DeepDoctor: Neural Networks in Medical Diagnostics

Chris Kapp
13816740

BSc (Hons) Computer Science

Abstract

This project explores the power of machine learning, in particular convolutional neural networks, to assist doctors in patient diagnosis. I test and compare the performance of several convolutional neural network architectures on a data set of chest x-rays. I present the viability of these models as a computer aided diagnostic (CAD) tool, and the correlations with these models' performance in existing computer vision tasks.

The models are implemented in Python[34] using the TensorFlow[1] library, with pretrained models coming from the TF-slim[40] model library. I train the networks using a single NVidia GTX1080Ti. The patient x-rays are entirely anonymised and sourced from the US National Institutes of Health (NIH) [46]. This data set is also referred to as the Chest X-Ray 14 (CXR14) data set.

Acknowledgements

Firstly I would like to acknowledge and thank Dr Ran Song, my supervisor, for his support and guidance throughout the project, particularly when interpreting the results of my experiments.

I would also like to acknowledge and thank Dr Karina Rodriguez Echavarria, my second reader, for her support and guidance during the earlier phases of my project and helping me to direct my research to a real world application.

Finally I would like to acknowledge and thank Kevin Kapp for his love and support during the difficult times in my project and acting as a sounding board to help me develop my ideas.

Contents

1.0 Introduction

In 2012, Krizhevsky et al. [22] entered the ImageNet Large Scale Visual Recognition Challenge (ILSVRC[38]) with a convolutional neural network architecture called AlexNet. This revolutionary entry achieved a top-5 error rate of just 15.3%, compared to the previous winners at over 25%. This sparked research into deeper convolutional neural network architectures and in just a few years the state of the art now stands at under 5%.

The chest x-ray is one of the most widely available medical tests and is cheap to perform. However skilled radiologists are in short supply[36]. In addition to this, chest x-rays can be difficult to read and extract appropriate diagnoses[5]. A convolutional neural network capable of suggesting likely diagnoses with a reasonably high degree of confidence could dramatically improve workflow for radiologists. Consequently, health services could reduce overtime and outsourcing costs, and improve patient care[37].

1.1 Aim

As an extension of my artificial intelligence modules CI213 and CI342, this project aims to apply several of the existing deep convolutional neural network (DCNN) models to the problem of diagnosing chest x-rays (CXR). The recent advancements in computer vision with DCNN models provide a solid baseline, and I intend to see how useful they can be in a more specialised domain. I will look at several well known models that have performed well on the ILSVRC2012 verification data set: VGG, GoogLeNet (Inception), ResNet, and InceptionResNet. Each of these networks has previously held state of the art performance and should be useful subjects for exploring any correlation between performance on ILSVRC and other data sets, and therefore how generalised this approach is.

1.2 Objectives

I have defined a number of objectives that I wish to complete during the project, necessarily limited by resource and time constraints:

1. To research and understand the theory behind DCNNs, in particular, the test subjects
2. To implement these networks from scratch in TensorFlow
3. To compare the performance with a model pretrained on ImageNet
4. To experiment with hyperparameter tuning
5. To evaluate each models performance against a fixed training set of data

This should allow:

1. Determination of the viability of a CAD tool that relies on DCNNs
2. Comparison of the strength of different architectures on CXR14
3. Determination of any correlation between ILSVRC and CXR14 performance
4. Highlight any issues or limitations of the approach and how they might be overcome

Additionally, the most successful model can be used as a demonstration tool.

1.3 Scope

I chose this project as I feel that it is something with a very strong real world potential. However, given the limited time constraints of the project, I have had to restrict myself to one area of diagnosis from medical imaging. As such I will not be looking into the efficacy of DCNNs with computed tomography (CT) or magnetic resonance image (MRI) scans, although this is definitely an area for further research if a large enough sample of anonymised data could be obtained.

However there is a large set of publicly available CXR data so I will be focusing on that, augmenting the images as necessary, for example pretrained models require RGB input as opposed to greyscale.

2.0 Project Methodology

2.1 Approach

I wanted to run several tests on each model, with differing hyperparameters, so that I could look at wider result sets. Since I was already familiar with Python, but the TensorFlow library was new to me, I decided that the best approach would be an iterative approach like Agile. As the project was not for a specific client, the effectiveness of a given model's design would be based purely on how well it performed on the testing subset of the data. Any changes made to the model would result in a change in this performance metric after training that would determine which set of hyperparameters worked best for that model. Once this was done for each model the best performing model could then be used to continue to tune further hyperparameters iteratively to see what is the best performance that could be achieved.

Additionally, I could set clear deadlines for each model which would be important due to the significant computation time required to train a DCNN. Even with the CUDA[31] acceleration of TensorFlow, and the reasonably powerful hardware I had access to, a single model could take anywhere from 6-18 hours to train. This meant that multiple training runs for differing hyperparameters on a single model could quite easily consume an entire week.

In terms of source control, I had to deal with this manually as websites like GitHub have a file size limit of 100MB[8]. The model checkpoints that store the learned weights would range between 30MB to 1.6GB depending on the model complexity. Therefore I decided on taking a daily manual backup on an external hard drive.

2.2 Technology

The main technology used during the project was to be TensorFlow due to it's powerful CUDA acceleration and Python interface. Additionally, after Google open-sourced the TensorFlow library it has quickly become the most popular deep learning framework in academic papers[20] and on GitHub with nearly 100,000 stars to Keras' 29,000[16]. Notably, Keras is actually a framework for simplified learning that runs on top of another library such as TensorFlow, and the fourth most starred repository is a compilation of tutorials and examples for TensorFlow.

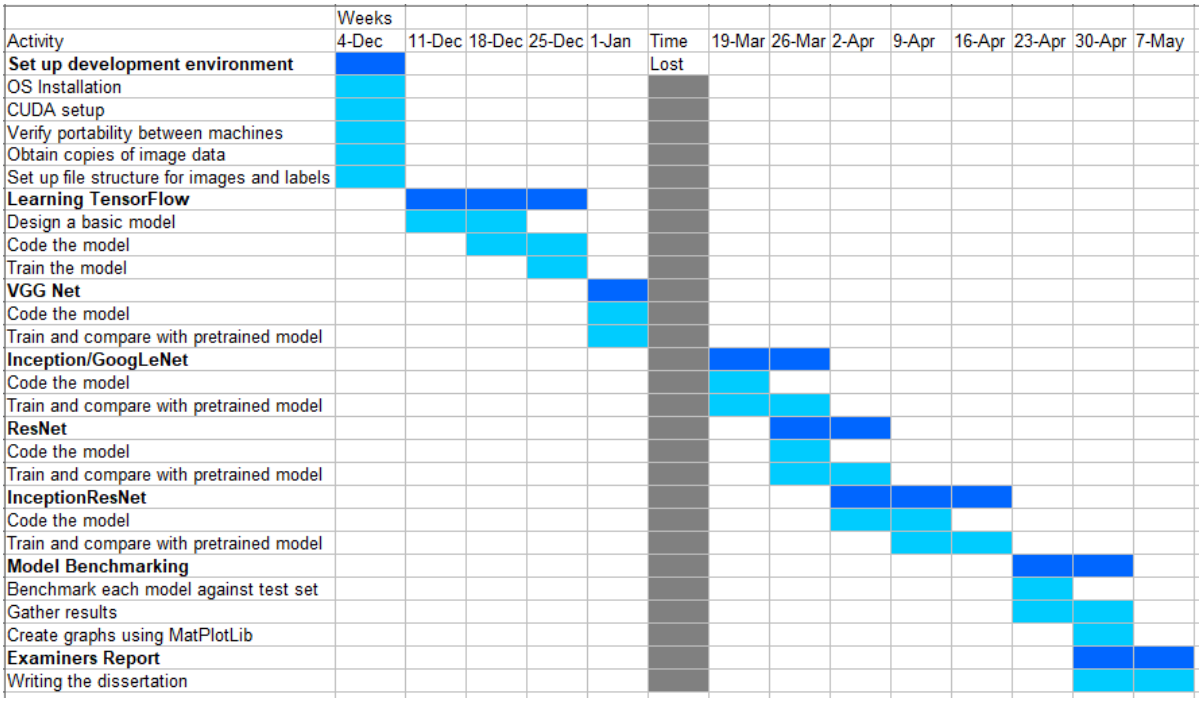
The TensorFlow library also has very efficient implementations for lots of common functions used in various DCNN models. This significantly simplifies development allowing experiments to be conducted much faster than would otherwise be possible.

2.3 Schedule

To begin in the project I drew up a Gantt chart to allocate my time in the initial project proposal. However this changed for two reasons:

1. At the early review meeting with my second reader we decided that my initial proposal, looking at content based image retrieval (CBIR), was not focused enough on a specific goal so I reorganised the plan for the revised project.

2. I lost a significant amount of time in the second semester (Jan-Mar) due to unforeseen personal circumstances requiring the plan to be condensed further.



With this I was able to focus on a specific area of the project each week. Most of my research had been done prior to December as my initial project also intended to focus on DCNNs so the change of project was not too problematic. The discussion primarily involved altering my efforts to target a more specific use case and end goal, which I think this project accomplishes. A CBIR system has a lot of potential use cases, but I was indecisive on a specific goal which would have made the project difficult to complete. In contrast, this project has a discrete level of accuracy that can be measured to verify my success.

Each week I was able to clearly define my goals for that agile cycle and make note of things that were taking more or less time than estimated, so that I could factor that in to future models that I would be building. The remaining time could then be used for further research, this happened a lot in December as it took a lot less time than anticipated to familiarise myself with the TensorFlow library. Despite the issues in the beginning of the second semester, I think this approach was critical to my projects success as the 'micro-goals' helped me to reinforce exactly where I was with the project at any given time, thus I was able to pick up where I left off very quickly.

3.0 Background Research

This section is a summary of the background research I undertook during the project; in it I explain the terms and architectures used during the project and why they were chosen.

3.1 Theory and Mathematics

3.1.1 Neural Networks

A Neural Network is, at it's core, an attempt to model the human brain. A neural network is described as a set of connected neurons in several layers that take some inputs, x, perform some calculation, f, and produce an output, y. Such that

$$y = f(x)$$

Therefore, there will be a set of functions, $\mathbb{F}$, that map the inputs, x, to the desired outputs, y. One member of the set, f', will be the optimal f for the equation $y = f(x)$ that provides the most correct outputs for the given inputs. The objective is to find f', by solving the equation for f.

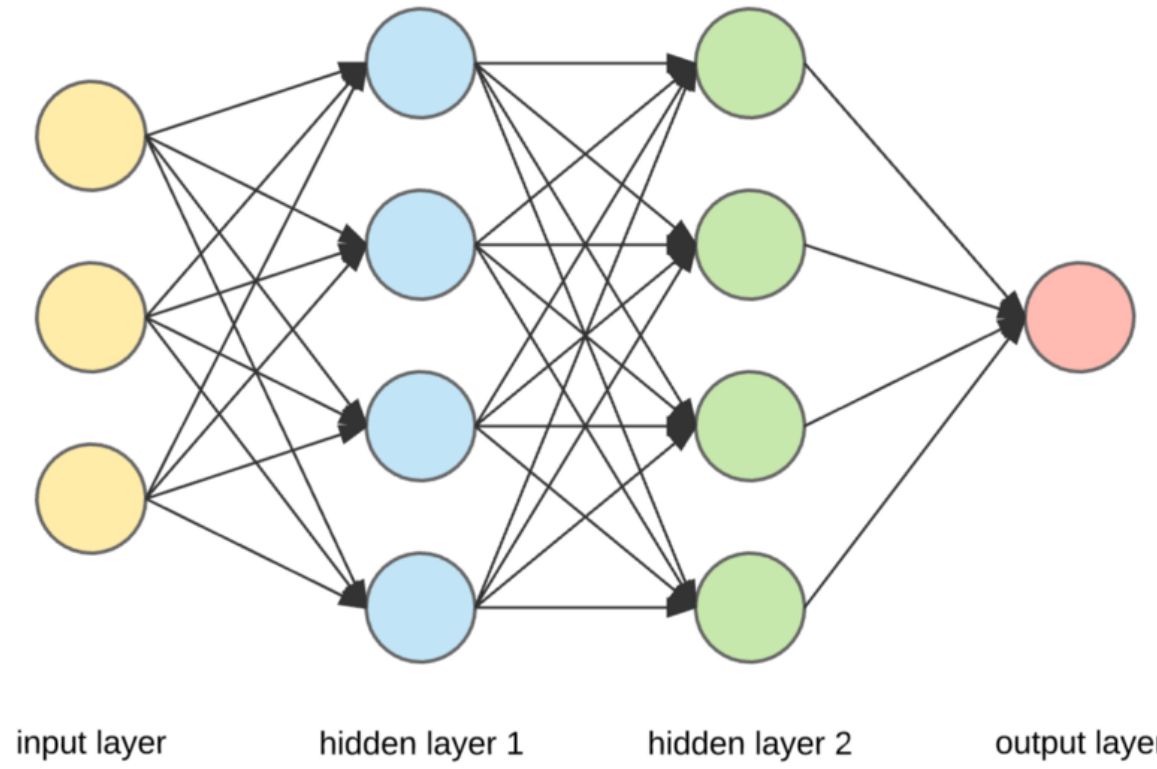


Figure 3.1: Artificial Neural Network, [7]

As seen in Figure 3.1, each circle or 'node' on the graph represents a neuron. The network takes some matrix of inputs, x, represented by the input layer. Then each neuron in next layer performs a calculation based on its inputs from the previous layer, represented by the arrows. In this example network, each neuron in hidden layer 1 will take some proportion of its inputs using its learned values to produce an output. These outputs will then be used by the second hidden layer to perform some further level of abstraction. Finally, the output layer will take some proportion of the values output by the second hidden layer to produce an output.

