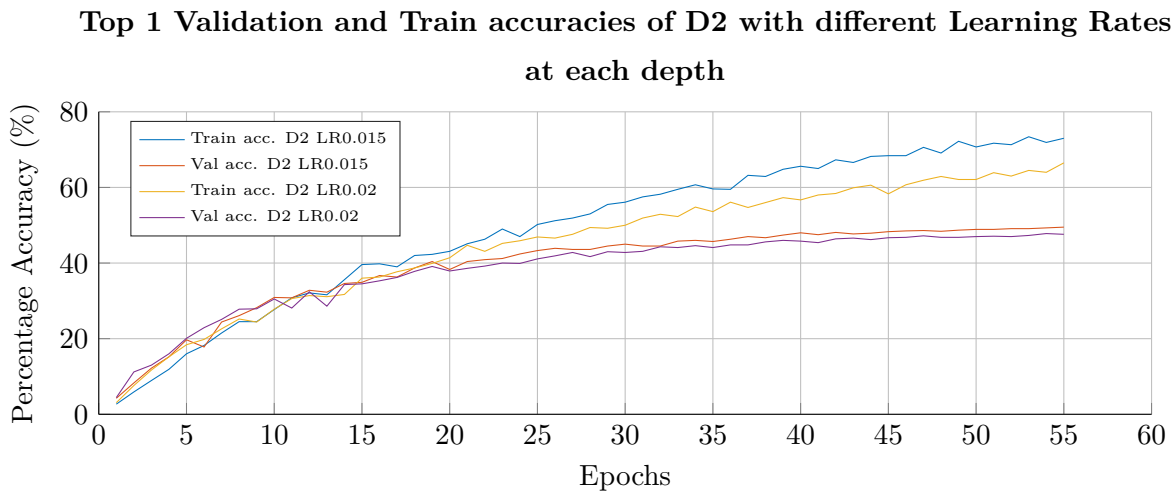


Figure 5.17: *Training and validation accuracies for D2*

From figure 5.17 it can be discerned that going deeper decreases validation accuracies by nearly by 1.6% (for a learning rate of 0.015) and 3.5% (for a learning rate of 0.02). It also increases the time of training. The lower validation accuracies can be accounted for by the reasoning of the ResNet development in 2.13: the deeper the model may lead to a decrease in validation accuracies as the weights are not being optimized efficiently. Hence, it was thought it was optimum to finish testing this model and trial other architectures.

5.2 Adapted GoogLeNet

As discussed in section 4, GoogLeNet represents a very different architecture due to the inception layers. To analyse the characteristics of the model, the initial model would be tested and then some alterations would be made to understand if another concatenation layer would increase validation accuracy. Finally some experiments would be performed to reduce the number of parameters by using

average spatial pooling instead of the fully connect layer. The hyper-parameters were tuned in the same manner as the VGG Net and a learning rate of 0.177 without augmented data was found to be the optimum (appendix 7.3). Hence this was used in the initial design.

Testing the initial design

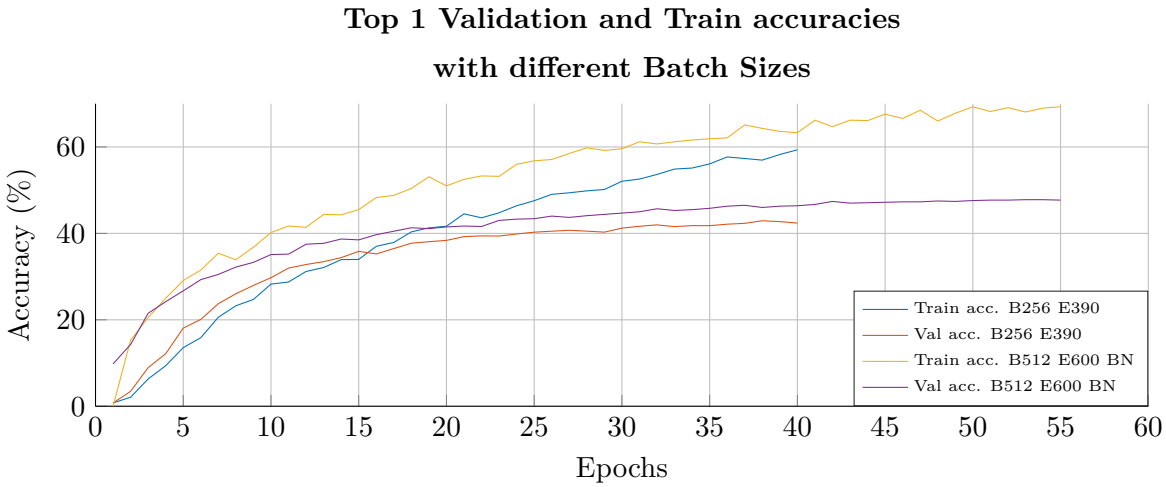


Figure 5.18: *Training and validation accuracies for initial design with and without augmentation techniques used to train the VGG Net*

	Top 5 Train. Acc.	**Top 5 Val. Acc.**	$\frac{\textbf{Time}}{\textbf{TrainEpoch}}$	**Mem. GPU**
B256 E390	85.06%	68.39%	399.2s	2838MiB
B512 E600 BN	89.6%	70.1%	716.8s	3918MiB

Table 5.11: Memory, complexity, timing and percentage accuracy of the correct prediction appearing in the top 5 predictions

The initial plan was to test without augmentation. However, given the success of the VGG Net with augmentation, it was trialled straight into the network. Although, the learning rate may needed to be optimised again, it was decided to test the initial design and make alterations to the batch size and the number of iterations. Hence it can be seen that for the initial design the learning rate for a batch size of 256 was too small as it failed to converge. Therefore, to ensure a large amount of augmented data was sampled, The number of iterations in an epoch was increased to 600, a batch size of 512 was taken, batch normalization was applied and ten crops of the validation set was taken which was to provide a 1.2% increase in accuracy. Due to memory limitations being exceed, batch normalization was added in the inception layers and not in the initial convolution layers: the inception modules compose majority of the model and so it would make logical sense that, if a trade off were to occur, these modules included a technique that would increase the validation accuracy. This increased the top 1 accuracy from 42.73% to 47.8% while inducing faster convergence. There was not much overfitting as the gap between two curves was circa of 20% and a test accuracy of 40% and 46% was

produced. This gave a basis of comparison for the further adjustments to the model.

Increasing the number of Inception Modules

The memory was already reaching its limits with a batch size of 512. If this batch size were to be maintained and benefits of augmentation were retained, a compromise would need to be made in the architecture. There was already a question with regards to whether the 192 feature maps was needed and given it composed of 23.6% of the memory of a a single image passed through the network. This was replaced by a single 5x5 convolution with 64 feature maps. This filter size was identical to the first convolution layer - hence it was found to be a natural extension to be used here. Additionally, to further decrease memory and to possibly add further inception modules, the 1x1 convolutions were removed from the first inception modules. The reason this could also be done, is that the initial layers could also provide the extra non-linearity that could feed into the inception module. This formed the architecture seen in 5.19

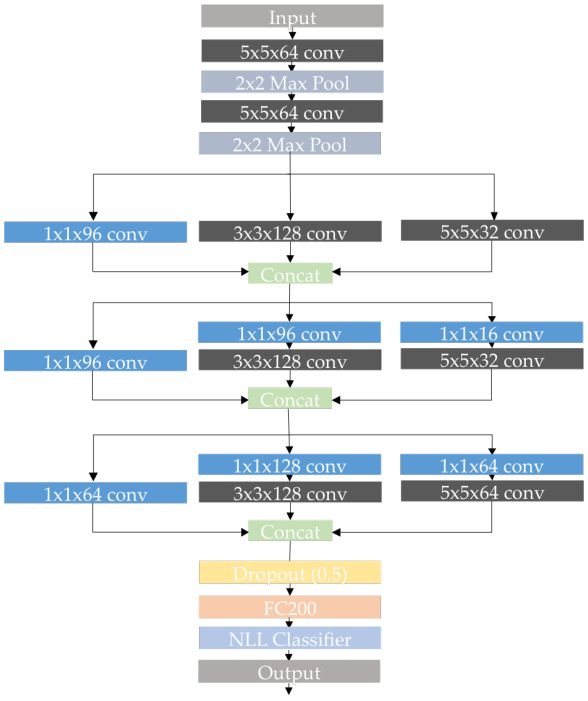


Figure 5.19: *Architecture of the reformed GoogLeNet to be tested with an extra concat layer*

This produced the following results:

	Top 5 Train. Acc.	**Top 5 Val. Acc.**	$\frac{\textbf{Time}}{\textbf{TrainEpoch}}$	**Mem. GPU**
Extra Incept.	93.80%	72.80%	703.2s	3745MiB
Initial model	89.6%	70.1%	716.8s	3918MiB

Table 5.12: Memory, complexity, timing and percentage accuracy of the correct prediction appearing in the top 5 predictions

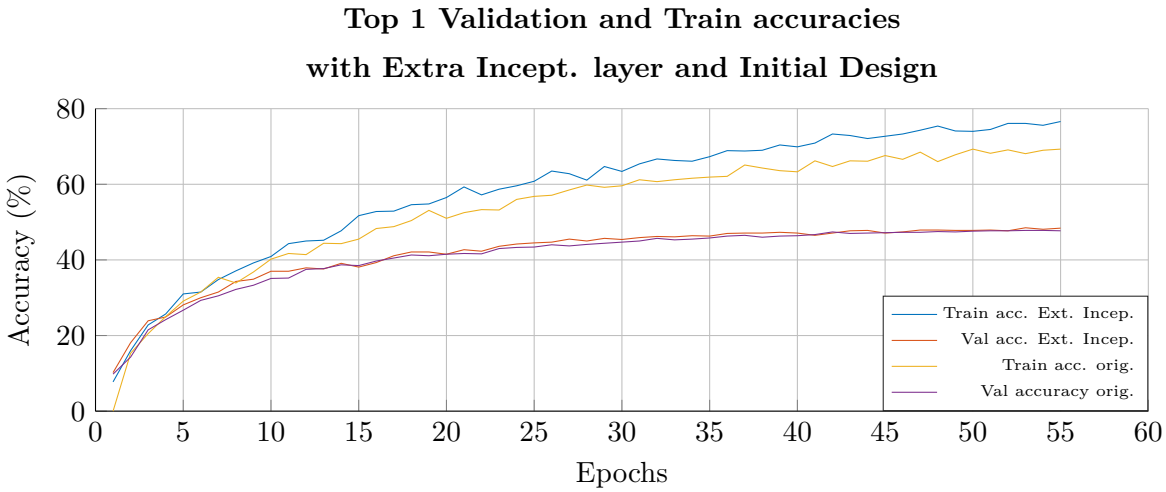


Figure 5.20: *Training and validation accuracies with an extra concat layer and the original network*

Here, the data set produced a slightly higher top 1 validation accuracy - 48.5% an increment of 0.7% - and a 2.75% increase in the top 5 accuracy. This proves that an extra inception layer can replace a very large initial convolution layer for shallower architectures. There has also been a very slight decrease in timing and a large decrease in memory usage: making this architecture more efficient. The test accuracies also prove that architecture generalizes well - producing a test accuracy of 45.4%.- but lower than the original architecture. In an attempt to better understand the extra inception layer, firstly the best and worst case scenarios were analysed and one of image of each case was passed through the network

class	**accuracy(%)**
umbrella	6
bucket	11
wooden spoon	12
syringe	14
bucket	14

(a) *bottom 5 prediction labels of model*

class	**accuracy(%)**
trolley-bus	80
sulphur butterfly	80
monarch butterfly	82
goldfish	84
school bus	84

(b) *Top 5 prediction labels model*

The table shows very similar top-5 and bottom-5 results to the VGG Net. This maybe because of the clutter in the bottom predictions and domination of the objects in pictures of the top 5 predictions. Since again the wooden spoon and the gold fish were one of the best and worst recognised classes, these images were taken and passed through the first set of convolution layers, just before the inception module. Another set of outputs of these images was taken just after the inception module. The results are displayed 5.21.

Here, it can be seen that the goldfish shape has been completely extracted from the picture after the first set of convolution layers - much like the VGG Net. However, after the inception modules, the images are much less sharper and only vague outlines can be seen of the goldfish can be seen. The wooden spoon is extremely faint and it seems that very vague features were extracted which are not

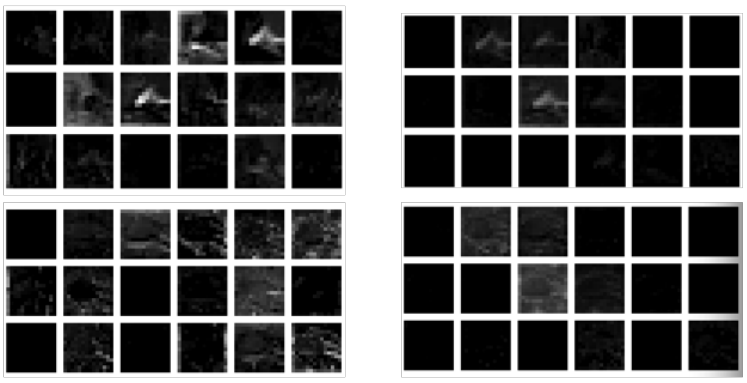


Figure 5.21: *Left: a sample of outputs from convolution layers, Right: a sample of outputs after the inception layer. The top row represents a goldfish and the bottom row represents a wooden spoon.*

intelligible - but may be intelligible to the CNN.

The bottom and top cases can also be compared to that of the original architecture:

class	**accuracy(%)**
wooden spoon	4
plunger	5
umbrella	12
pole	13
Labrador retriever	15

(a) *bottom 5 prediction labels with the original model*

class	**accuracy(%)**
trolley-bus	78
goldfish	81
sulphur butterfly	81
bullet train	81
espresso	82

(b) *Top 5 prediction labels with original model*

It can be seen that they both have different failure cases. If the closest predicted class to teach of the bottom cases was analysed it can seen there were some similarities between the two architectures. The wooden spoon's highest predicted class was a wok for both architectures (6% for the original and 6% for the other). This was different to the drumstick in the VGG Net and may suggest that the inception layer is examining the whole image (since the wooden spoon can be found in wok) than a particular subsection. This also could be explained by the larger 5x5 filter in the inception modules which could analyse a larger area. Additionally, there was a new top prediction for the original architecture: an espresso. On examination of the validation and training set images for this class, there was very little clutter in any of the images - hence explaining why it has a high success rate of being recognised. The extra inception architecture, can differentiate different intra-class variations: with the two butterflies in its top-5 predictions. The original architecture is not so strong. It was becoming confused between differentiating between two breeds of dogs. This, as in the VGG Net, could be because the extra depth allows for extra textures to be discerned and differentiate between intra-classes.

The confusion matrix shows a strong diagonal but its very difficult to discern the differences between the architectures. As a result, it was included in the appendix in 7.3.

It was therefore found this architecture increased the accuracy very slightly but decreased memory and the training time. Increasing the number of inception layers allows the network to analyse textures and allows for better intra-class differentiation. It does not generalize better as the tests accuracies are lower than that of the original architecture (0.6%). Given the training time limit was still reaching the upper limits and to reduce memory, focus was turned to understanding how spatial pooling effected the architecture.

Spatial Pooling

One of the main characteristics of the original GoogLeNet was the little number of parameters it used. The average pooling layer, found at the last part of the model, reduced the number of parameters nearly ten fold from the larger fully connected layer which was the source of the majority of the number of parameters in the network. This was inspiration behind this test. There were two average spatial pooling layers. One was found after 6 inception modules which integrated a full connected layer, average pooling and Softmax classification and. The second one was found at the very end of the model - taking an average pool of a 7x7 image output. Here, a 14x14 average pool would have to be taken which seems quite large. This test was aimed at understanding which three methods in 5.22 produced the greatest accuracies whilst reducing memory and time to train the model.

