Deep Learning for Medical Image Classification

Benjamin Keel

Project submitted to the School of Mathematics

University of Leeds

Contents

1 Abstract

Deep learning methods are a group of algorithms which can be used to model non-linear relationships in a high number of dimensions (i.e. at a deep level) between input variables. These models can be trained to 'learn' patterns between labelled input variables and make predictions/classification on unlabelled input variables. Specifically, these algorithms learn from input data through a process of trial and error, whereby the model can adjust itself in accordance with epochs of training. This is achieved by function that calculates the difference between the estimated and true labels of data, called a 'loss function'. Deep learning is a branch of machine learning whose models utilise much larger amounts of data and requires more computational power but can discover complex patterns across and between variables. As such, they can be used for solving complex classification problems such as patterns within language (speech recognition), video (driverless cars) and images (medical image diagnosis). In this study, we build and assess the ability of a various deep learning methods in the classification of medical images for use in diagnostics.

2 Introduction

Medical imaging currently exists as an important tool for correct diagnosis of various conditions and it is estimated that around 90% of all medical data is in image form [1]. Typically, these images have to be analysed and classified by eye, and the accuracy of the diagnosis can depend on a doctor's experience. In some hospitals radiologists are tasked with looking at thousands of images daily, resulting in many images not receiving enough attention and thus not being analysed in a uniform way which can affect the detection of serious conditions. Machine learning methods, and more specifically deep learning models can be used as both a substitute or an important supplement for this detection as fundamentally, they can improve themselves with extensive trial and error - known as model's training.

In this study, we will explore the different techniques for deep learning and how to apply these concepts to real medical image data to alleviate the increasing demand which is causing a shortage of qualified radiologists in the UK. Of course, deep learning algorithms need to be incredibly accurate to warrant their use, especially in medical situations where accurate diagnosis is paramount. The performance of these models however depends heavily on the quantity and quality of data available. This is one of the key issues with the medical image domain due to scarcity of large datasets with accurately labelled images.

In this report, we will first introduce foundational concepts of both machine and deep learning including: supervised and unsupervised learning, classification, dimensionality reduction techniques, clustering and artificial neural networks. Secondly, we will investigate the feasibility of 'transfer learning', a method of transferring an effective model built for one purpose for use in another purpose. Thirdly, we will apply data augmentation to the input set, where data is created synthetically to improve the quantity and thus diversity of the input in the model which is an effective tool to improve performance. Transfer learning and data augmentation will be used in combination to explore the effectiveness of using an existing model, pre-trained on a non-medical domain for a medical image classification model. Finally, this report will evaluate the capability of various methods of deep learning for use in classifying images in a medical realm. This proof of principal will be demonstrated by the construction of a high-performance binary classification model to discriminate between chest and abdominal X-rays. The model will be trained using a small dataset of 75 chest and abdominal X-rays, in the framework of open source machine learning libraries. Further, we will apply the same network architecture for the purpose of diagnosing COVID-19 from chest X-rays to demonstrate the ability of deep learning models to direct themselves for more specific medical image analysis on the same type of image.

3 Background

3.1 Image Processing

Image files form the basis of the data used in this project. Image files themselves can be converted into 3-dimensional matrices of pixels and each pixel have red, green and blue values (RGB) ranging from 0 to 255. The combination of these three values gives the colour of the pixel. In computational terms, the data of an image is expressed as a 3-dimensional *'tensor'* which has shape (image width, image height, colour channels (RGB)). Machine learning methods can be used to analyse this data to find underlying structure and patterns within these matrices of pixels and across the RGB values within them. Greyscale images have all three RGB values the same, when this is the case a greyscale is produced where 0 is pure black, 255 is pure white and numbers in between are different shades of grey. Colour images will have RGB values which range across a spectrum where all three values can be any number in the range. For the purpose of image classification tasks, features are measurable quantities which can be anything in an image. They are represented by the RGB values of an individual pixel or combinations of pixels, which can be any area of an image. Deep learning methods can be used to encode pixels into 'generalised edges' and then objects to be recognised in images across a dataset. For instance, an image recognition model might recognise a nose and two eyes and then be able to classify that the image contains a face based on generalised edges across pixels in an image.