To achieve this, the layer a^z is defined as the product of the previous layer a^{z-1} multiplied by some set of weights, W. Next a bias value b is added to each neuron to require the neuron to reach a certain level of 'activity' before firing. Finally, a logistic function like the sigmoidal function (σ) is applied to limit the output to within a defined range.

Mathematically it can be expressed as

$$a^{(z)} = \begin{bmatrix} a_0^{(z)} \\ a_1^{(z)} \\ \vdots \\ a_n^{(z)} \end{bmatrix} = \sigma \left(\begin{bmatrix} W_{0,0} & W_{0,1} & \cdots & W_{0,n} \\ W_{1,0} & W_{1,1} & \cdots & W_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ W_{k,0} & W_{k,1} & \cdots & W_{k,n} \end{bmatrix} \begin{bmatrix} a_0^{(z-1)} \\ a_1^{(z-1)} \\ \vdots \\ a_n^{(z-1)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} \right)$$

For the sake of brevity, this is often shortened to

$$a^{(z)} = Wa^{(z-1)} + b$$

or more simply

$$y = Wx + b$$

3.1.2 Training

In order to solve for *f*, the model is trained to learn it's weights and biases in an attempt to find *f'*. For this a cost function, C, is defined that determines how much error there is between the network's prediction for a given input and the desired output. By trying to minimise the result of this function the network is trained toward an optimal *f*. This output is a function of every single weight and bias in the network, therefore our cost function exists somewhere in high dimensional space. As these networks get more complex, it becomes increasingly unlikely that the true global minima for this cost function is found but rather a local minima. Typically this local minima is acceptable, however depending on the training parameters, or hyperparameters, it is possible that the optimisation process can get stuck in a sub-optimal location.

Mathematically the cost function is bound by the optimal function *f'* as its output is minimal for the optimal *f*, such that

$$C(f') \leq C(f) \forall f \in \mathbb{F}$$

To achieve this we utilise Backpropagation to determine the gradient, or derivative, of each variable in the network for a given input. This gradient is used to alter the variables by a value proportional to the learning rate to reduce the output of the cost function for the given examples. While it is extremely important to understand how this process works, most deep learning libraries including TensorFlow will handle this computation. There are several somewhat unintuitive ways that this process can cause problems like vanishing or exploding gradients, or cause neurons to die. For some examples in which these problems occur, see this post by Karpathy [19].

Training a neural network can take a very long time, particularly on a large data set like ImageNet. One of the major issues that can occur is overfitting of the training data. Looking at only the orange line in Figure 3.1.2 it appears that the network is continuing to improve the longer it trains. However the blue line shows the loss against a validation data set. It is important that training is stopped when these two lines begin to diverge otherwise the training data has been overfit, indicated by the shaded area.

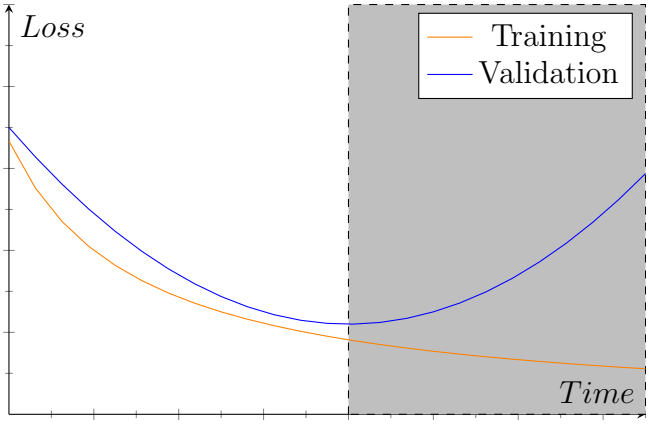


Figure 3.2: Example of a neural network overfitting

3.1.3 Convolutional Neural Networks

Convolutional Neural Networks are a more advanced type of neural network designed specifically for computer vision. For these the input is defined as a four dimensional matrix of size $[Batch\_Size \times Height \times Width \times Channels]$ where

- Batch_Size denotes the number of images to be processed
- Height denotes the height in pixels of each image
- Width denotes the width in pixels of each image
- Channels denotes the number of colour channels in the image

In this type of network sets of convolutional and pooling layers are used, followed by fully connected layers like the hidden layers in a traditional artificial neural network.

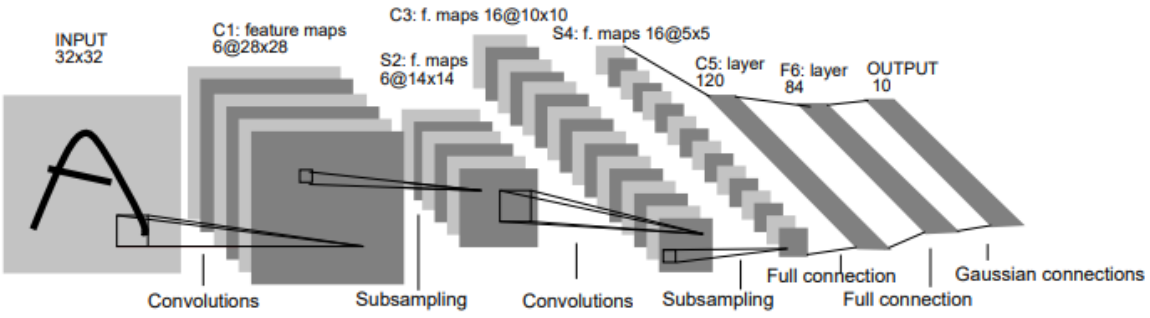


Figure 3.3: Convolutional Neural Network, LeCun et al. [24]

Convolutional layers

In a convolutional layer, a set of feature maps of a given dimension F, are learned which are convolved with the input image. In a single layer, this set F contains D number of filters each with a set of weights. Each filter has a size $n \times n$ which is convolved with the image by moving it around with some stride s. One problem with the diagram in Figure 3.4 is that the outer layer of pixels is lost.

Some amount of zero-padding, P, is used to solve this problem. It also serves to help control the output sizes. The output size can be calculated with the following formula when the input has height and width, W. For non-square convolutions the height and width are calculated separately.

$$Size = \frac{w - n + 2P}{s} + 1$$

For each filter in the set this kernel convolution is performed creating an output volume of $Size \times Size \times D$ for each image or $BatchSize \times Size \times Size \times D$ for a batch of images.

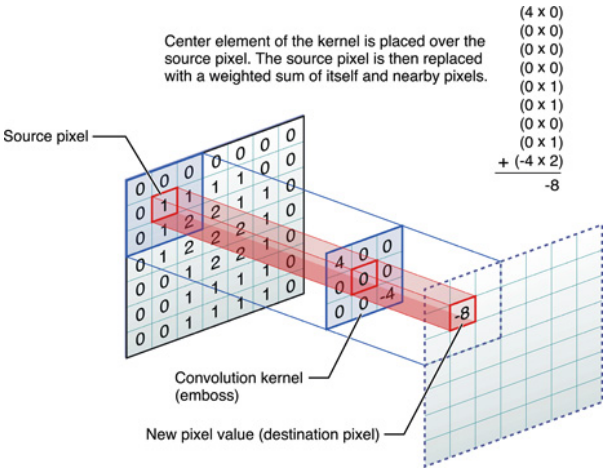


Figure 3.4: Kernel Convolution, [2]

It is important to note that the weights used in a feature map are shared. That is to say there is only one set of weights and one bias per kernel. This drastically reduces the number of parameters when compared with a fully connected approach by making one crucial assumption. If a feature is useful to compute at some spatial location (x, y) then it is also useful to compute at (x_2, y_2). This assumption makes logical sense as if it is helpful to detect a horizontal line here to determine if a cat is in the image, if the cat is moved it should still be useful.

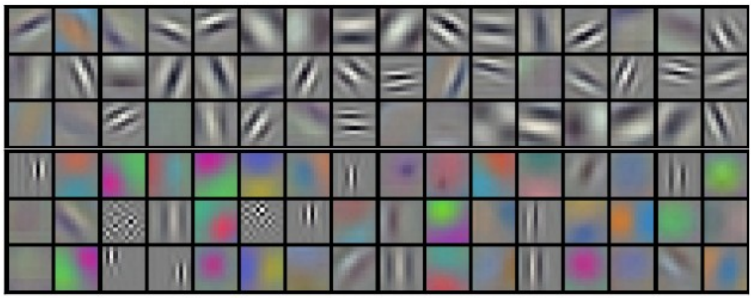


Figure 3.5: Example feature maps learned by Krizhevsky et al. [22]

Figure 3.5 is a visualisation of the networks feature maps. These feature maps are represented by brightness, where the brighter the colour the higher the associated weight and vice versa. The colours correspond to the weighting given to each of the RGB channels at that location, therefore greyscale represents a colour agnostic feature. It can be seen that some of these are just serving as line detectors as mentioned prior.

These convolutional layers are therefore essentially serving as localised feature detectors. The later convolutional layers then serve as an aggregation of sets these features that the network considers to be useful in detecting something like a cat or a dog.

Pooling layers

Typically pooling layers are inserted between successive convolutional layers to reduce the spatial dimensions and consequently the number of parameters and computational

cost of the network. A benefit of this reduction is helping to control overfitting in the network, when in the extreme case, learning too many parameters is akin to memorising the inputs and desired outputs rather than a generalised function.

There are several methods of pooling such as MAX, AVG, or LogSumExp (LSE) pooling. These differ purely in the mathematical function used to pool the values over a specified pool size. For example, a max pool operation with pool size of 2 would take the maximum value in a 2x2 pixel area. Similar to a convolutional layer this kernel is then moved across the image by the stride distance calculating the value at each set of pixels.

Fully Connected layers and output

The spatially downsampled four dimensional matrix is connected to a fully connected layer as in a typical artificial neural network. In these layers a technique known as dropout is also used[13]. During training a probability is given to the network to decide what proportion of the neurons to randomly set to zero. This helps prevent overfitting of the training data by reducing reliance on specific neurons for prediction. At inference or testing time, none of the neurons are dropped. Finally, this high dimensional vector can be passed to a final output layer which has a node for each possible output of the network.

A logistic function will then need to be applied to this final output to create a more useful output. For example, if a network had two outputs, one for cat and one for dog, one would want to use a *softmax* function to turn this output into a probability distribution of whether the network thinks this image is a cat or a dog.

However in my case, I will be using a *sigmoid* function as this gives me the independent likelihood of the presence of each disease in the image, which is important as the diseases are not mutually exclusive. That is to say there may be zero or more diseases in any given input and the presence of one does not necessarily prevent the presence of another.

Local Response Normalisation

Local Response Normalisation attempts to mimic the concept of lateral inhibition from neurobiology. In essence, an excited neuron in the brain can reduce its neighbours activity. It is responsible for preventing the spread of activity to nearby neurons and creates a contrast that allows for increased perception, initially described by Krizhevsky et al. [22], and is used to aid feature detection.

It is achieved by enhancing the peaks in localised neuron activation whilst also dampening large areas of highly activate neurons. This is useful because the increased activity of a single neuron is much more indicative of the presence of a particular stimuli that the neuron is tuned to. Therefore a single, particularly active neuron will stand out much more while an area of active neurons will suppress each other and reduce the overall activity.

Mathematically described as the following where α, β, k, and n are hyper-parameters.

$$b^i_{x,y} = \frac{a^i_{x,y}}{\left(k + \alpha \sum_{j=max(0,i-n/2)}^{min(N-1,i+n/2)} (a^j_{x,y})^2\right)^\beta}$$

It takes the weighted sum of squares of a kernel around an input a at location x, y, adds a small bias to avoid division by zero, to the power of β. The original input a is then divided by this figure to produce the output.

Batch Normalisation

Batch Normalisation, developed by Ioffe and Szegedy [17], is a technique that acts as a regularisation to improve both time to converge and model accuracy. It works by normalising the input to a given layer, x_n, by subtracting the batch mean from the output of the previous layer x_{n-1} and then dividing by the batch standard deviation. However this change will mean that the existing weights in the next layer are no longer optimal. To fix this problem, it introduces two trainable parameters. γ, the standard deviation, and β, the mean. These two parameters act as a denormalisation to keep the stability of the network.

When the distribution of the input to the network changes, the network experiences covariate shift [39]. For example, a car classifier trained only on red cars and non-car objects would be likely to struggle to identify cars if they were blue. Batch Normalisation is a solution to this problem.

Since it is a regulariser, it helps reduce overfitting and allows the reduction of the dropout rate used. Additionally, a higher learning rate can be used because the range of activations is a lot smaller.