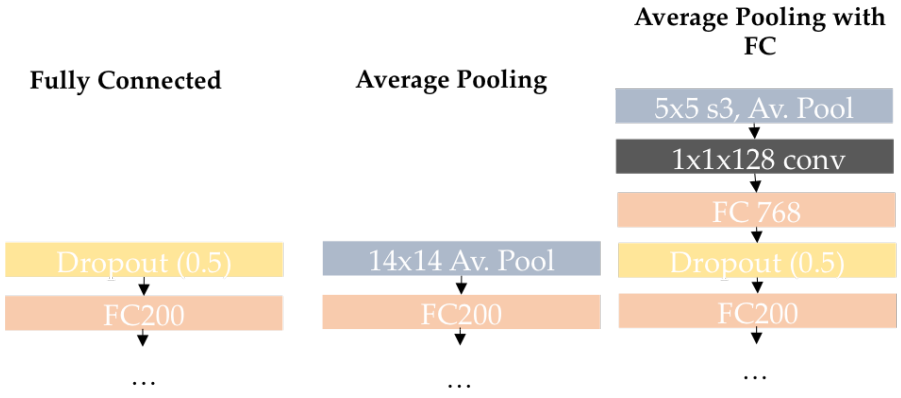


Figure 5.22: *Different Last Layer Connections tested*

	FC Orig.	Average Pooling	Average Pooling with FC
			0
			32,768
			153,600
	0	0	0
	10,035,200	153,600	1,572,864
Num. Parameters	10,035,200	153,600	1,759,232

Table 5.15: The number of parameters for different types of spatial pooling

Figure 5.23 shows three interesting characteristics. Firstly, when average pooling was used, the

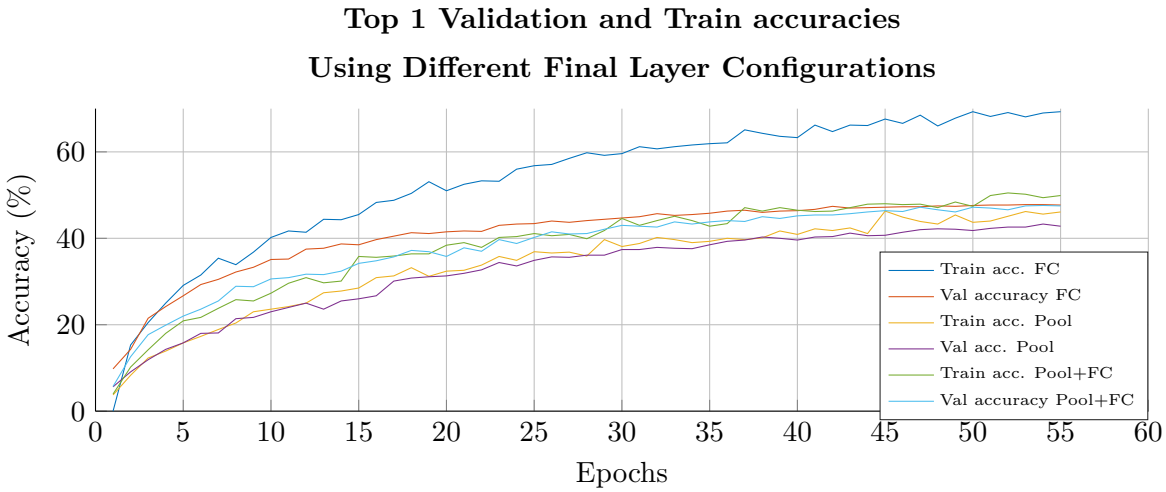


Figure 5.23: *Training and validation accuracies with the 3 different final layer architectures*

	Top 5 Train. Acc.	Top 5 Val. Acc.	$\frac{\text{Time}}{\text{TrainEpoch}}$	Mem. GPU
FC	89.6%	72.80%	716.8s	3918MiB
Av. Pool	77.9%	73.6%	619.8s	3617MiB
Av. Pool+FC	72.8%	70.2%	645.7s	3670MiB

Table 5.16: Memory, complexity, timing and percentage accuracy of the correct prediction appearing in the top 5 predictions

noise increased. Secondly, there is a decrease in gap of accuracies between the training set and the validation set - it seems that underfitting occurs. Finally, the combination of fully connected and pooling layers produced an accuracy that was comparable to the initial design for nearly 250MB less memory and 74 seconds less training time per iteration (an estimated hour less training in total). Specifically, the top validation accuracy was 47.6% and the top training accuracy was 50.5% while producing a 47% test accuracy. This showed that this last layer allowed the model to generalize better than the initial design (by nearly a 1%). The average pooling configuration produced 43.3%, 46.2% top validation and training accuracies and 42% test accuracy. The lower accuracies may be due to the fact the average spatial pooling was placed over a large number of pixels (14x14 compared to 7x7 used in the GoogLeNet). Thus in conclusion, it is better to use a combination of average pooling and fully connected layers for a smaller architecture. However, one needs to pay close attention to the noise of the batch. This could be could be controlled by the learning rate or further increasing the batch size but the memory limitations of the project inhibit this.

The loss functions in figure 5.24 is surprisingly much more smooth than the training scheme. One observation is the convergence to higher losses when using pooling layers in the architecture. This may not be a negative factor as it may mean better generalization.

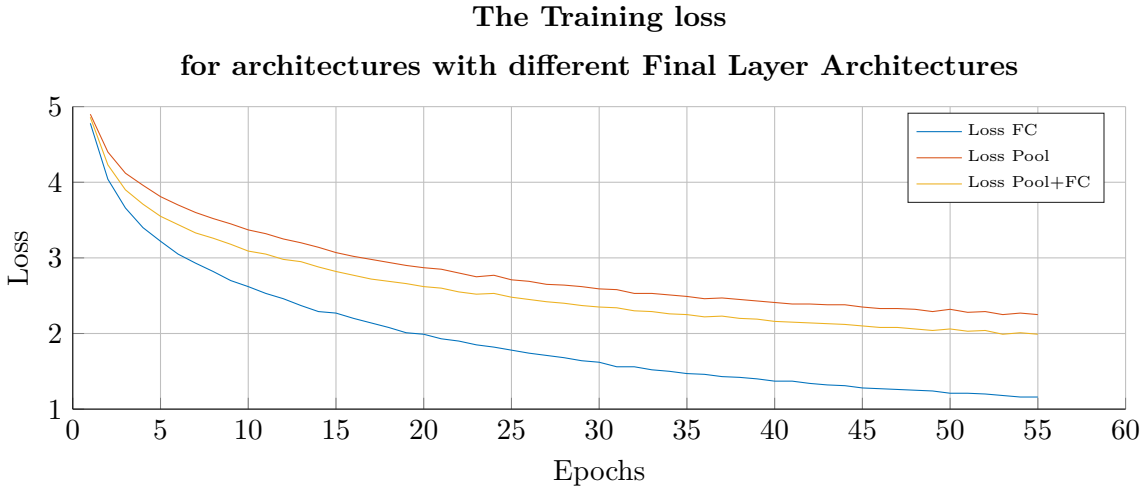


Figure 5.24: *Loss with the 3 different final layer architectures*

5.3 Adapted ResNet

The initial design was catered to be based on the iterations of the VGG Net. One of those iteration though, was increasing the batch size to 512. Whilst this was within the memory limits of the adapted VGG Net, the short-cuts with the residual architecture led to this batch size exceeding it. This could be proved by taking memory used per image in the initially designed adapted Res Net architecture 4.23 of 9.424MB. If there are 512 images per batch, 4.82GB would be used (exceeding the memory limit of the GPU) compared to the 2.4GB of a 256 batch size. Hence a compromise would need to made: to test shallow architectures and retain the advantages of augmentation of a 512 batch size or to build deeper architectures with less augmentation with a 256 batch size. In order to perform the rest of the tests efficiently, this was something to be determined initially. Hence, two architectures were tested: one contained an extra residual module per feature map (for the 256 batch size) than the other. The smallest number of feature maps were taken so that the model did not exceed memory or timing requirements. Figure 5.25 shows the model tested with 512 batch size.

When testing with the batch size of 256, the number of iteration per epoch was increased to 822. This may not have been the same amount of augmentation as the 512 batch size but - but it was still an increase. Also, this number was chosen as it was maximum (with a little margin) before the timing limit was exceeded. The results of the test are shown in figure 6.1.