3.2 'Learning' Within Machine Learning

Machine learning and deep learning algorithms 'learn' from the training dataset using a variety of methods falling into the following categories: supervised learning, unsupervised learning and transfer learning. Firstly, supervised learning is the task of finding the function mapping an input to an output, therefore the output is known. When the output is known, the data is said to contain 'labels' which occurs if each input (vector) in the dataset has a piece of information (the label) attached to it classifying what the observation actually is. In supervised learning, an algorithm iteratively predicts an output and checks the label to see if it makes a correct prediction and then improves its prediction on the incorrect labels using an 'optimisation algorithm'. This process of self-improvement or learning via an optimisation algorithm is known as 'back propagation', where a model's parameters are slightly tweaked or tuned each time the model makes an incorrect prediction on a piece of data. This is done to iteratively reduce the level of error in the model and the process terminates when a suitable level of performance is achieved. Commonly, supervised learning techniques include regression and classification tasks. Regression is primarily used for modelling the relationship between independent variables for use in forecasting and producing quantitative predictions. Whereas classification is used for attaching qualitative labels to data, which requires a more complex analysis. Classification tasks, and more specifically medical image classification tasks will the focus of this report and will be used to predict the diagnosis of medical images. Broadly, classification task are broken down into categorical, where the data is put into multiple classes, or binary where the model will distinguish inputs between two classes e.g.'disease' or 'no disease'.

Unsupervised learning allows an algorithm to train the model on incomplete data without predetermined classes i.e. where the output is not known. This involves identifying broad and casual patterns within the input data. Learning is called 'unsupervised' when the algorithm determines the interdependent structure between input variables and will find any probability distribution that represents their spread in data space. Key techniques of unsupervised learning include clustering and the discovery of principal components within data. In this report we include

Principal Component Analysis (PCA), t-Distributed Stochastic Neighbour Embedding (t-SNE) and Cluster Analysis including the Hierarchical and K-means algorithms. These methods are forms of dimensionality reduction which involves breaking down a dataset into a simplified form or into a lower dimension whilst retaining the key features which optimally represent the data. For instance, changing the quality of 1080p video to 360p is a form of dimensionality reduction which retains the broad visual information of the data but reduces the size of the video file.

Transfer learning is the task of extracting the knowledge of previously trained algorithm to improve the performance of a new and related supervised learning task. This is carried out through the transfer of learned parameters and features of a pre-trained 'source' model to a newly constructed model. Transfer learning methods are usually employed in cases where the training data available is limited which impedes a model's ability to generalise a mapping function. This is particularly relevant for medical classification tasks, where the availability of data is restricted. It will be discussed in this report how knowledge can be transferred to a specialised classification model from a source model with a wider and more diverse domain with a large number of classes. More specifically, how transfer learning of a pre-trained source model can be applied to medical image domains, in situations where the data available would otherwise not be sufficiently large to train an accurate model.

3.3 Classification Modelling

Classification in a deep learning network is the process of predicting the class that given data points belong to, this is a form of supervised learning where the algorithm is provided with labelled data to start with. Classification modelling is the task of approximating a predictive mapping function.

$$f : X \to Y$$

Here, f is the function which maps the input set X to the output set Y, which the contains the predicted labels. A deep learning algorithm takes training data consisting of pairs of individual data $\{x_i, y_i\}$, where $x_i \in X$ and $y_i \in Y$. The learning task $\mathcal{T}$ summarises what the algorithm intends to do, which is to optimise the mapping function $f(\cdot)$ using the labels Y of some known data X. This formalised as

$$\mathcal{T} = \{Y, f(\cdot)\}$$

A deep learning algorithm takes the training set X_{train} with known labels Y_{train} and produces a prediction Y_1. The model then iteratively tunes its parameters using an optimisation algorithm so that f more accurately predicts the labels of X_{train}. The performance of the models can then be tested using a second set of inputs X_{test} with labels Y_{test} unknown to the model. The predictions Y_2 of the data X_{test} are then also checked with the testing labels Y_{test}. Since these labels are not known to the model, the accuracy of Y_2 acts as an indicator of how well the model would perform on new observations. After this training process the mapping function $f(\cdot)$ can be used to predict the label $y = f(x)$ of a new observation x.