3.1.4 Rectified Linear Unit

The Rectified Linear Unit, or ReLU, is a simple mathematical function that takes the maximum value between 0 and x.

$$y = max(0, x)$$

This has been empirically shown to be the most successful activation function when building neural networks, rather than the more traditional *sigmoid*. Additionally, the ReLU activation function incurs very little computational cost as it is a simple thresholding function. In practice it has also been shown to converge much faster than *sigmoid* and to improve accuracy [27]. These reasons are why it has become the standard in recent research.

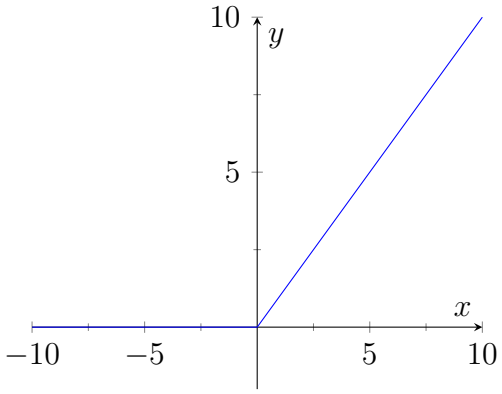


Figure 3.6: The ReLU function, $y = max(0, x)$

However, ReLU is not without flaws. When computing the gradient of the ReLU during backpropagation, the gradient will always be zero when the neuron is outputting a negative number (See Figure 3.1.4). This means there will be no update to the neurons' weights in the learning process and so it is possible that this neuron will become stuck. This is known as the dead ReLU problem and can happen either due to poor initialisation such that the neuron never fires, or due to a large update during training which might be caused by a high learning rate. This is not at all uncommon in a network as described by Andrej Karpathy[19], but fortunately there is a solution to this problem.

Leaky ReLU

The leaky ReLU looks to solve this problem by adding a very slight gradient to the negative x area of the graph. While this does prevent the dead ReLU problem and performs better than a standard ReLU, Xu et al. [48] have shown that there are even better alternatives.

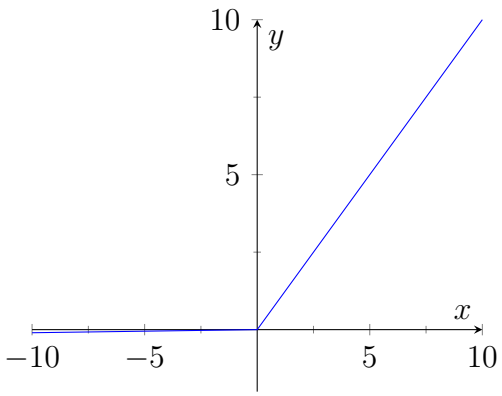


Figure 3.7: The Leaky ReLU function, $y = max(0.01x, x)$

Parametric Leaky ReLU

$$y = max(\alpha x, x)$$

The Parametric Leaky ReLU follows a similar concept to the Leaky ReLU but defines a new variable for each neuron, α. This α should be learned by the network during backpropagation just like another weight or bias, and shows yet further improvement over the Leaky ReLU.

Randomised Leaky ReLU

Finally, as described by Xu et al. [48], the Randomised Leaky ReLU was proposed in a Kaggle National Data Science Bowl competition. It specifies that instead of learning α, it is sampled from a uniform distribution for each batch during training, such that the ReLU will be somewhere within the grey area in the below figure. In a similar method to dropout, at testing time it is substituted for the average of the distribution, as shown by the blue line in the figure.

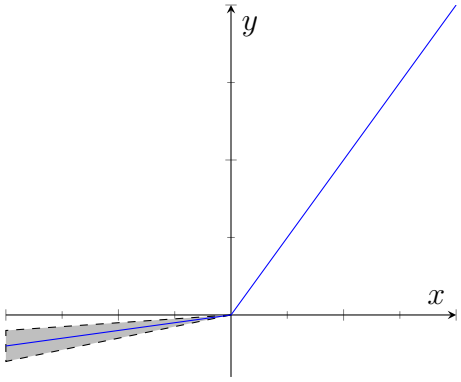


Figure 3.8: Randomised Leaky ReLU

3.1.5 Sigmoidal Function

The logistic sigmoidal function, is used for the final classification in a multi-label problem such as this project. It is defined as

$$y = \frac{1}{1 + e^{-x}}$$

It is a bounded, real function that is defined for all real values of x, which means that it can map the output of a network into the range $0 \leq y \leq 1$ regardless of its actual output.

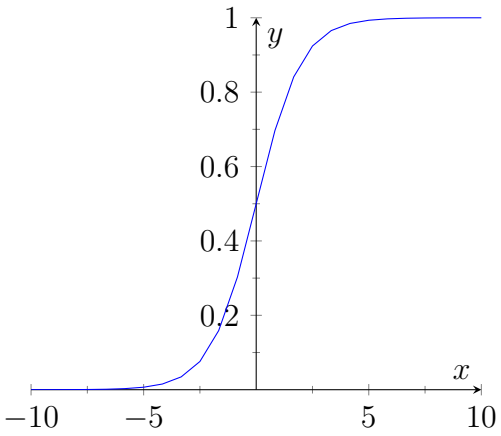


Figure 3.9: Sigmoid logistic function

It is particularly useful in this case as the models' output can be interpreted as a percentage likelihood of the presence of each label, and trained accordingly.

3.1.6 Cell-based Architectures

In 2014, Szegedy et al. [43] proposed a DCNN architecture code-named Inception. This network made use of a new 'Inception cell' as a method of feature extraction. By stacking several types of operation into a single concept and then merging them with a concatenation operation, it parallelises the network. Each of these operations can be computed separately, thus speeding up computation. It is also shown that this method improves performance of networks significantly.

In this paper they begin with a few traditional convolutional, pooling, and local response normalisation layers. These first few layers should act as a very low level feature extraction, for example edge and line detection. These groups, or cells, of parallelised layers are then designed to extract another set of features. Logically this implies that stacking several of these would lead to further and further levels of feature extraction. Additionally these cells can be designed to reduce computation through 1x1 convolution shielding.

While not the only novel innovation, building networks in this style has significantly improved the state of the art. Another paper by Liu et al. [26] took an interesting approach to this topic. They make use of a genetic algorithm with a performance predictor LSTM network. They begin with a simpler cell structure and mutate some children. Using the predictor LSTM they can avoid fully training networks that will perform significantly worse, thus reducing the search space and computation cost. They finally train this subset of models to choose a new parent architecture. PNASNet, while an interesting point of research, has not been included as a test network in the project as the paper was written in March 2018.

As LSTM (Long Short-Term Memory) networks are out of the scope of this project, more information can be found in the paper that describes them by Hochreiter and Schmidhuber [14].

3.1.7 Residual Connections

In 2015, He et al. [11] proposed a novel model in the ILSVRC called ResNet. This network utilises the residual connection as seen in the following diagram from their paper. In addition to typical convolution layers with ReLU activation, they make use of an identity connection. This identity connection means that the input to this layer is concatenated with the output of this layer in the fourth dimension.

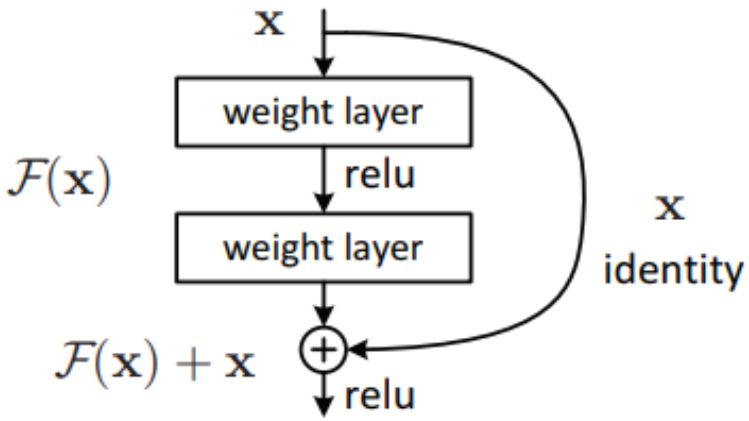


Figure 3.10: Residual block diagram from He et al. [11]

The idea of this passthrough connection is to allow stacking of previously inconceivable numbers of layers. The theory behind this is that each layer will now either learn to extract some useful feature from the input, or simply act as a passthrough and perform no computation. This set a new state of the art at the time due to the vast depth of the network. For example, a previous network like Inception used 22 layers, while the state of the art ResNet proposed by He et al. [11] had 152 layers.

3.2 VGG-Net

VGG, specifically VGG16, as proposed by Simonyan and Zisserman [41] is a network that secured first place in localisation and second place in classification in the ILSVRC in 2014. It's main contribution over the previous state of the art was the use of much smaller 3x3 convolutions compared to the 11x11 convolutions used by Krizhevsky et al. [22]. This change allowed them to create a much deeper network with 16 layers at a much lower computation cost. They explained

> So what have we gained by using, for instance, a stack of three 3x3 conv. layers instead of a single 7x7 layer? ...
>
> ... we decrease the number of parameters: assuming that both the input and the output of a three-layer 3 x 3 convolution stack has C channels, the stack is parameterised by 3 $(3^2C^2) = 27C^2$ weights; at the same time, a single 7 x 7 conv. layer would require $7^2C^2 = 49C^2$ parameters, i.e. 81% more.

This network along with another ILSVRC 2014 competitor, Inception, could be credited for the shift in convention of neural network architecture. Previously networks had just been made larger by increasing the number of neurons on a given layer, or making them 'wider'. In the case of Krizhevsky et al. [22], this caused a bottleneck on the hardware available at that time and requiring them to split the network across multiple GPUs.

At a monolithic 140 million parameters, VGG would be one of the largest networks to date. Interestingly, the output of the final pooling layer before the fully connected layers contains just $7 \times 7 \times 512 = 25,088$ outputs. However, the first fully connected layer contains an astonishing $7 \times 7 \times 512 \times 4096 = 102,760,448$ weights. This means that nearly 75% of the networks parameters are contained in a single layer. More recent models have shown that these kinds of monolithic layers can be removed in favour of methods like global average pooling to reduce the dimensionality of the network with far fewer parameters.

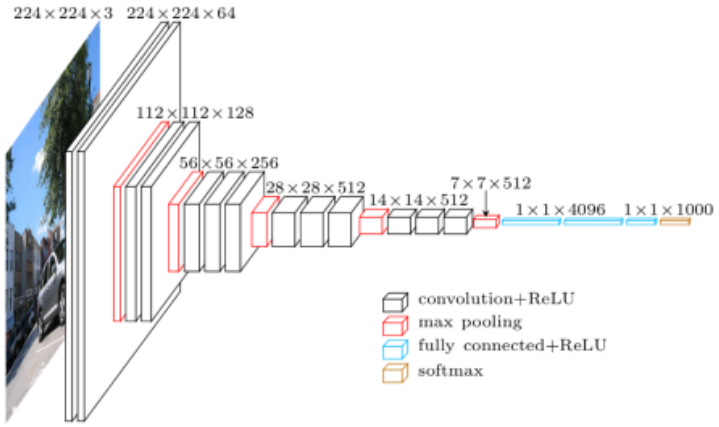
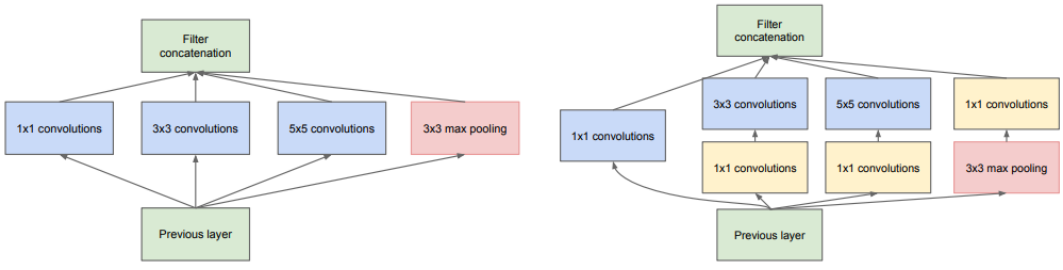


Figure 3.11: Macroarchitecture of VGG16 [3]

3.3 Inception

Inception, also known as GoogLeNet in honour of Yann LeCun's LeNet 5 [23], was the winner of the ILSVRC classification task in 2014 [43]. It proposed the idea of an inception module, a type of cell that was stacked nine times to set a new state of the art in image classification.



(a) Inception module, naïve version

(b) Inception module with dimension reductions

Figure 3.12: Inception modules from Szegedy et al. [43]

The initial inception module, (a) from Figure 3.12, showed the use of parallelised convolutions and pooling of differing sizes which are then concatenated together to produce another output with the same number of dimensions. However stacking even a small number of these modules together would quickly see an astronomical number of filters, due to pooling layers outputting the same number of outputs as their inputs. When the pooling layer is concatenated with yet more convolutions the number of outputs quickly becomes unmanageable.

The ingenious solution to this comes in the form of several 1x1 convolutions, also referred to as 'network in network', that act as a 'computational shield'. First described by Lin et al. [25], these counter-intuitive 1x1 convolutions allow us to reduce the size of the output dimension.

These 1x1 convolutions each consume the entire depth of the input, or the number of filters, to produce a single output with the same height and width as the input. By computing n number of these convolutions the input's depth can be mapped to a new size, n. This allows significant dimensionality reduction of the inputs and outputs to each inception module. As a result stacking several of these will keep within the bounds of the computational limits of the hardware used, hence the 'computational shield' as seen in (b) from Figure 3.12.