	Top 5 Train. Acc.	**Top 5 Val. Acc.**	$\frac{\textbf{Time}}{\textbf{TrainEpoch}}$	**Mem. GPU**
Model B512	73.5%	71.65%	684.52s	4050MiB
Model B256	72.1%	71.9%	691.2s	3617MiB

Table 5.17: Memory, complexity, timing and percentage accuracy of the correct prediction appearing in the top 5 predictions

From figure 6.1 it can be deduced the validation accuracies with the two batch sizes were comparable. The model with 512 batch size has a top 1 validation accuracy of 43.95% where as the deeper

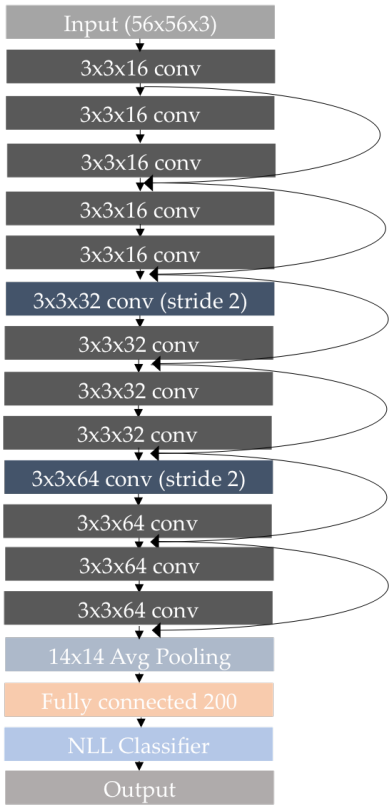


Figure 5.25: *Architecture of model that was tested with batch size 512. The second model was a replica of this one with an extra residual layer per feature map*

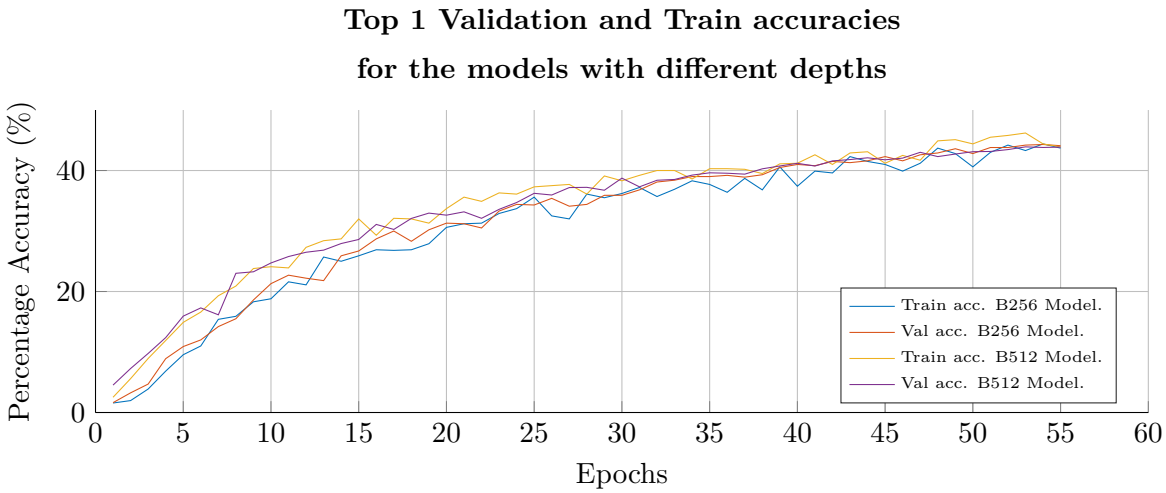


Figure 5.26: *Comparison of train and validation accuracies of the two different models for two different batch sizes*

model with 256 batch size has top-1 validation accuracy of 44.68%. There is not much difference between the training and validation accuracies. the test accuracies were similar too - 40.2% and 41% for batch size 512 and 256 respectively. A similar effect was seen when the average pooling was used with GoogleNet. Furthermore, the training regime is noisy. This may be a characteristic of the archi-

tecture itself. Additionally a similar model was tested with a VGG Net and higher validation accuracy were produced. The reason for this can be accounted for by the 14x14 spatial average pooling: when tested with the GoogleNet, the spatial average with these parameters found to reduce the validation accuracy by nearly 4%. Hence this may be the case here but due to memory limitations, this could not be tested with this model.

Thus in conclusion it can be seen that augmenting the data-set increases the validation accuracy but incrementing the depth using the ResNet shortcut's can lead to the same effects with a smaller batch size and less memory. Hence, a smaller batch size was taken and deeper models were built.

Using the Average Pooling more Efficiently

At the end of both the ResNet and GoogLeNet architectures, a spatial average pooling of 7x7 was used. In the architectures tested for both the ResNet and GoogleNet a 14x14 average pooling was used and found to reduce the validation accuracies from the expected. Hence, an architecture with 3 convolution layers with a stride of 2 was tested to reduce the dimensions to 7x7. Additionally, from the results of the adapted VGG Net, feature maps above 16 perform significantly better. Hence this test was aimed at reducing the depth of the architecture but included more feature maps. The model was constructed as in figure 5.29.

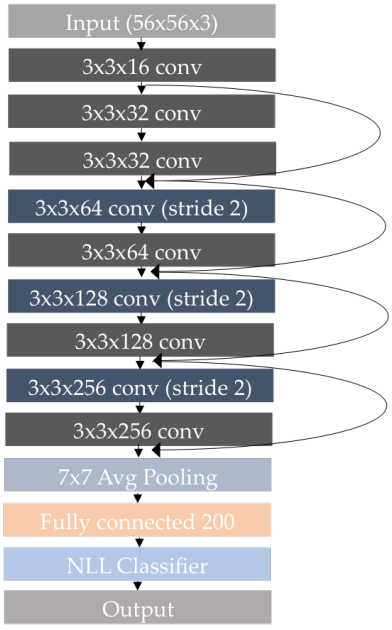


Figure 5.27: *Model with a shallower architecture, 7x7 average pooling and greater number of feature maps*

A learning rate of 0.3 was found to be optimum (by using a course fine approach). The results are presented in figure 6.1

The model produces a top validation accuracy of 50.3% and a top training accuracy of 55.5%. The test accuracy is 49.1%. Therefore, the network generalizes the data much better in comparison. Additionally, a smooth learning curve is produced with minimum loss of 1.72. This verifies that

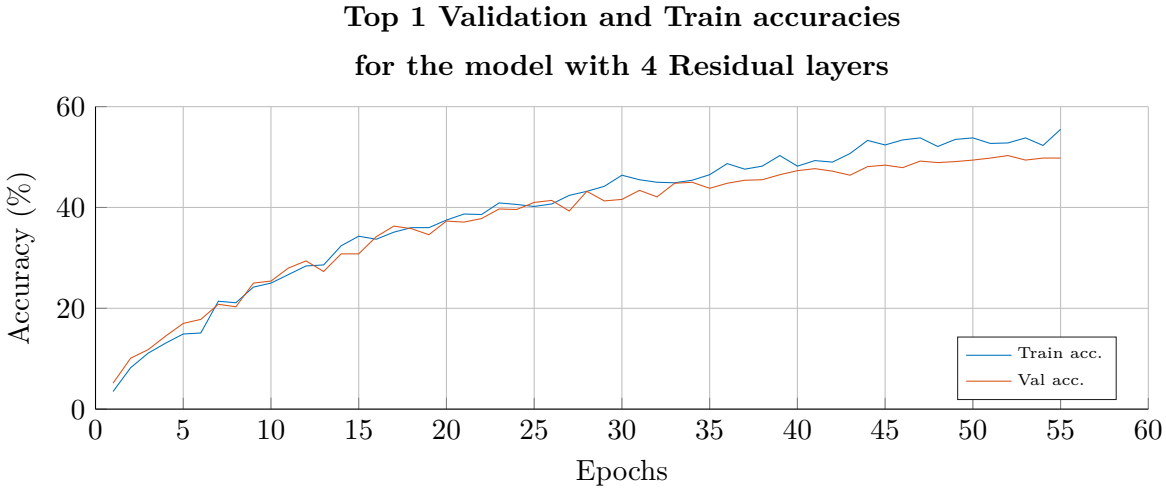


Figure 5.28: *Comparison of train and validation accuracies of the model*

the 0.03 value and step decay were optimally set. The train time is reaching close to 11 hours and around 280MiB remains. Additionally a top-5 error rate of 24% was produced. Once again, there is only around 6% difference to the AlexNet classification accuracy. This also proves that perhaps the Average pooling can be used to efficiently if the receptive field is smaller. This can reduce number of parameters and subsequently the timing to train a model. Additionally, this proves that the fully connected layers are one of the highest sources of overfitting in a network and using average pooling significantly reduces this. Thus shallower architectures should reduce the image size to a suitable size before it reaches the last layer and use average pooling.

	Top 5 Train. Acc.	**Top 5 Val. Acc.**	$\frac{\textbf{Time}}{\textbf{TrainEpoch}}$	**Mem. GPU**
Model	80.1%	76%	697.88s	3722MiB

Table 5.18: Memory, complexity, timing and percentage accuracy of the correct prediction appearing in the top 5 predictions

Additionally, this model produces one of the highest test accuracies out all the architectures tested. Subsequently, it may be worthwhile to examine the failure and success cases to gain further insight of how the classification is performed.

class	**accuracy(%)**
dumbell	5
water jug	6
plumber	9
umbrella	12
wooden Spoon	13

(a) *bottom 5 prediction labels with the original model*

class	**accuracy(%)**
goldfish	78
European Fire Salamander	81
bison	81
gondola	81
dugong	82

(b) *Top 5 prediction labels with original model*

There are different failure cases in this class compared to the other architectures. Including the

introduction of the dumbbell and water jug. The next closest class to the the water jug is a beaker (14% accuracy) and a bucket (10% accuracy) which indicates that the model is struggling with some intra-class variations of objects containing water. The wooden spoon and the umbrella still appear suggesting that the network may be struggling with clutter. If the best 5 cases are considered, there are some classes not seen with the best performing GoogLeNet and the VGG Net such as the Salamander. Some of the salamanders in the validation images have very similar textures to the background - they are camouflaged. This provides further evidence that increase the number of feature maps - allows for a finer differentiation of textures.