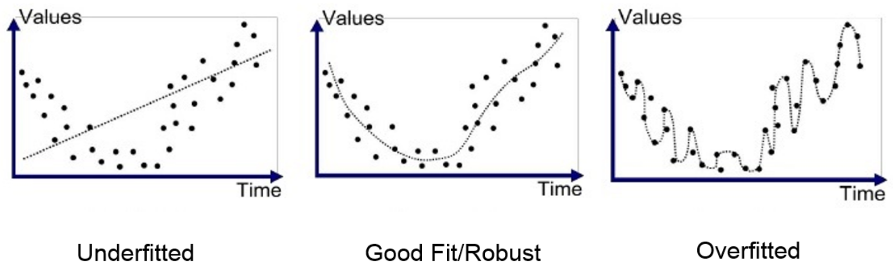


Figure 1: Diagram of fitting a model from [2]

If the model is not performing correctly then it can be assessed whether f has been over or under-fitted, that is the model has incorrectly generalised the data. A visual representation of how the mapping function f can be incorrectly fit is shown in Figure 1. When the model is under-fitted during the training, a simple trend is fitted to the data which over generalises the data (see 'Underfitted' Figure 1). When this is the case, f would fail to give accurate predictions of the labels of both X_{train} and X_{test}. On the other hand, when f is over-fitted it 'forces' the polynomial through too many of the points which can also lead to poor performance on unseen data (see 'Overfitted' Figure 1). This is because the model doesn't generalise the data enough, it learns the training set specifically which means that it is not flexible enough to model unseen data. A model can be seen to be over-fitted when the training accuracy is high, but testing accuracy is lower. There exists an optimal zone where the model is not too finely or too loosely tuned, which we aim for when training deep learning algorithms. The model is a 'good' fit (see 'Good Fit/Robust' Figure 1) when the difference between the level of error of the two data sets is minimal. This is when the difference between the proportion of incorrect labels of Y_1 and Y_2 is as small as possible, or geometrically where the accuracy curves follow the same trend.

3.3.1 Transfer Learning

Transfer learning is a machine learning technique used to improve the accuracy of models by a transfer of learned parameters, from a source model repurposed in a new model for a second more specific task. This can be seen as fine tuning a neural network pre-trained on a generalised dataset, directing the model towards a subset of its domain. Formally, a machine learning model M has the domain

$$\mathcal{D} = \{\mathcal{X}, P(X)\}$$

which consists of a feature space $\mathcal{X}$ and a marginal probability distribution $P(X)$ [3]. The feature space $\mathcal{X}$ is the geometric space which contains all possible data in the domain of the model. This is the n-dimensional space where the observed data are situated and new data could be expressed in. Also, the domain has a marginal probability distribution $P(X)$ which is used to initialise the weight parameters of the model. The probability distribution is marginal based on the assumption all new observations will be independent and identically distributed. It is used as a description of how likely a new instance x is to take values in the label space $\mathcal{Y}$. Here the observed data $X = \{x_1, ..., x_n\} \in \mathcal{X}$ is the dataset to be used for the new task. This could be used in the application of a machine learning model M, which is used to classify different types of animals in the feature space $\mathcal{X}$. Then through the transfer of learned parameters in M a new model is trained for a new purpose on a dataset X containing only different breeds of dog. A transfer learning task

$$\mathcal{T} = \{\mathcal{Y}, f(\cdot)\}$$

is similar to a classification learning task, it requires the label space $\mathcal{Y}$ of the data space $\mathcal{X}$ and a predictive mapping function $f(\cdot)$. This can be rewritten in conditional probability terms as $P(Y|X)$, the probability that the label will be $y_i \in Y$ given the data $x_i \in X$.