Periodically they also make use of 2x2 max pooling layers with stride 2 that halves the resolution (height and width) of the input.

In comparison to the other competitors from that year, VGG, this is also achieved with far fewer parameters. The entire GoogLeNet architecture consists of around 6.8 million parameters, more than 20 times fewer parameters.

The network itself consists of a small 'stem' of conventional convolution, pooling, and local response normalisation layers. They found this improved the training process in terms of memory requirements rather than stacking purely inception modules straight from the input. A total of nine inception modules follow with intermittent max pooling to further reduce dimensionality as mentioned above. Finally a global average pooling layer is implemented before a fully connected layer and final classification through softmax.

A full diagram of the network can be found on page 7 of the paper describing it by Szegedy et al. [43].

3.4 ResNet

ResNet, as described by He et al. [11], makes heavy use of the residual connections and batch normalisation referenced in sections 3.1.7 and 3.1.3, respectively. Each layer of a ResNet is a residual module that computes $F(x) + x$ given some set of weights that form the function F.

They stack several residual blocks together to form a larger block of a consistent set of dimensions. Each of these larger blocks results in a doubling of the number of filters. Essentially this means the network grows wider as it's depth increases. Finally a global average pooling layer is used before a fully connected layer for final classification through softmax.

In the paper they create a single model 152-layer ResNet that outperforms even ensemble models of the previous state of the art.

As an exploratory task they train a 1202 layer network on the CIFAR-10 data set[21]. They report that this network has no optimisation difficulties which speaks to the potential of residual connections and their scalability. However this network performs worse which is theorised to be due to overfitting as it may be overcomplicated for such a data set.

A follow up paper in 2016, introduced the ResNet v2[12] which is used for this project. Their paper proposes a modified version of the residual connection found in the original network as seen in Figure 3.13.

They also demonstrate the importance of their residual connections by performing a slight modification such that

$$x_{l+1} = F(x_l) + \lambda_l x_l$$

That is to say the next layer $l+1$ is calculated by the residual layer where the residual connection has been multiplied by a value λ. They prove that for $\lambda > 1$ this value can be exponentially large and for $\lambda < 1$ this value can be exponentially small and vanish. This forces the backpropagated signal through the weight layers and results in optimisation difficulties.

This implies that for optimal performance the residual connection needs to remain an identity function. In the previous paper this would be passed through a ReLU activation which would prevent this and potentially cause problems as demonstrated.

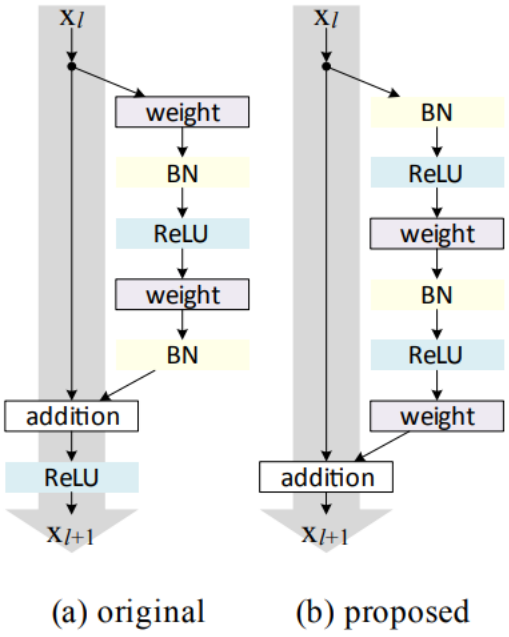


Figure 3.13: Residual blocks from He et al. [12]

The improved residual block results in a significant 1.1% reduction in error rate when comparing 200-layer ResNet models on the ImageNet[6] data set.

In this project, the 50-layer ResNet v2 model will be used due to memory limitations on the GPU and the superior accuracy over ResNet v1.

3.5 InceptionResNet

InceptionResNet, developed in 2017 by Szegedy et al. [42], is a hybrid network formed from the Inception modules by Szegedy et al. [43] and the residual connections by He et al. [11]. In the paper they describe several networks with differing costs and accuracies, the Inception-ResNet-v2 model is used for this project.

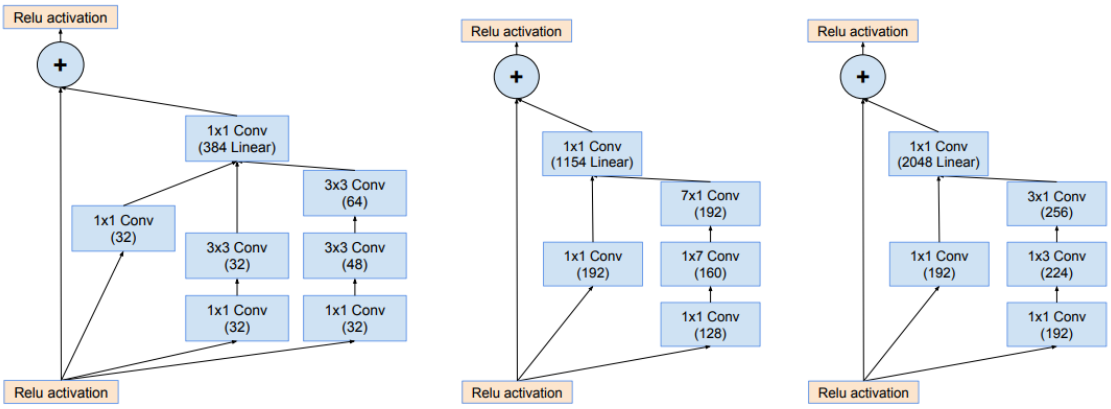


Figure 3.14: From left to right: Inception-ResNet-A, Inception-ResNet-B, and Inception-ResNet-C modules from Szegedy et al. [42] for the model used in this project

These modules are used in repetition in different sections of the network. Each block of modules is separated by a reduction block. The reduction blocks serve to reduce the size of the input along the height and width, achieved through a mix of convolutions and max pooling layers. Each of these has stride > 1 causing a reduction in the outputs size. Additionally they make use of 'valid' zero-padding rather than traditional zero-padding which reduces the output size slightly further. Another point of note is the significantly more complex 'stem' of the network when compared to previous inception models. This now consists of a more aggressive dimensionality reduction in the height and width dimensions which brings a 299×299 image down to 35×35 before any residual inception modules are used.

As can be seen in Figure 3.14, each type of module has been simplified from the original inception module. In addition, they implement a filter expansion layer in the form of 1x1 convolutions as described by Lin et al. [25]. This is required to offset the dimensionality reduction of the inception module so that the output depth matches the input.

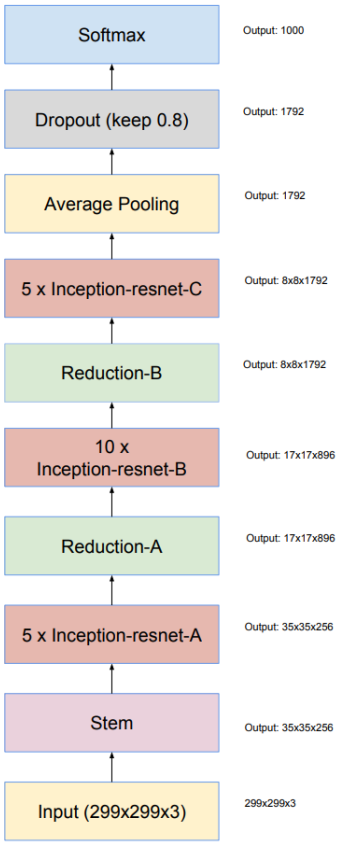


Figure 3.15: Inception-ResNet-v2 Architecture from Szegedy et al. [42]

They also find that their residual models train much quicker than their non-residual counterparts in accordance with He et al. [11].

In conclusion they present a more computationally expensive hybrid network that effectively combines inception modules and residual connections for significantly improved performance.

3.6 Data and prior research

The data set consists of 112,120 frontal CXR images across 30,805 patients. Each of the CXR images has some associated metadata. This metadata includes patient IDs, age, gender, the view type posterioranterior (PA) vs anteriorposterior (AP), and some statistics about the resizing of the image. For some images, a bounding box is provided but this is not used during the project.

PA views are the default in CXR imaging, where the X-ray tube or emitter is placed 6 feet behind them with the film in front of them. This is such that the x-ray enters the posterior and exits the anterior of the patient. In the AP view this is reversed, but typically performed with the X-ray tube above the patient laying on a bed. Additionally this is usually reserved for particularly ill patients that cannot stand as the AP view is much less useful.

3.6.1 Classes

There are a total of 15 classes within the data set: Cardiomegaly, Emphysema, Effusion, Hernia, Infiltration, Mass, Nodule, Atelectasis, Pneumothorax, Pleural Thickening, Pneumonia, Fibrosis, Edema, Consolidation, and No Finding. I encode this as a 14-dimensional vector with an output to represent each disease, and a vector of zeroes to represent 'No Finding'. For more information about these within the context of CXRs, please refer to the Mayo Foundation for Medical Education and Research [28] or NHS [29] websites.

3.6.2 Previous work

When I began the project, there were two papers of interest on this data set. The first by Wang et al. [46] describes the release of the data set and gives some benchmark values for a few DCNN architectures on this data set. In it they describe the process of labelling the images using a natural language processing algorithm to extract the diagnoses from the written radiologist reports. This is not perfect, but the labels are expected to be >90% accurate and so are suitable for weakly supervised learning[18].

The second paper by Rajpurkar et al. [35] describes a 121-layer DenseNet, another cell-based architecture that makes use of dense blocks, that classifies the images with impressive accuracy. They focus primarily on the pneumonia class using the F1 score. The F1 score is the harmonic mean between precision and recall, which is calculated from the networks predictions. It is calculated using the following formula where TP is the number of true positive outputs, FP is the number of false positive outputs, and FN is the number of false negative outputs.

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \quad Precision = \frac{TP}{TP + FP} \quad Recall = \frac{TP}{TP + FN}$$

They compare the networks F1 score with that of human radiologist and find that it outperforms the mean performance of the humans.

During the project, Guan et al. [10] submitted a paper to arXiv[4] describing a multi-network approach. This multi-network saw a performance improvement over the existing state of the art, CheXNet[35]. This approach uses three branches to improve accuracy. The first, or 'global', branch is used to create a heatmap of activations right before the final pooling layer to use as a mask for the image. The image is then cropped around this area and fed through a second DCNN in the 'local' branch. They take the final output of each network after pooling and concatenate the result in a 'fusion' branch before a

fully connected layer and sigmoid to classify the image. They show that the output of the fusion network maintains a better AUROC than the output of the global or local branches.

4.0 Implementation

4.1 Images

4.1.1 Collecting the data

To begin the project I first needed to obtain a copy of the data such that I could preprocess it as necessary to being training. This can be found either directly from the NIH website [30], or from the Kaggle datasets page[18]. Additionally Wang et al. [46] have also supplied their training/validation and testing splits from their paper so for the sake of comparable results, I decided to use theirs. I still had to split the training/validation list in two so I wrote a small function to hash the patient number and use that to split the remaining files 87.5%/12.5%. This left me with a 70/10/20 training/validation/testing split. The function can be seen in Appendix A.

There is also a CSV file that contains a table of file names and the corresponding ground truth labels.

4.1.2 Processing the data

The data is saved in 1024×1024 pixel greyscale PNG format. As some of the networks consume $224 \times 224 \times C$ input while others require $299 \times 299 \times C$ input, I would have to transform the images at some point. I chose to do this during the input pipeline rather than preprocessing it and saving them to disk as GPUs are already very optimised for this type of transformation. This also saves me significant disk space as I would not have to duplicate the data set and as I would be performing some image augmentation anyway, it seemed like a relatively small computational cost.

For the pretrained networks I had to duplicate the greyscale image across the RGB channels to create an image of the correct depth before resizing to the appropriate height and width.

In terms of image augmentation, I decided to utilise small, colour, scaling, cropping, and flipping distortions. I do this to make the network invariant to changes in the image that do not affect the label. In the case of an X-ray, some things that could affect the image but would not affect the diagnosis are:

- The image may be brighter or darker depending on the exact amount of x-ray radiation used
- If the patient is slightly closer or further from the X-ray emitter, the patient will appear to be smaller or larger respectively.
- The patient may not be perfectly centered in the image.
- The image may be taken PA or AP, this is arguably the most important distortion.

I chose to only utilise vertical axis (left-right) flipping as this allows us to train on a PA and AP version of each image. However horizontal axis (up-down) flipping is not

useful as the patient should never be upside down when the x-ray is taken. PA is the default projection for most cases, however sometimes the patient is unable to stand and the AP view will be used instead. As a result it is important to train on both types. Image flipping is not a perfect solution to this as the images will normally look slightly different as the relative distances between the organs and emitter, and organs and film have changed slightly.

4.2 Models

4.2.1 Hyperparameters

Each model is created inside its own python file that exposes a class. Constructing this class builds the computation graph associated with that model, and holds a reference to this TensorFlow graph. This also allows me to specify various hyperparameters specific to a given model, for example learning rate and batch size. These parameters are particularly important as an incorrect learning rate will result in poor optimisation or failure to converge. The batch size is a problematic hyperparameter as it has been shown that convolutional neural networks are sensitive to changes in batch size[9]. While their experiments looked at particularly excessive batch sizes, there is still a fluctuation between the smaller batch sizes. The main limitation here is GPU Memory, so I decided to attempt a default batch size of 64 for all networks. If this would not fit on the GPU I would find the largest multiple of 8 that would. The smallest batch size required was 24 for Inception-ResNet-v2.