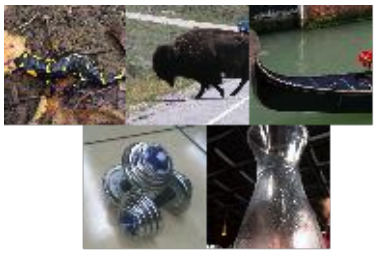


Figure 5.29: *Top row: new success cases (Salamander, bison and gondola), bottom row new failure cases (dumbbell and water jug)*

The bison and gondola usually form a large part of the image - and hence may explain why they are recognised with more frequency. It can be concluded that increasing the number of feature maps allows finer textures to be identified. Furthermore, by comparing the deeper architecture tested in the previous experiment and accounting for the 2-3% reduction in accuracy due to a larger average pooling layer used, feature maps are as important as increasing depth. A deeper model can be built but if it has a low number of feature maps (16), validation accuracies may not be as high.

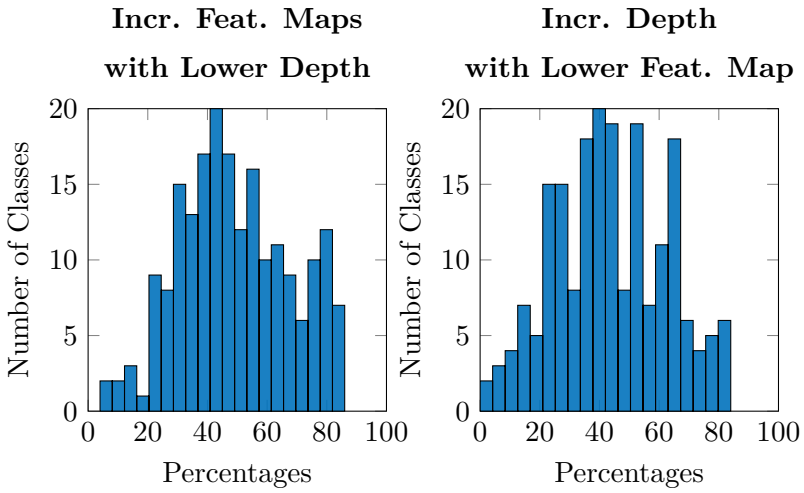


Figure 5.30: *Class distribution of model with increased number of feature maps per model and model tested with increased depth but reduced number of feature maps per each layer*

Figure 5.30 shows that class distribution accuracies. For a lower number of feature maps but increased depth, there more classes predicted with a lower accuracies. Whereas the class distribution for the lower depth but greater number of feature maps has higher percentage per class. Thus verifying For shallower architectures, a trade off needs to be made between a number of feature maps and increase in depth.

Chapter 6

Evaluation

Three architectures that have won the imageNet challenge were adapted. Varying iterations were produced based on the results observed while testing. It was found the augmenting data of low resolution images in shallower architectures results in nearly 3% increase in validation accuracies. The augmentation was performed using a variation of ten crops as colour jittering via PCA was found to take too long and memory in training. A major limitation of certain architectures is the initializations - adding the newly developed technique of batch normalization helped counteract this problem, and presented a further increase in validation accuracies (around 2% with the VGG Net). For the most recent architecture, ResNet, it eliminated the need of even using dropout. Thus for low memory architectures, batch normalization should be applied even if it increases complexity and training time by 9%.

The validation accuracies across the three models show the the VGG Net and the ResNet have the highest top-5 error rates of 24%. This is 6% lower than the AlexNet thereby proving that these architectures can classify images efficiently in relative short time periods. On closer analyses, nearly all three models struggled with fine-grained classes - particularly where clutter in the images was present.The validation and test images show that the reasoning could be that, with low resolution images which contain low interest regions, it is difficult to discern the key features that separate a class. An example of this was discovered with the VGG Net where an umbrella and a sombrero were misclassified. If the pole of an umbrella was more clearly visible that may help separate the two classes.

Increasing the number of feature maps though extracts more information particularly of texture. This helps with the differentiation of fine-grained classes such as types of butterfly or frogs. However, as the analysis of the VGG Net showed, there seems to be a convergence of the number of feature maps. Increasing the number of feature maps above a certain value may not lead to added benefits that can compensate for the extra memory usage. Thus a shallower architecture, for which memory is limited, must find the optimum number of feature maps for a given depth. The classes classified with the best accuracy nearly all had the largest regions of interest of the 64x64.

Comparing the architectures of the structure, it is found that the best iteration of the adapted GoogLeNet inception architecture has the lowest validation accuracies of the tree models. This may

have been for two reasons: the 5x5 filters used are too large for low resolution images: its convolution reduces the spatial content and perhaps too much for the input of low resolution images. The accuracy is found to increase when an extra inception module replaced a convolution layer indicating spatial averaging does have a positive effect on low resolution images. However, due to memory limitations, only three inception layers could be built into the model (compared to the near 20 of the original GoogLeNet). The combination of average pooling and fully connected layers found to increase the accuracy and reduce both time and memory used. Despite this, the validation accuracies were not as high as the other architectures.

The simpler adapted VGG Net, produced results comparable to the adapted ResNet. However, a larger batch size was used and thus more augmentation. The ResNet's extra memory requirements, due to the short cut layer, inhibited such a large batch size to be used for deeper models. A compromise was made to reduce the batch size but increase depth. It was found to have comparable results with a shallower architecture with a larger batch size (512) but it used 222 more iterations (to still retain the benefits of some augmenting) and a higher learning rate - making the training iterations noisier. Too much noise, would mean very little convergence as the model weight's would fluctuate. As a result, lower batch sizes, may not be able to perform as well. This means that a careful trade off is usually needed between augmentation, batch size, and depth of a network. Ideally, more experiments would be performed to find this optimum relationship.

Due to the time limitations an architecture with reduced depth was tested but increased number of feature maps. It was found to increase the accuracies to that of the best performing VGGnet. However, it used nearly 900MiB more memory and a longer time to train. The average pooling layer helped improve generalization and thus the test and validation accuracies and reduced the number of parameters used in a VGG Net. The key downfall was the memory used for a single image to be trained on the network. On the other hand, thr key downfall of the VGG Net was that increasing depth did not result in increasing the validation accuracy because of the reasoning described [8]. Hence it is important that the memory of the ResNet, is reduced. One way of reducing this would be possible finding another method to decrease the dimensionality of the images in the short-cut connections. One method is to use padding instead.

By analysing the class distributions, of the best performing iteration of each model, several things can be discerned: The adapted ResNet predicts less classes with a less percentage of accuracy compared to the other networks. The number of classes predicted with high classification accuracy is comparable to that of the VGG Net. The Googlenet, has a more centred distribution - predicting a majority of classes with 35%-60% accuracy. This indicates that it may classify some fine-grained classes to an equal extent as the coarse classes.

	VGG Net	**GoogLeNet**	**ResNet**
Test accuracies (%)	47.4	47	49.1

Table 6.1: maximum test accuracies across the three models

Thus in conclusion, the adapted ResNet inspired architecture with increased number of feature

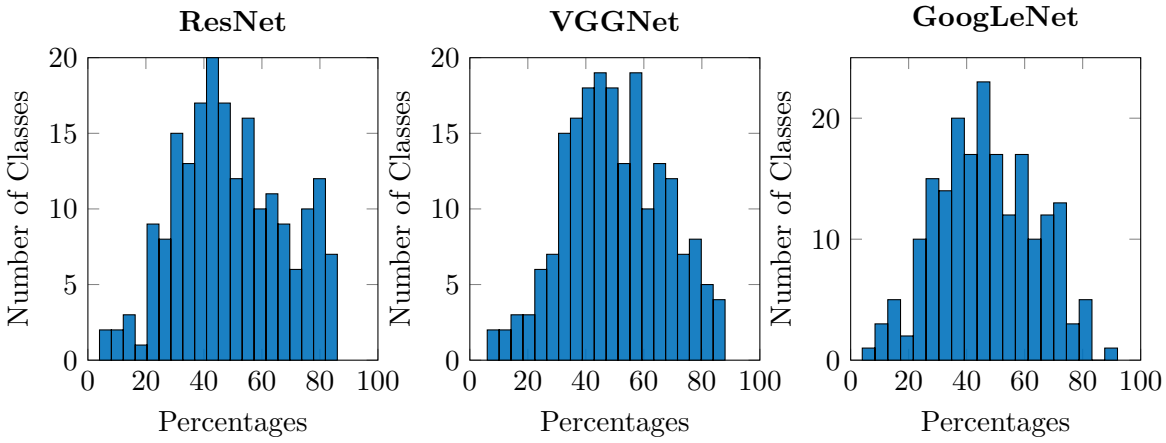


Figure 6.1: *Class accuracy distributions for the best performing architectures from different models*

maps was the best performing network and produced validation error rates comparable with the AlexNet. However, methods to reduce the memory need to be addressed. The hyper-parameters that were changed were the batch size, number of iterations and depth. A fine compromise needs to be found between the three. If more train timing was available, a greater amount of augmentation could be used and it is likely even greater validation accuracies could be achieved.