4.2.2 Determining Epochs

As training time can be a major problem in convolutional neural networks, I decided to implement an early stopping algorithm. Usually a learning strategy revolves around learning for a defined period of time and decaying the learning rate periodically[42][43]. An alternative involves decaying the learning rate each time the validation accuracy plateaus[41][11]. As I would be dealing with a large range of networks and training each several times to test differing setups, it would not be feasible within the time limit to tune an optimal fixed learning strategy for each network. Therefore, I decided to implement a strategy whereby the learning rate decays by a factor of 10 each time the validation loss at the end of an epoch plateaus. In addition, if the validation loss does not improve in the epoch succeeding a learning rate decay then the training stops.

4.2.3 Modifying the networks for multi-label classification

As these networks are defined for tasks like ImageNet, they are not immediately suitable for application to this data set.

Classification

In the classification layer, the models implement a softmax classifier that forces the output to be a probability distribution. As there can be more than one label in my data set, this will not work. Therefore I replace the softmax with a sigmoid classifier which gives us the probability of the presence or absence of each disease.

Loss function

I also need to redefine the loss function, however, there are many choices here. I looked at previous work in the field for examples as the loss function is extremely important to optimisation of the network. In Rajpurkar et al. [35], they demonstrate the use of the normal unweighted binary cross entropy loss function. This is defined as the following where $f(x_c)$ denotes the output of the network after applying the elementwise sigmoidal function for that class, and y_c denotes the ground truth for that class.

$$\sum_{c=1}^{14} [-y_c ln\left(f(x_c)\right) - (1 - y_c) ln\left(1 - f(x_c)\right)]$$

In Wang et al. [46], they demonstrate the use of a weighted binary cross entropy function where β_P corresponds to the total number of examples in the batch divided by the number of positive examples in the batch, and β_N corresponds to the total number of examples in the batch divided by the number of negative examples in the batch. It is defined as the following

$$\beta_P \sum_{y_c=1} -ln(f(x_c)) + \beta_N \sum_{y_c=0} -ln(1 - f(x_c))$$

They implement this as there are significantly more negative examples than positive examples in the data set. In other words, the data set is very sparse. As a result this function punishes false negative examples more harshly in an attempt to enforce the learning of positive examples. As this seemed like an interesting point of comparison, I decided to test both loss functions for each network to determine which one is more optimal for this task.

As a side note, when implementing these algorithms it is important to add a small ϵ to $f(x_c)$, typically in the range 1×10^{-5} to 1×10^{-7} to avoid calculating $ln(0)$.

Network initialisation

When training a network there are several options for initialising the layers. If it is intended to train the network from scratch then the layers are typically initialised with a small value around zero. For example, a normal truncated distribution with mean 0 and standard deviation 0.02 discarding any values further than two standard deviations from the mean.

An alternative to this which has shown great success is transfer learning. First proposed by Pratt et al. [33], transfer learning is the practice of using weights from one network trained on a certain task to aid another network in learning a similar task. Due to the huge variation in subjects of the ImageNet data set, it makes an ideal candidate for this. However there is still a decision to be made. When using model weights from a model trained on ImageNet as an initialiser, it is possible to freeze a certain portion of the network. This is usually achieved by loading the pretrained weights up until the final fully connected layer and then constructing a new fully connected layer with the appropriate number of outputs. As an experiment with this I chose to perform two differing types of initialisation with ImageNet, to see which would perform better.

1. Freeze the initialised weight layers and only train the final fully connected layer

2. Train the entire network end-to-end using the initialised weights as a starting point

4.2.4 Training and evaluating the models

In order to train the models I chose to write a python script that would instantiate an imported network class, and a data handler class. By doing this, I can logically separate the different functions and easily swap individual models out without changing any other code. This helps to minimise the number of variables that are changing, as well as being good coding practice.

For each model, I set a fixed batch size based on the memory limitation I had and began with a low learning rate of 0.001 for the pretrained models and 0.1 for the fresh models. I then ran a set of 6 training runs on each model for each permutation of weighted cross entropy vs unweighted cross entropy loss and training all the layers vs training the final layer only vs de novo initialisation.

It quickly became apparent the likely reason why previous research had been using models pretrained on ImageNet[46][35]. Most of these models had a large number of optimisation problems and would not converge properly. While it is generally accepted that transfer learning from ImageNet is a good thing for most tasks, it seems to be a requirement for this data set. As a result, I had to discard these results as the networks were not performing any better than random guess and sometimes becoming one-way functions. This left me with 4 runs of results.

At the beginning of each epoch I shuffle the data before feeding the training data through the model and calculating the training updates. Every 20 training batches, I run a validation batch to track the validation accuracy on TensorBoard[44], TensorFlow's visualisation tool. At the end of the epoch, I benchmark the network performance over the entire validation data set, and adjust the learning rate if necessary. I save a copy of this model's weights for later use and continue until the early stopping conditions mentioned above are met.

Subsequently I can use those model checkpoints to benchmark on the testing data to look at how the model performs over time as well as the final fully trained model.

4.3 Evaluation methodology

As this is a multi label classification problem, to evaluate the accuracy of the network I measure the Area Under Receiver Operating Characteristic Curve, or AUROC, for each class. The ROC curve is determined by plotting the True Positive Rate (TPR) against the False Positive Rate (FPR) at various thresholds. The TPR is also referred to as the sensitivity or recall, and the FPR is also referred to as (1 - Specificity) as this is another method of calculation. Random guess at a task would result in a $y = x$ line with exactly 0.5 AUC. A perfect model would ideally predict 1.0 TPR with changes in the FPR as the threshold changes. Therefore our perfect score would be 1.0 AUC.

This will generate 14 values between 0 and 1 to compare the performance of each network on each class. One method of comparing performance as a whole across all classes is to take a mean of these values. This method is not ideal as it is not robust, and can be significantly impacted by very good or very poor performance on a single class. However as it has been computed in other papers I also provide this metric[10].

5.0 Results

I present AUROC over time graphs representing the networks AUROC on the testing data after each epoch. Additionally I present ROC graphs for each category from the best fully trained network of each type.

In each of the following graphs, CE refers to the unweighted cross entropy loss, WCE refers to the weighted cross entropy loss, All refers to all layers being trained, and Last refers to only the final fully connected layer being trained.

5.1 VGG

The VGG network contains the most parameters of any of the networks tested, yet is the worst in terms of ImageNet classification performance. This speaks largely to improvements in architecture since it was proposed. We can see from the AUROC over time graph that the weighted cross entropy with only the last layer trained performed best. Despite this, when training all the layers in this network with this method it failed to converge and resulted in 0.5 AUROC. Repeating the experiment resulted in the same outcome, leading me to believe this is an issue with the optimisation of the loss function. It is also worth noting the speed at which it converges in contrast to the cross entropy with all layers trained. I suspect this is due to the much larger number of parameters being tuned each iteration. For VGG I used a batch size of 64 and a starting learning rate of 0.001.

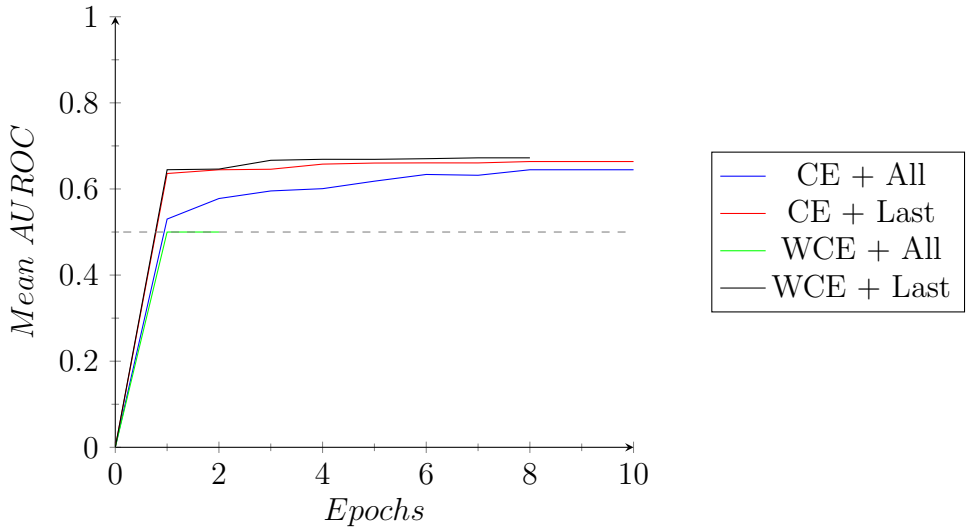


Figure 5.1: VGG models AUROC over time

The ROC curve for each pathology for the weighted cross entropy with only the last layer trained shows a significant gap between the network's best and worst pathologies. The network is very good at detecting Hernia at nearly 0.8 AUC, but has a very tough time detecting Nodule and Pneumonia at just 0.6 AUC.

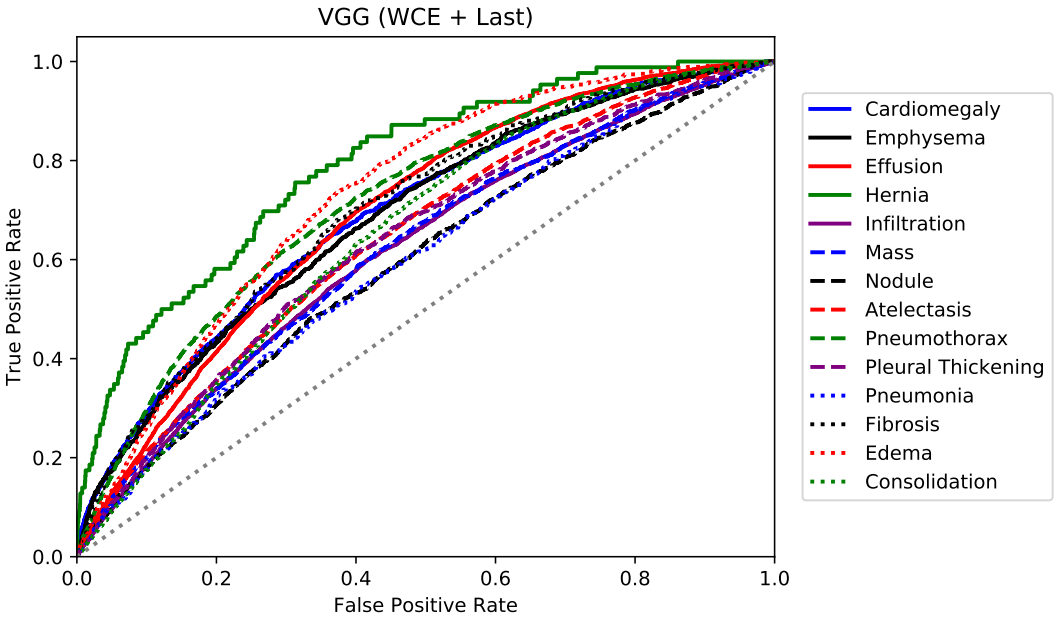


Figure 5.2: VGG best model: weighted cross entropy with only the final layer trained

5.2 Inception

Inception is the least complex of all of the models in terms of number of parameters, and 3rd best in terms of ImageNet classification performance. As can be seen from the AUROC over time graph, the network very quickly learns a positive correlation significantly above the random guess line. It can also be seen that for this type of network the difference between weighted and unweighted cross entropy is negligible. However there is a definite bonus to training all of the layers than just training the final layer. The winner here is unweighted cross entropy with all layers trained. For Inception I used a batch size of 64 and a starting learning rate of 0.001.

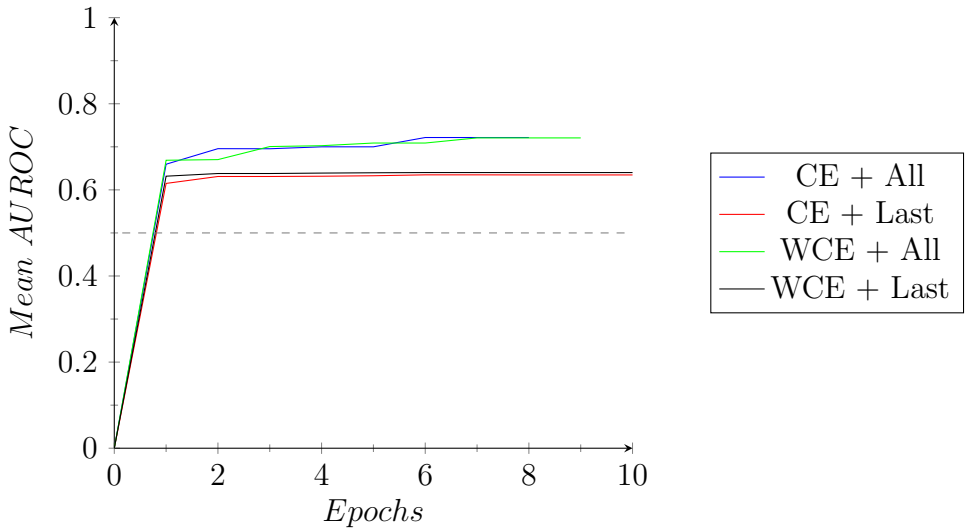


Figure 5.3: Inception models AUROC over time

While it converges relatively quickly after just 6 epochs, there is also a rather large disparity between the best and worst categories. Like VGG, this network is good at detecting Hernia. While it performed significantly better than VGG on Nodule and Pneumonia, it had similar troubles identifying Infiltration.