Chapter 7

Conclusions and Further Work

7.1 Future Work

It was found that ResNets can use limited memory (3722MiB) and limited number of parameters (2M) to classify low resolution images with top-5 validation accuracy of 76%. However much work is still left to determine the optimum relationships between the hyper parameters in the model. Finding relationships between depth augmentation (batch size) and feature maps would be key to further increasing accuracy. Additionally other methods to further reduce the memory used by the ResNet could be found particularly to do with the identity functions. Bottelneck architectures could also be implemented [8]. Although the GoogLeNet adaptation had the worst test performance, different filter sizes could be tried in the first layer to see if this can increase validation accuracy. Another optimization technique should also be implemented - Nestrov Momentum [61]. It is found to have much success and this may speed convergence and produce higher accuracies.

Additionally, data sets with more fine grained classes could be tested on this adaptation of ResNet to see if it generalizes well. Also, Transfer Learning - a technique where only a part of already pre-trained network is trained - could be explored [64]. However, this method does not ensure reduction in the number of parameters and memory use that is needed for embedded systems.

There is also still many low level optimisations that could be implemented to further reduce the memory so that CNNs so that it can be used on embedded systems. For instance, compressing deep neural networks looking at vector quantization [12] and possibility of using lower precision bits [65].

7.2 Conclusion

Deep learning models were designed to classify low resolution images and be implemented on lower memory GPUs. This work's main contribution found that certain models perform better in classification of such images. In particular simpler stacked architectures such as the adpated ResNet produced the best validation error rate of 24% which is comparable to the award winning AlexNet. This project found that increasing the number of feature maps was as important as increasing the depth of network. This is different to the direction in which stae-of-the art CNNs are heading: where depth has been increased exponentially for every architecture. It was also found that augmentation

increases validation accuracy by 3%-4% and is controlled by the batch size as this determines the amount of memory allocated on the GPU. An optimum batch of 256 with larger number of iterations per epoch produced comparable, though nosier, results than a 512 batch size with less iteration per epoch. Additionally newly developed techniques such as batch normalization and combining fully connected layers with average spatial pooling should be used instead of fully connect layers. This reduces the number of parameters by nearly a tenth and provides better test accuracy due to their nature of underfitting. Thus in the future, a fine compromise between augmentation, number of feature maps, depth and memory needs to be found when classifying low resolution images. This project is one step closer to implementing better accuracy architectures to be trained on embedded systems and making architectures like these increasingly available in daily life.

Bibliography

[1] Linda B Smith. Learning to recognize objects. *Psychological Science*, 14(3):244–250, 2003.

[2] Matthew D Zeiler. *Hierarchical convolutional deep learning in computer vision*. PhD thesis, NEW YORK UNIVERSITY, 2013.

[3] Stanford University. Tiny imagenet data set. `http://cs231n.stanford.edu/project.html`. Accessed: 2016-01-01.

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

[5] Yann LeCun, Koray Kavukcuoglu, and Clément Farabet. Convolutional networks and applications in vision. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 253–256. IEEE, 2010.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[7] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1026–1034, 2015.

[10] Nvidia. Gpus, geforce btx black titan. `http://www.geforce.co.uk/hardware/desktop-gpus/geforce-gtx-` Accessed: 2016-02-20.

[11] Zhangyang Wang, Shiyu Chang, Yingzhen Yang, Ding Liu, and Thomas S Huang. Studying very low resolution recognition using deep networks. *arXiv preprint arXiv:1601.04153*, 2016.

[12] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

[13] Marion Chevalier, Nicolas Thome, Matthieu Cord, Jérôme Fournier, Gilles Henaff, and Elodie Dusch. Lr-cnn for fine-grained classification with varying resolution. In *Image Processing (ICIP), 2015 IEEE International Conference on*, pages 3101–3105. IEEE, 2015.

[14] Clément Farabet, Berin Martini, Polina Akselrod, Selçuk Talay, Yann LeCun, and Eugenio Culurciello. Hardware accelerated convolutional neural networks for synthetic vision systems. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 257–260. IEEE, 2010.

[15] Sebastien Roux, Franck Mamalet, and Christophe Garcia. Embedded convolutional face finder. In *Multimedia and Expo, 2006 IEEE International Conference on*, pages 285–288. IEEE, 2006.

[16] Seyyed Salar Latifi Oskouei, Hossein Golestani, Mohamad Kachuee, Matin Hashemi, Hoda Mohammadzade, and Soheil Ghiasi. Gpu-based acceleration of deep convolutional neural networks on mobile platforms. *arXiv preprint arXiv:1511.07376*, 2015.

[17] Daisuke Miyashita, Edward H Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*, 2016.

[18] Xingchao Peng, Judy Hoffman, Stella X Yu, and Kate Saenko. Fine-to-coarse knowledge transfer for low-res image classification. *arXiv preprint arXiv:1605.06695*, 2016.

[19] Artem Vasilyev. Cnn optimizations for embedded systems and fft.

[20] Konstantinos Plataniotis and Anastasios N Venetsanopoulos. *Color image processing and applications*. Springer Science & Business Media, 2013.

[21] Stanford. Convolutional neural networks for visual recognition. `http://cs231n.github.io/linear-classify/`. Accessed: 2016-01-20.

[22] Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[23] University of Washington Neuroscience. Brain, facts and figures. `https://faculty.washington.edu/chudler/facts.html`. Accessed: 2016-01-12.

[24] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

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

[26] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[27] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[28] Markus Oberweger, Paul Wohlhart, and Vincent Lepetit. Hands deep in deep learning for hand pose estimation. *arXiv preprint arXiv:1502.06807*, 2015.

[29] Jordi Sanchez-Riera, Yuan-Sheng Hsiao, Tekoing Lim, Kai-Lung Hua, and Wen-Huang Cheng. A robust tracking algorithm for 3d hand gesture with rapid hand motion through deep learning. In *Multimedia and Expo Workshops (ICMEW), 2014 IEEE International Conference on*, pages 1–6. IEEE, 2014.

[30] Christian Szegedy, Alexander Toshev, and Dumitru Erhan. Deep neural networks for object detection. In *Advances in Neural Information Processing Systems*, pages 2553–2561, 2013.

[31] Ivan Laptev, Marcin Marszałek, Cordelia Schmid, and Benjamin Rozenfeld. Learning realistic human actions from movies. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[32] Piotr Dollár, Vincent Rabaud, Garrison Cottrell, and Serge Belongie. Behavior recognition via sparse spatio-temporal features. In *Visual Surveillance and Performance Evaluation of Tracking and Surveillance, 2005. 2nd Joint IEEE International Workshop on*, pages 65–72. IEEE, 2005.

[33] Moez Baccouche, Franck Mamalet, Christian Wolf, Christophe Garcia, and Atilla Baskurt. Sequential deep learning for human action recognition. In *Human Behavior Understanding*, pages 29–39. Springer, 2011.

[34] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, 2014.

[35] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 815–823, 2015.

[36] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[37] Sanjeev Arora, Aditya Bhaskara, Rong Ge, and Tengyu Ma. Provable bounds for learning some deep representations. *arXiv preprint arXiv:1310.6343*, 2013.

[38] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[39] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

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

[41] James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian Goodfellow, Arnaud Bergeron, et al. Theano: Deep learning on gpus with python. In *NIPS 2011, BigLearning Workshop, Granada, Spain*, 2011.

[42] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. Lua-an extensible extension language. *Softw., Pract. Exper.*, 26(6):635–652, 1996.