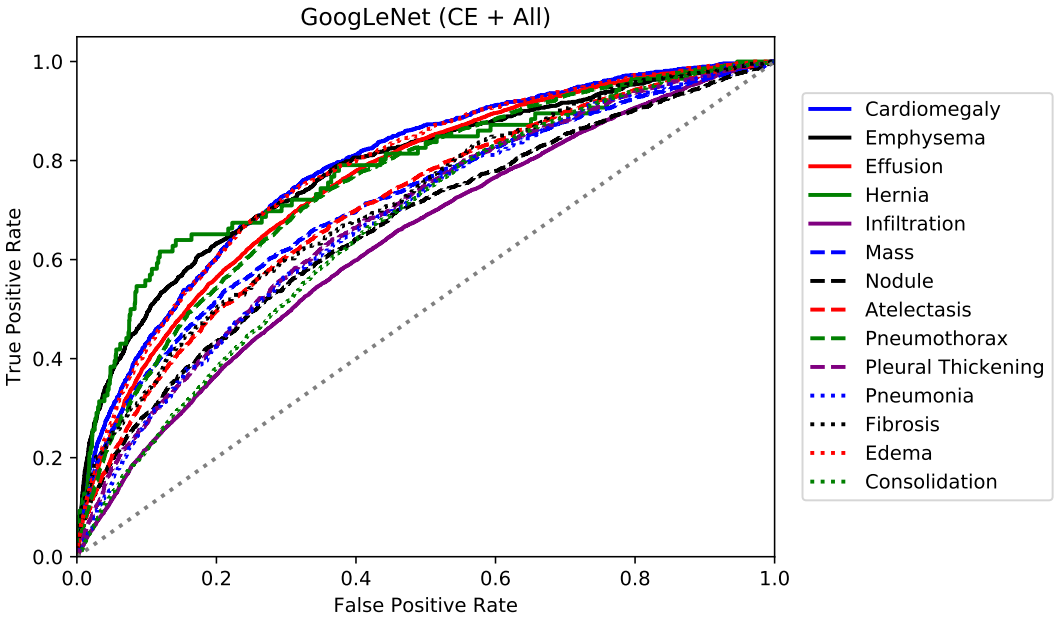


Figure 5.4: Inception best model: unweighted cross entropy with all layers trained

5.3 ResNet

ResNet is the second best performing network on the ImageNet data set, and a much deeper architecture than VGG or Inception. This 50-layer network seems to perform best with weighted cross entropy and all layers trained. However again the gap between weighted and unweighted cross entropy is very small, just 0.0086 mean AUROC. As with the Inception architecture, there is a large gain to be had by fine-tuning all the layers rather than just the final one. For ResNet I used a batch size of 32 and a starting learning rate of 0.001.

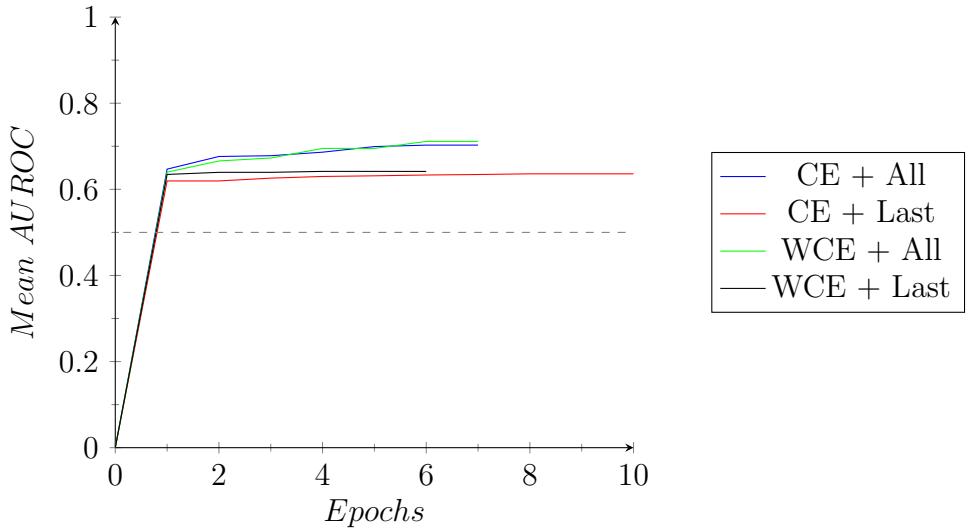


Figure 5.5: Inception models AUROC over time

I find that the best ResNet models converge more quickly than Inception and VGG models, but it does not outperform Inception. While the mean AUROC is lower than Inception, it achieves an impressive 0.81 AUROC on Cardiomegaly. Overall it still has a solid performance on the prediction task. It achieves significantly better than random guess performance across the board, and the faster convergence compared to Inception

is noteworthy given their similar average performance.

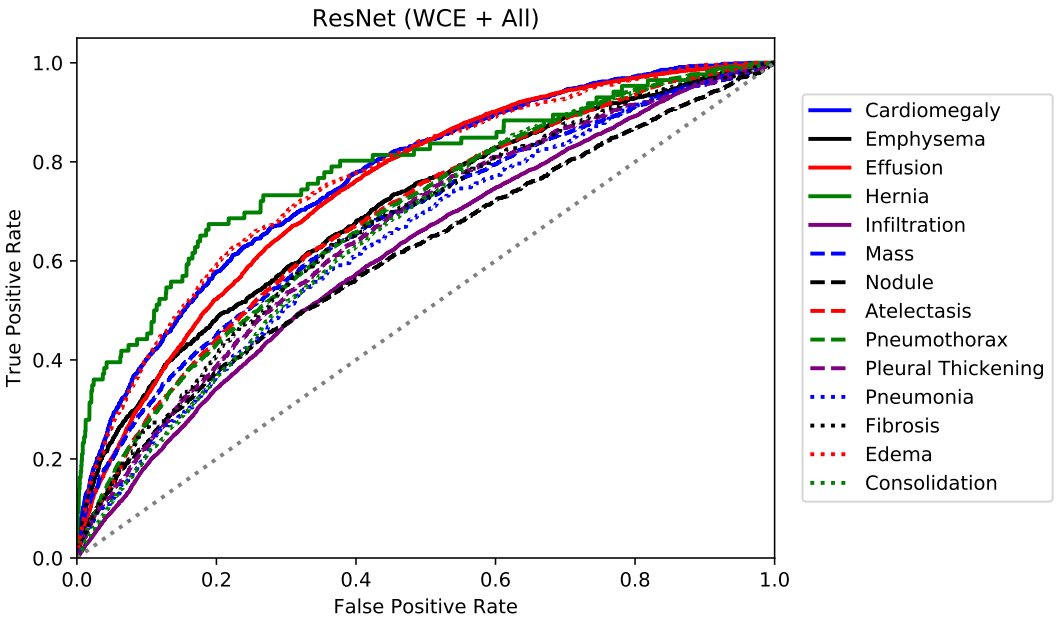


Figure 5.6: ResNet best model: weighted cross entropy with all layers trained

5.4 InceptionResNet

InceptionResNet is the most recent and strongest classifier for the ImageNet task. One might hypothesise that it would be a runaway winner but this does not appear to be the case. This model sees a very quick improvement over random guess and continues to improve over time as the learning rate drops. With this network the best model is unweighted cross entropy with all layers trained by a reasonable margin. It also has the largest difference of all the models between the best result when trained end-to-end compared to the best result from only training the final layer. For InceptionResNet I used a batch size of 24 and a starting learning rate of 0.001.

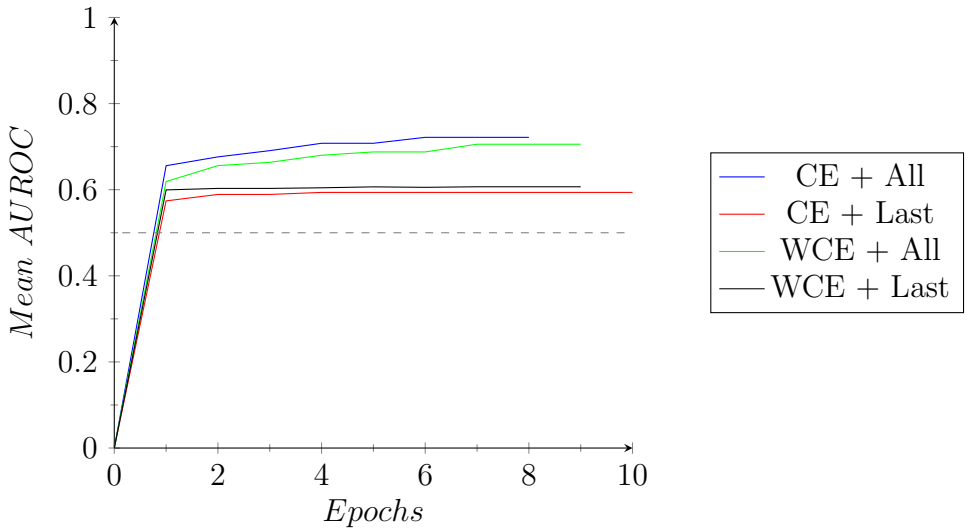


Figure 5.7: Inception models AUROC over time

I find that the best InceptionResNet model significantly outperforms the other models

tested. It has around 0.8 AUROC for several classes, which shows that the network has definitely developed a good understanding of what each disease looks like.

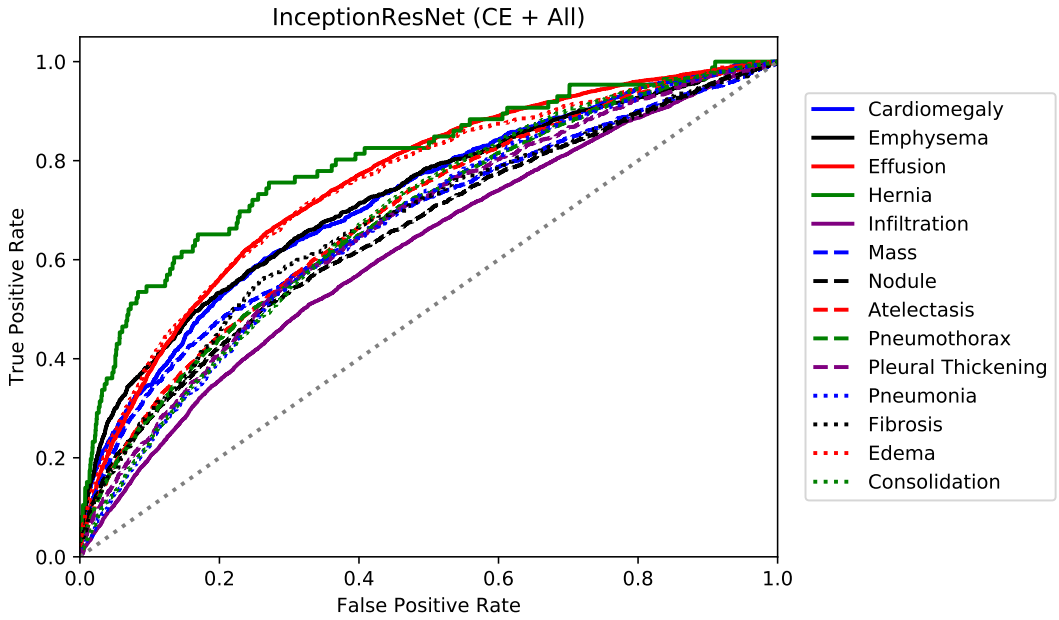


Figure 5.8: InceptionResNet best model: unweighted cross entropy with all layers trained

5.5 Summary

Pathology	**VGG** (WCE + Last)	**Inception** (CE + All)	**ResNet** (WCE + All)	**InceptionResNet** (CE + All)
Cardiomegaly	0.7016	0.7848	**0.8113**	0.7486
Emphysema	0.6864	**0.7826**	0.7221	0.7779
Effusion	0.7017	0.7598	0.7603	**0.7729**
Hernia	0.7905	0.7823	0.7801	**0.8024**
Infiltration	0.6175	0.6355	0.6345	**0.6384**
Mass	0.6208	**0.7119**	0.6967	0.7104
Nodule	0.6019	0.6667	0.6351	**0.6757**
Atelectasis	0.6462	0.7090	0.7017	**0.7142**
Pneumothorax	0.7190	**0.7504**	0.7123	0.7122
Pleural Thickening	0.6406	0.6836	0.6783	**0.6916**
Pneumonia	0.5864	0.6790	0.6646	**0.6868**
Fibrosis	**0.7194**	0.7123	0.6994	0.6920
Edema	0.7292	0.7788	0.7857	**0.7905**
Consolidation	0.6515	0.6660	0.6768	**0.6887**
Mean	0.6723	**0.7216**	0.7113	**0.7216**

Figure 5.9: Table summarising the AUROC values for each network's best result

Surprisingly the mean AUROC for Inception and InceptionResNet models are equal, however this looks to be due to InceptionResNet's particularly poor performance on Cardiomegaly and Pneumothorax. InceptionResNet provided the best performance across 9 of the 14 classes. There appears to be very little performance difference with respect to loss functions. However there is a very clear difference between the optimisation of

all layers versus only optimising the final fully connected layer. It also appears to be the case that the more recent networks that perform better on ImageNet have a larger differential between these two methods.

I suspect the difference between methods of training is related to the depth of the model. For example a shallower model can only perform a certain amount of abstraction, whereas a deeper model may have a much more defined concept of the classes it has been trained to classify. Therefore freezing these weights might make it difficult for the fully connected layer to draw meaningful abstraction. It is also possible that some middle ground of freezing only the first few layers might provide even better performance than end-to-end training.

Additionally, the second best InceptionResNet model performed closer to Inception for the Cardiomegaly class at 0.7703 AUROC. This suggests multiple runs or an ensemble model would likely lead to a higher mean AUROC.

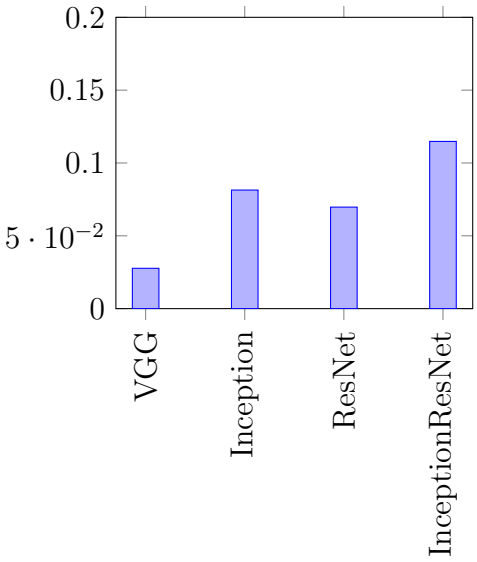


Figure 5.10: Delta between best mean AUROC of training all layers and training only the last layer

There also appears to be a correlation between performance on ImageNet and performance on this task for VGG, ResNet, and InceptionResNet. Inception appears to be an outlier here, it is possible that this run found a particularly good local minima or that it's architecture is just very suited for this task.

In summary, every network was capable of classifying all of the classes to some degree.

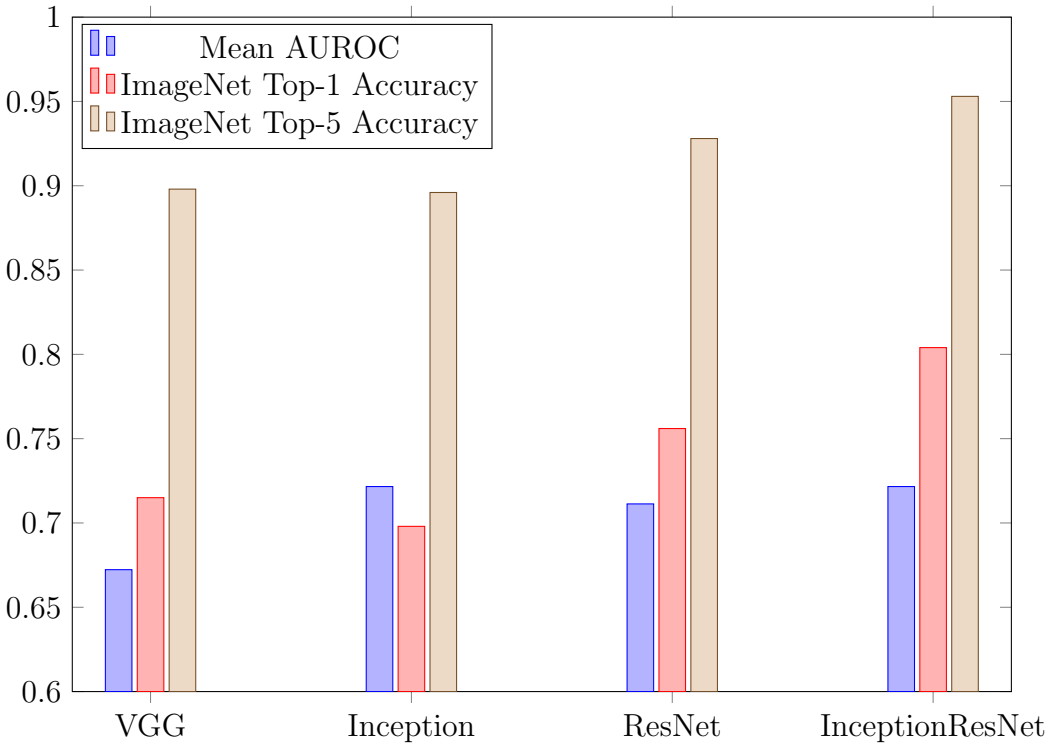


Figure 5.11: Comparison of the performance on this task to the performance on the ImageNet test set of the pretrained weights used[45]

6.0 Conclusion

6.1 Potential Improvements

There are definitely some areas of the project that could have been improved, especially given more time or computational power. I was only able to train each network once within the time frame of the project. A change as simple as altering the order of the data shown to the network during training will cause it to traverse a different area of the problem space. Therefore we may find differing local minima that will result in a different optimisation and subsequent accuracy. This could have been achieved either through repeated experiments, or through the use of a model ensemble. Usually a model ensemble will improve model accuracy at the cost of computation, but a recent paper suggests this might be possible without increased computation cost[15].

There are also some potential issues with the data set that have been addressed in previous papers [46][35]. The method of labelling is a small issue but we are still not quite modelling the human approach to diagnosis. Due to issues of patient privacy and confidentiality, no patient history is provided with the CXR data which has been shown to impact human performance[32]. It seems logical that this would have an impact on machine performance too.

At the end of the project I became aware of a paper that attempts to mimic the human diagnostic process with a second network that analyses a section extracted from the first networks activations[10]. They report some extremely impressive results and if I were to repeat this project I would look to perform a variation of this using only one network. This might be achieved by using multiple inputs to a shared CNN in a method similar to the one described by Wei et al. [47] but using the networks activations for the whole image to determine the inputs.

6.2 Deliverables

To recap, the objectives I set out to achieve were:

1. To research and understand the theory behind DCNNs, in particular, the test subjects
2. To implement these networks from scratch in TensorFlow
3. To compare the performance with a model pretrained on ImageNet
4. To experiment with hyperparameter tuning
5. To evaluate each models performance against a fixed training set of data

This should allow:

1. Determination of the viability of a CAD tool that relies on DCNNs

2. Comparison of the strength of different architectures on CXR14

3. Determination of any correlation between ILSVRC and CXR14 performance

4. Highlight any issues or limitations of the approach and how they might be overcome

I have succeeded in completing all of the desired objectives, and I am very comfortable with the models in question. I have drawn comparison between each of the models tested, with varying hyperparameters. Most importantly I have managed to conclude that all the networks were capable of learning to diagnose the CXR data, some more accurately than others. In the process I have found that CAD tools are definitely a possibility in the future and that there is a correlation between performance in different image tasks, specifically ILSVRC and CXR14. In Section 6.1 I have outlined potential improvements to the process, including a method that might improve the models significantly if the experiment were to be repeated.

6.3 Personal Reflection

Overall, I think that the project was very helpful to me. I learnt a huge amount about convolutional neural networks in both theory and their application. I feel as though I could very easily apply my knowledge to an entirely different problem and complete it in a much shorter time span.

I think that my approach to completing the project was good as I had very clear goals and milestones. This is partly due to the modular design of the project with each network architecture being it's own chunk that can be managed separately from the rest. If I were to repeat the project, I think I would approach it in a similar manner, although I would note that I could probably have managed my time slightly better as a very large portion of the early section of the project was spent on research. I am happy that it has given me a very thorough understanding of the theory behind DCNNs but did leave me less time to complete training and benchmarking of the models.

If I had more time to complete the project I would like to have been able to investigate a few more hyperparameters and ensembles as I think this could significantly improve my models' accuracy.

6.4 Conclusion

In conclusion, I believe that my project was very successful. Despite having no prior experience training neural networks nor with TensorFlow, I have managed to produce several models that all performed significantly better than random guess. In addition to this I had very little knowledge of Convolutional Neural Networks when I began, and now feel very comfortable with the concepts and intricate details involved. I believe there are still some issues with this data set, however, as it does not allow us to entirely mimic the human diagnostic process. It lacks any form of patient history which has been shown to improve human performance[32]. While these diagnostic scores are not yet up to a point that they could be feasibly used in a hospital, it shows promise for the future of CAD tools. I am particularly happy with these results due to the time and resource constraints of the project. There is definitely further work to be done, especially with the knowledge I now have, but I have achieved what I set out to accomplish, and learnt a huge amount along the way that will definitely help me a lot in the future.

References

[1] Abadi, M. et al. [2015], 'TensorFlow: Large-scale machine learning on heterogeneous systems'. Software available from tensorflow.org.
URL: *https://www.tensorflow.org/*

[2] Apple Inc. [2016], 'Kernel convolution diagram'. Accessed: 01/05/2018.
URL: *https://developer.apple.com/library/content/documentation/Performance/Conceptual/vImage/ConvolutionOperations/ConvolutionOperations.html*

[3] Blier, L. [2016], 'A brief report of the heuritech deep learning meetup #5'. Accessed:01/05/2018.
URL: *https://blog.heuritech.com/2016/02/29/a-brief-report-of-the-heuritech-deep-learning-meetup-5/*

[4] Cornell University Library [2018], 'arxiv.org'. Accessed: 01/05/2018.
URL: *https://arxiv.org*

[5] Delrue, L. et al. [2011], Difficulties in the interpretation of chest radiography, *in* 'Comparative Interpretation of CT and Standard Radiography of the Chest', Springer, pp. 27–49.

[6] Deng, J. et al. [2009], ImageNet: A Large-Scale Hierarchical Image Database, *in* 'CVPR09'.

[7] Dertat, A. [2017], 'Neural network figure'. Accessed: 01/05/2018.
URL: *https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6*

[8] GitHub Inc. [2018], 'What is my disk quota?'. Accessed: 01/05/2018.
URL: *https://help.github.com/articles/what-is-my-disk-quota/*

[9] Goyal, P. et al. [2017], 'Accurate, large minibatch sgd: training imagenet in 1 hour', *arXiv preprint arXiv:1706.02677* .

[10] Guan, Q. et al. [2018], 'Diagnose like a radiologist: Attention guided convolutional neural network for thorax disease classification', *arXiv preprint arXiv:1801.09927* .

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

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

[13] Hinton, G. E. et al. [2012], 'Improving neural networks by preventing co-adaptation of feature detectors', *arXiv preprint arXiv:1207.0580* .

[14] Hochreiter, S. and Schmidhuber, J. [1997], 'Long short-term memory', *Neural computation* **9**(8), 1735–1780.

[15] Huang, G. et al. [2017], 'Snapshot ensembles: Train 1, get m for free', *arXiv preprint arXiv:1704.00109* .

[16] Inc., G. [2018], 'Github deep learning repositories by most stars'. Accessed: 01/05/2018.
URL: *https://github.com/topics/deep-learning?o=desc&s=stars*

[17] Ioffe, S. and Szegedy, C. [2015], 'Batch normalization: Accelerating deep network training by reducing internal covariate shift', *arXiv preprint arXiv:1502.03167* .

[18] Kaggle Inc. [2017], 'Nih chest x-rays kaggle page'. Accessed: 01/05/2018.
URL: *https://www.kaggle.com/nih-chest-xrays/data*

[19] Karpathy, A. [2016], 'Yes, you should understand backprop'. Accessed:01/05/2018.
URL: *https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b*

[20] Karpathy, A. [2017], 'A peek at trends in machine learning'. Accessed: 01/05/2018.
URL: *https://medium.com/@karpathy/a-peek-at-trends-in-machine-learning-ab8a1085a106*

[21] Krizhevsky, A. and Hinton, G. [2009], 'Learning multiple layers of features from tiny images'.

[22] Krizhevsky, A. et al. [2012], Imagenet classification with deep convolutional neural networks, *in* 'Advances in neural information processing systems', pp. 1097–1105.

[23] LeCun, Y. et al. [1989], 'Backpropagation applied to handwritten zip code recognition', *Neural computation* **1**(4), 541–551.

[24] LeCun et al. [1998], 'Gradient-based learning applied to document recognition', *Proceedings of the IEEE* **86**(11), 2278–2324.

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

[26] Liu, C. et al. [2017], 'Progressive neural architecture search', *arXiv preprint arXiv:1712.00559* .

[27] Maas, A. L., Hannun, A. Y. and Ng, A. Y. [2013], Rectifier nonlinearities improve neural network acoustic models, *in* 'Proc. icml', Vol. 30, p. 3.