[43] Facebook. Imagenet winners benchmarking. `https://github.com/soumith/convnet-benchmarks`. Accessed: 2016-02-115.

[44] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[45] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[46] NVIDIA. Nvidia cudnn gpu accelerated deep learning. `https://developer.nvidia.com/cudnn`. Accessed: 2016-02-09.

[47] NVIDIA. Embedded systems- jetson tx1. `http://www.nvidia.com/object/jetson-tx1-module.html`. Accessed: 2016-06-07.

[48] Amazon Web Services. Specifications of an ec2 instance. `https://aws.amazon.com/ec2/instance-types/`. Accessed: 2016-02-15.

[49] Facebook. Training an object classifier in torch-7 on multiple gpus over imagenet. `https://github.com/soumith/imagenet-multiGPU.torch`. Accessed: 2016-03-01.

[50] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images, 2009.

[51] Sebastian Raschka. About feature scaling and normalization – and the effect of standardization for machine learning algorithms. `http://sebastianraschka.com/Articles/2014_about_feature_scaling.html`. Accessed: 2016-01-01.

[52] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[53] Ken Chatfield, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Return of the devil in the details: Delving deep into convolutional nets. *arXiv preprint arXiv:1405.3531*, 2014.

[54] Alfredo Canziani. Machine learning with torch - pca. `https://github.com/Atcold/Machine-learning-with-Torch/blob/master/PCA/README.md`. Accessed: 2016-05-01.

[55] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256, 2010.

[56] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[57] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30, page 1, 2013.

[58] Yichuan Tang. Deep learning using linear support vector machines. *arXiv preprint arXiv:1306.0239*, 2013.

[59] Léon Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*, pages 421–436. Springer, 2012.

[60] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th international conference on machine learning (ICML-13)*, pages 1139–1147, 2013.

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

[62] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[63] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[64] Yoshua Bengio. Deep learning of representations for unsupervised and transfer learning. *Unsupervised and Transfer Learning Challenges in Machine Learning*, 7:19, 2012.

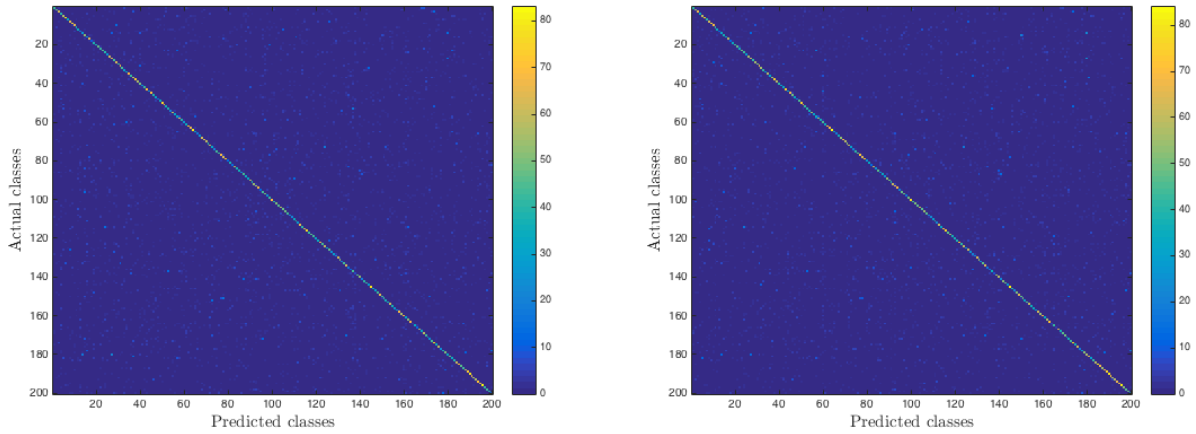
[65] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *arXiv preprint arXiv:1502.02551*, 2015.

Appendices

.1 Source Code

can be found at https://github.com/h912/FYPR

.2 VGG Net testing



(a) *Model A validation Confusion Matrix* (b) *Model B validation confusion matrix*

Figure .1: *Validation Confusion matrix for Model A and Model B. Note the increased in accurate predictions (blue dots) around the diagonal*

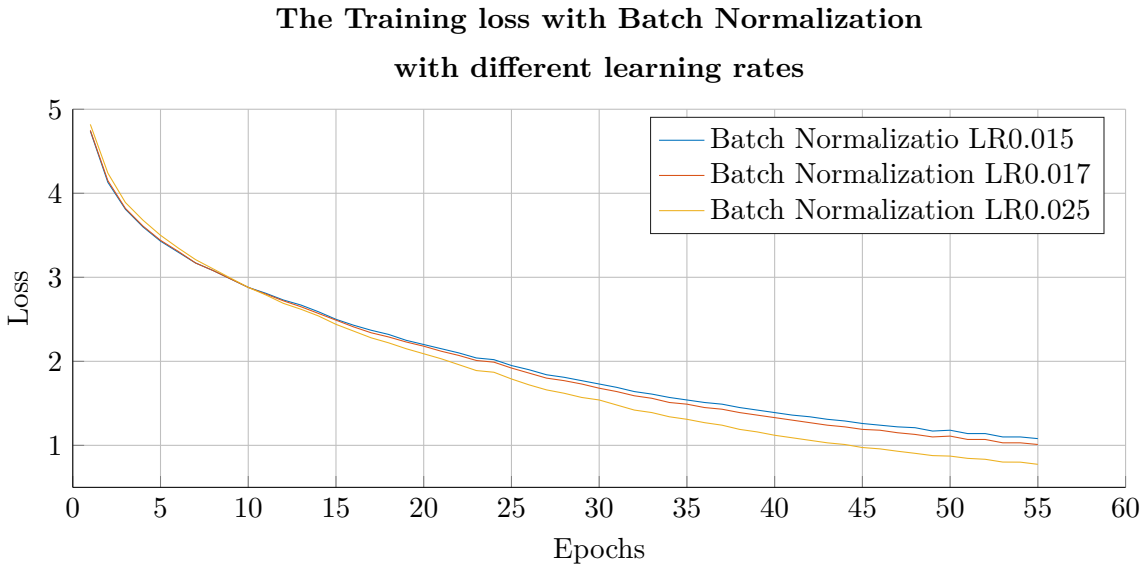


Figure .2: *Batch Normalization applied before ever ReLu layer and different learning rates applied*

(a) *Weights of 1st layer of original model* (b) *Output of 1st layer of original model*

Figure .3: *Weights and output of butterfly image in first layer of the original model*

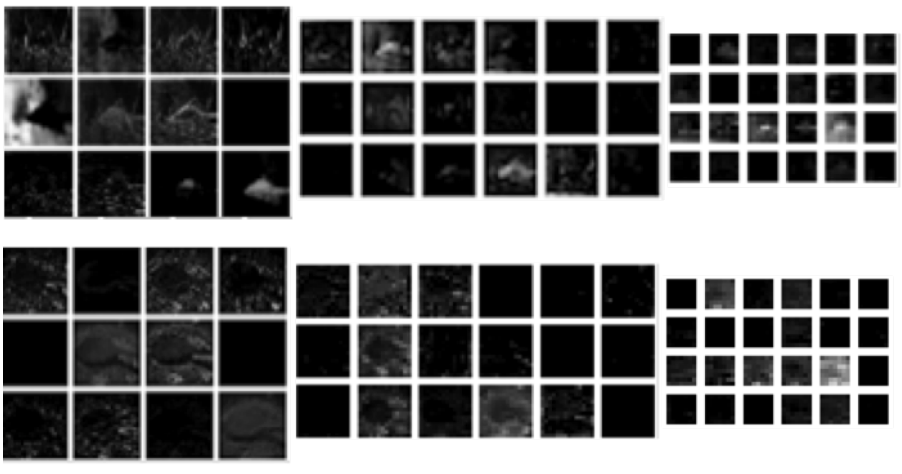


Figure 7.4: *Feature maps for three layers of a goldfish and wooden spoon of the model A*