[28] Mayo Foundation for Medical Education and Research [1998-2018], 'Diseases and conditions, comprehensive guides on hundreds of conditions'. Accessed: 01/05/2018.
URL: *https://www.mayoclinic.org/diseases-conditions*

[29] NHS [2018], 'Health a-z - conditions and treatments'. Accessed: 01/05/2018.
URL: *https://www.nhs.uk/Conditions/Pages/hub.aspx*

[30] NIH Clinical Center [2017], 'Nih chest x-rays repository'. Accessed: 01/05/2018.
URL: *https://nihcc.app.box.com/v/ChestXray-NIHCC/folder/36938765345*

[31] Nvidia Corporation [2007-2018], 'Cuda parallel computing'. Accessed: 01/05/2018.
URL: *https://developer.nvidia.com/cuda-zone*

[32] Potchen, E. et al. [1979], Effect of clinical history data on chest film interpretation-direction or distraction, *in* 'Investigative Radiology', Vol. 14, LIPPINCOTT-RAVEN PUBL 227 EAST WASHINGTON SQ, PHILADELPHIA, PA 19106, pp. 404–404.

[33] Pratt, L. Y. et al. [1991], Direct transfer of learned information among neural networks., *in* 'AAAI', Vol. 91, pp. 584–589.

[34] Python Software Foundation [2001-2018], 'Python programming language'. Accessed 01/05/2018.
URL: *https://www.python.org/*

[35] Rajpurkar, P. et al. [2017], 'Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning', *arXiv preprint arXiv:1711.05225* .

[36] Rimmer, A. [2017], 'Radiologist shortage leaves patient care at risk, warns royal college', *BMJ* **359**. 10.1136/bmj.j4683.
URL: *https://www.bmj.com/content/359/bmj.j4683*

[37] Royal College of Radiologists [2016], 'Clinical radiology uk workforce census 2016 report'. Accessed: 01/05/2018.
URL: *https://www.rcr.ac.uk/system/files/publication/field_publication_files/cr_workforce_census_2016_report_0.pdf*

[38] Russakovsky, O. et al. [2015], 'ImageNet Large Scale Visual Recognition Challenge', *International Journal of Computer Vision (IJCV)* **115**(3), 211–252.

[39] Shimodaira, H. [2000], 'Improving predictive inference under covariate shift by weighting the log-likelihood function', *Journal of statistical planning and inference* **90**(2), 227–244.

[40] Silberman, N. and Guardarrama, S. [2016-2018], 'TensorFlow-Slim model library'. Accessed 01/05/2018.
URL: *https://github.com/tensorflow/models/tree/master/research/slim*

[41] Simonyan, K. and Zisserman, A. [2014], 'Very deep convolutional networks for large-scale image recognition', *arXiv preprint arXiv:1409.1556* .

[42] Szegedy, C., Ioffe, S., Vanhoucke, V. and Alemi, A. A. [2017], Inception-v4, inception-resnet and the impact of residual connections on learning., *in* 'AAAI', Vol. 4, p. 12.

[43] Szegedy, C. et al. [2015], Going deeper with convolutions, Cvpr.

[44] TensorFlow [2018*a*], 'Tensorboard: Visualizing learning'. Accessed: 01/05/2018.
URL: *https://www.tensorflow.org/programmers_guide/summaries_and_tensorboard*

[45] TensorFlow [2018*b*], 'Tensorflow slim pretrained model accuracy'. Accessed: 01/05/2018.
URL: *https://github.com/tensorflow/models/tree/master/research/slim#pre-trained-models*

[46] Wang, X. et al. [2017], Chestx-ray8: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases, *in* '2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)', IEEE, pp. 3462–3471.

[47] Wei, Y. et al. [2014], 'Cnn: single-label to multi-label', *arXiv preprint arXiv:1406.5726* .

[48] Xu, B., Wang, N., Chen, T. and Li, M. [2015], 'Empirical evaluation of rectified activations in convolutional network', *arXiv preprint arXiv:1505.00853* .

A.0 Training/Validation split function

```
def get_percentage_hash(self, file_name):
    # Hash only the patient number so that multiple images from the same patient
    # compute the same hash so they will be placed in the same subset.
    big_number = 2 ** 27 - 1 #~134M
    file_name = re.sub("_[0-9]{3}\.png", "", file_name)
    file_name_hashed = hashlib.sha1(compat.as_bytes(file_name)).hexdigest()
    # Bring hash in range [1-big_number], multiply by factor to set range [0-100]
    percentage_hash = ((int(file_name_hashed, 16) % (big_number + 1)) * (100.0 / big_number))
    return percentage_hash
```

B.0 Social, Ethical, and Legal Issues Addressed

Despite the use of medical imagery, the data set used is a publicly available, fully anonymised, data set. Therefore there were no social, ethical or legal issues to address.

C.0 Record of Meetings with Supervisor

Aside from the small gap in the project due to unforeseen personal circumstances, here is a record of the regular meetings with my Supervisor. The e-mails to arrange these meetings can be found at `https://github.com/Kaapp/chestxray-cnn-thesis-example/tree/master/supervisor-meetings`

Date	Subject	Outcome
11/10/17	Initial project discussion	Discussed a few potential ideas, created a small list to think about
16/10/17	Decision on initial project	Decided to begin working on a content based image retrieval system with Ran supervising. Will need to write initial proposal
6/12/17	Viva with Ran and Karina	Discussed the CBIR project, most of my proposal is good but Karina thought I should refocus on something more specific, with a defined goal
13/12/17	Discussion on various data sets as a new project focus	Discussed some data sets that I found, decided to look into the chest X-ray set as it looks the most interesting. Need to begin building the VGG model to test viability of data set
9/3/18	Getting back to the project	Met with Ran to explain the reasons for my absence, VGG results are promising. Rescheduled the plan and will begin on Inception testing.
15/3/18	Discussing methods to reframe the problem to improve accuracy	Rather than having a class for each output, I will see if accuracy can be improved by using two networks, one to detect a disease and another to classify which disease, or by encoding 'no finding' as all zeroes
17/4/18	Discussion of current progress and structure of dissertation	Current progress is looking promising so far, some difficulties with model convergence in de novo initialisation so I will focus on pretrained models. Also discussed approximate structure of dissertation
1/5/18	Some last minute pitfalls and discussion of extension	There was a small bug in my code involving an activation function that should not have been there, additionally discussed a small extension to the project to help me complete all modules in time due to my earlier absence
18/5/18	Final meeting regarding dissertation	General structure of dissertation is good and results look promising. Small amount of tidying up left to be ready to submit!

D.0 Example Inception Model and code

Code used to build an Inception model from scratch in TensorFlow using the tf.layers API for simplicity.

```
from __future__ import absolute_import
from __future__ import division

import tensorflow as tf
import numpy as np

# See: https://arxiv.org/pdf/1409.4842.pdf
class GoogLeNet:

    def __init__(self):
        self.num_labels = 14
        self.NAME = "GoogLeNet"

        #training params
        self.batch_size = 64
        self.learning_rate = 0.1
        self.weight_decay = 0.001

        self.graph = tf.get_default_graph()

        self.lr = tf.placeholder(tf.float32, shape=[], name='learning_rate')
        self.keep_prob = tf.placeholder(tf.float32, shape=[], name='keep_prob')
        self.is_training = tf.placeholder(tf.bool, shape=[], name='is_training')

        return None

    def construct_graph(self, x, y):
        with self.graph.as_default():
            model = self.conv(x, filters=64, kernel_size=7, stride=2, name='conv1_k7_s2')
            model = self.max_pool(model, pool_size=3, stride=2, name="maxpool1_p3_s2")
            model = tf.nn.local_response_normalization(input=model, alpha=0.0001, beta=0.75)
            model = self.conv(model, filters=64, kernel_size=1, stride=1, name='conv2_k1_s1')
            model = self.conv(model, filters=192, kernel_size=3, stride=1, name='conv2_k3_s1')
            model = tf.nn.local_response_normalization(input=model, alpha=0.0001, beta=0.75)
            model = self.max_pool(model, pool_size=3, stride=2, name='maxpool2_p3_s2')

            model = self._inception_module(model, filters=[64, 96, 128, 16, 32, 32],
                                           name='inception3a')

            model = self._inception_module(model, filters=[128, 128, 192, 32, 96, 64],
                                           name='inception3b')
            model = self.max_pool(model, pool_size=3, stride=2, name='maxpool3_p3_s2')
            model = self._inception_module(model, filters=[192, 96, 208, 16, 48, 64],
                                           name='inception4a')
            model = self._inception_module(model, filters=[160, 112, 224, 24, 64, 64],
                                           name='inception4b')
            model = self._inception_module(model, filters=[128, 128, 256, 24, 64, 64],
                                           name='inception4c')
            model = self._inception_module(model, filters=[112, 144, 288, 32, 64, 64],
                                           name='inception4d')
            model = self._inception_module(model, filters=[256, 160, 320, 32, 128, 128],
                                           name='inception4e')
            model = self.max_pool(model, pool_size=3, stride=2, name='maxpool4_p3_s2')
```

```
            model = self._inception_module(model, filters=[256, 160, 320, 32, 128, 128],
                                           name='inception5a')
            model = self._inception_module(model, filters=[384, 192, 384, 48, 128, 128],
                                           name='inception5b')
            model = self.avg_pool(model, pool_size=7, stride=1, name='avgpool5_p7_s1')

            logits = self.fully_connected(model)
            self.ys_pred = tf.nn.sigmoid(logits, name='prediction')

            with tf.name_scope('loss'):
                total_labels = tf.to_float(tf.multiply(self.batch_size, self.num_labels))
                num_positive_labels = tf.count_nonzero(y, dtype=tf.float32)
                num_negative_labels = tf.subtract(total_labels, num_positive_labels)
                Bp = tf.divide(total_labels, num_positive_labels)
                Bn = tf.divide(total_labels, num_negative_labels)
                # The loss function
                cross_entropy = -tf.reduce_sum((tf.multiply(Bp,  y * tf.log(self.ys_pred + 1e-9))) +
                                               (tf.multiply(Bn, (1-y) * tf.log(1-self.ys_pred + 1e
    -9))),
                                               name="cross_entropy")

                self.loss = cross_entropy # + l2 * self.weight_decay

            # Training the network with Adam using standard parameters.
            self.train_step = tf.train.AdamOptimizer(
                learning_rate=self.lr,
                beta1=0.9,
                beta2=0.999).minimize(self.loss)

    # Define some wrapper functions for brevity/readability
    def conv(self, inputs, filters, kernel_size, stride, name, padding='SAME',
             activation=tf.nn.relu):
        return tf.layers.conv2d(
            inputs=inputs,
            filters=filters,
            kernel_size=[kernel_size, kernel_size],
            strides=stride,
            padding=padding,
            activation=activation,
            kernel_initializer=tf.truncated_normal_initializer(stddev=0.001),
            name=name)

    def max_pool(self, inputs, pool_size, stride, name):
        return tf.layers.max_pooling2d(
            inputs=inputs,
            pool_size=[pool_size, pool_size],
            strides=stride,
            padding='SAME',
            name=name)

    def avg_pool(self, inputs, pool_size, stride, name):
        return tf.layers.average_pooling2d(
            inputs=inputs,
            pool_size=[pool_size, pool_size],
            strides=stride,
            padding='VALID',
            name=name)

    def fully_connected(self, inputs):
        dropout = tf.layers.dropout(inputs, rate=1 - self.keep_prob, training=self.is_training)
        # Need to reshape dropout to 2D tensor for FC layer, multiply the dimensions excluding
        # batch size
        new_shape = int(np.prod(self._get_tensor_shape(dropout)[1:]))
        dropout = tf.reshape(dropout, [-1, new_shape])
        return tf.layers.dense(dropout, self.num_labels)

    def _get_tensor_shape(self, tensor):
```

```
        return tensor.get_shape().as_list()

    def _inception_module(self, inputs, filters, name):
        if len(filters) != 6:
            raise ValueError('Invalid filters input')
        # From left to right in the graph @ https://arxiv.org/pdf/1409.4842.pdf fig.3
        with tf.name_scope(name):
            conv1_k1_s1 = self.conv(inputs, filters=filters[0], kernel_size=1, stride=1,
                                    name=name + '_conv1_k1_s1')
            conv2_k1_s1 = self.conv(inputs, filters=filters[1], kernel_size=1, stride=1,
                                    name=name + '_conv2_k1_s1')
            conv3_k3_s1 = self.conv(conv2_k1_s1, filters=filters[2], kernel_size=3, stride=1,
                                    name=name + '_conv3_k3_s1')
            conv4_k1_s1 = self.conv(inputs, filters=filters[3], kernel_size=1, stride=1,
                                    name=name + '_conv4_k1_s1')
            conv5_k5_s1 = self.conv(conv4_k1_s1, filters=filters[4], kernel_size=5, stride=1,
                                    name=name + '_conv5_k5_s1')
            pool1_p3_s1 = self.max_pool(inputs, pool_size=3, stride=1, name=name + '_pool1_p3_s1')
            conv6_k1_s1 = self.conv(pool1_p3_s1, filters=filters[5], kernel_size=1, stride=1,
                                    name=name + '_conv6_k1_s1')
            tensor_list = [conv1_k1_s1, conv3_k3_s1, conv5_k5_s1, conv6_k1_s1]
            return tf.concat(tensor_list, axis=3, name=name + '_merge')
```

Some more code examples can be found at the following GitHub repository: `https://github.com/Kaapp/chestxray-cnn-thesis-example`. It contains an example fresh and pretrained model, and the relevant training and benchmarking scripts. It does not include model checkpoints due to their size.