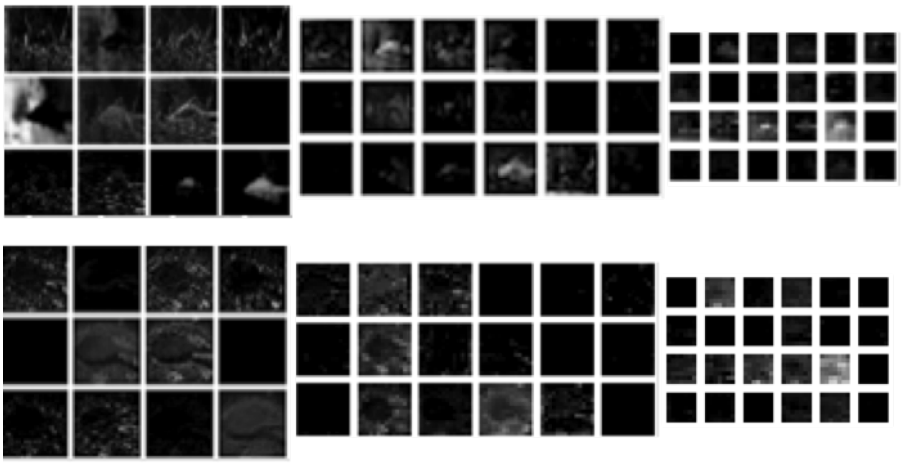


Figure 7.5: *Feature maps for three layers of a goldfish and wooden spoon of the original model*

7.3 GoogleNet testing

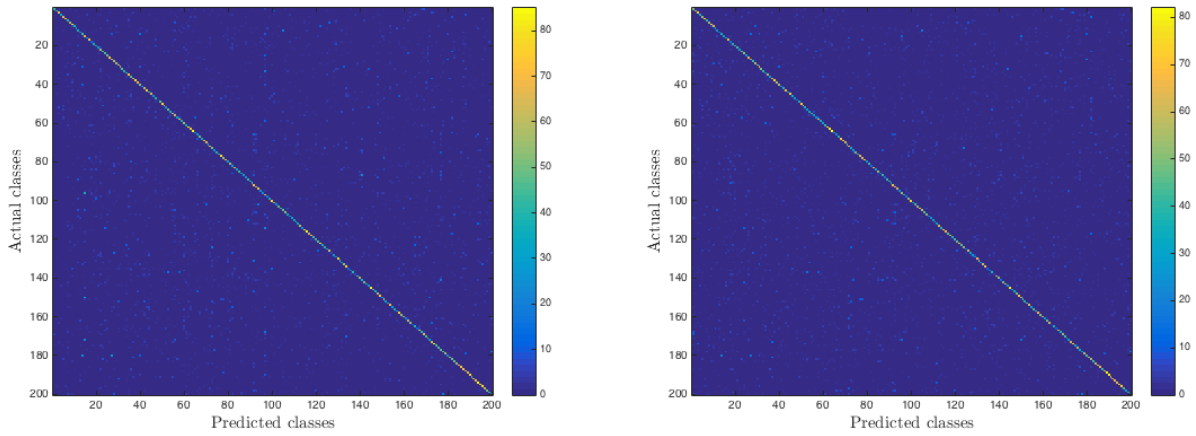
Hyper-parameter optimization

Learning Rate	Training accuracy	Validation accuracy	Cost
0.09	0.97	1.01	5.28
0.045	1.54	1.64	5.29
0.025	1.43	2.20	5.26
0.02	1.97	2.01	5.25
0.017	2.18	2.49	5.25
0.015	1.56	2.37	5.25
0.012	1.01	1.36	5.27
0.01	0.79	1.23	5.27
0.001	0.00	0.7	5.29

Table 7.1: Table showing course-fine approach to learning rate

Learning Rate	Training accuracy	Validation accuracy	Cost
0.01984	1.12	1.43	5.23
0.01907	2.02	2.00	5.22
0.01777	1.97	2.67	5.22
0.01700	1.98	2.34	5.22
0.01658	1.67	2.41	5.22
0.01602	1.23	2.43	5.22
0.01578	1.55	2.21	5.22
0.01500	1.43	2.36	5.22
0.01440	1.78	1.97	5.22
0.01407	1.32	1.10	5.22

Table 7.2: Table showing random search of learning rate



(a) *Validation Confusion Matrix, extra incept.*

(b) *orginal architecture*

Figure 7.6: *Validation Confusion matrix for Googlnet models (extra incpt and orginal model)>. Note the increased in accurate predictions (blue dots) around the diagonal*

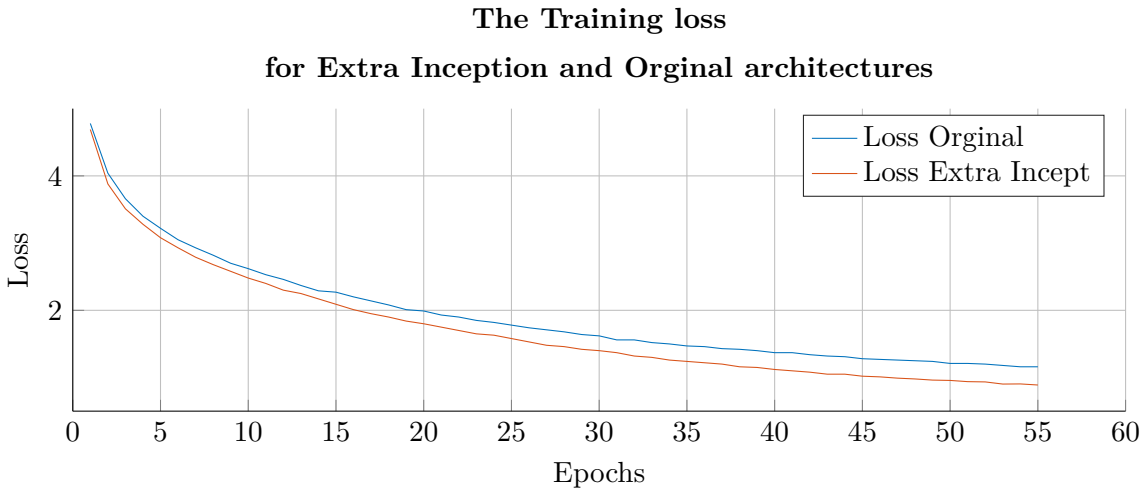


Figure 7.7: *Loss of the extra inception layer architecture and original architecture. Note the greater loss for the extra inception layer architecture*

7.4 ResNet testing

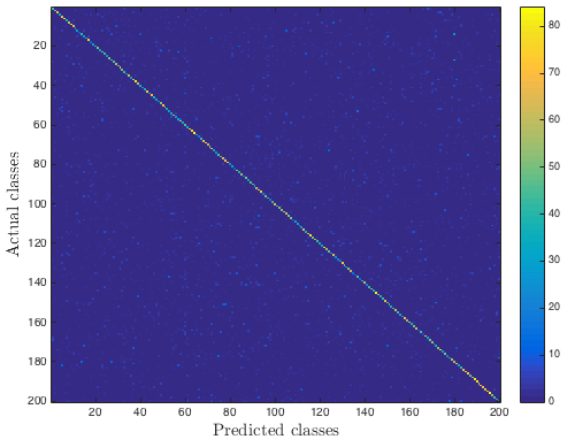


Figure 7.8: *Validation Confusion Matrix, extra incept.*

Figure 7.9: *Validation Confusion matrix for ResNet model with 4 residual layers*

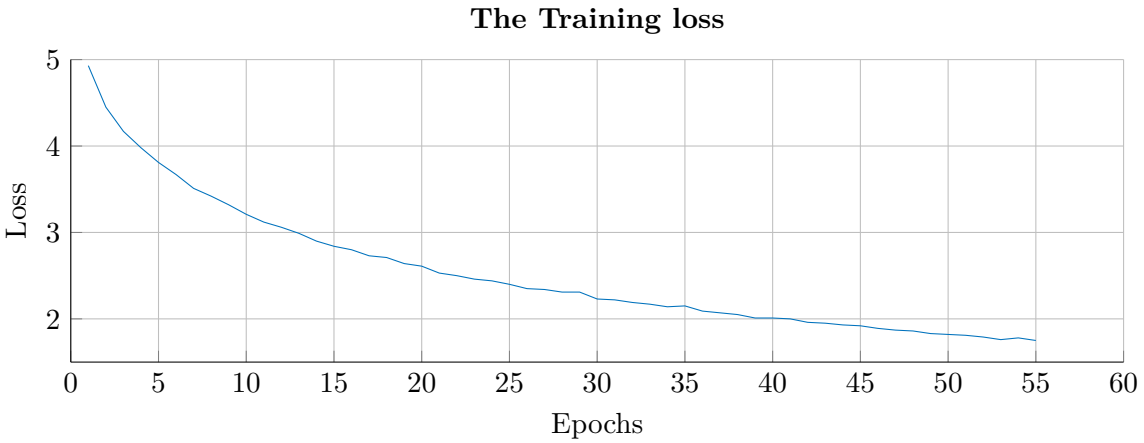


Figure 7.10: *Loss of the increased feature map resnet*