$$\mathcal{T} = \{\mathcal{Y}, P(Y|X)\}$$

Given a pre-trained source model M_S with the domain $\mathcal{D}_S$ and its learning task $\mathcal{T}_S$, which has optimised parameters. A new model M_N can be trained using knowledge in $\mathcal{D}_S$ and $\mathcal{T}_S$ [3]. M_N has the 'target' domain $\mathcal{D}_N$ and its own learning task $\mathcal{T}_N$, transfer learning improves the learning rate η of the target predictive mapping function $f_N(\cdot)$ in $\mathcal{T}_N$. Note also that the domains and learning tasks of both models are not equal [3]. This means in practice that the transfer learning is used to adapt the marginal and joint probability distribution of the data in the source model to optimise the performance of a new model. This is achieved by minimising the differences between these probability distributions which maps the larger domain $\mathcal{D}_S$ to $\mathcal{D}_N$ and the generalised task $\mathcal{T}_S$ to $\mathcal{T}_N$.

Starting a new model from scratch may require complex datasets and large computational resources which can be difficult to obtain, making the construction an accurate algorithm a huge task. Transfer learning provides a convenient approach to deep learning; the knowledge gained from the source trained model M_S can be passed on to improve the accuracy of a similar model. Feature maps learned from M_S can be used to extract meaningful features from new samples. These may be features representations generically applicable to classifying images which can yield higher initial accuracy for the new model M_N. The new model classifier is trained from scratch and is specific to the task required. The source model could be trained on a general image domain and mapped to a medical image domain for the new model to be trained on top. For example, in the methodology section, the Inception V3 model is used as a source model repurposed for a medical image binary classifier. This source model is trained on the ImageNet database which a natural image dataset containing a huge range of 22,000 classes and is reduced to just 2 classes for the new learning task. Transfer learning can improve the overall limit of accuracy since the model will be more skilful than it otherwise would be from training parameters from scratch. This is particularly true in cases where the size of the training set is relatively small. Further, transfer learning can be easily applied in the framework of machine learning libraries and there exists a plentiful supply of accurate source models which are available for public use. These models are the product of a network of machine learning developers which are actively producing skilled models with high level learned features. The issue with transfer learning is that it is not guaranteed to improve the performance of a given model. The choice of a source model can be an initial hurdle, where its suitability to fit a particular problem can require intuition and knowledge to map the domain of the source model to the target problem.

3.4 Dimensionality Reduction

Dimensionality reduction is the process of reducing the number of parameters (or dimensions) within an input dataset. In the case of deep learning classification models, this can be used for a few purposes. Firstly, to reduce the time needed to train a deep learning algorithm which is achieved by finding a lower dimensional representation of the data. For example, reducing a high-definition image to a blurrier one, whilst keeping most of the important information in the image. Simplifying the data and reducing the number of variables reduces the computational complexity of the model, which in turn allows the model to train faster. This is because there are less inter-variable relationships to be considered in the analysis. Secondly, dimensionality reduction can be used as a means to visualise high dimensional data and to identify underlying spatial structures and patterns in otherwise unstructured datasets. Dimensionality reduction algorithms can be used to accurately group observations, which is useful in the medical image domain as access to labelled data in the real world can be difficult to obtain. Further, visualisation of high-dimensional data in two or three dimensions can help a deep learning developer obtain an understanding of the data and how it is distributed in high dimensions.

One type of dimensionality reduction that will be discussed is feature extraction. This is the process of reducing the dimensionality of the data by creating new independent features (variables) which optimally represent the information of the data in a fewer number of (independent) input variables. These new features are combinations of the old independent features using linear or non-linear transformations of the data. PCA and t-SNE are two types of dimensionality reduction algorithms which can be used to simplify high dimensional data to reduce the size of input vectors in the training of a model. Further, we will discuss the use of clustering algorithms to group similar observations in high-dimensional space into a smaller set of classes. These unsupervised learning methods help to bring structure to datasets and remove statistically redundant components and information which allows the use of effective supervised learning methods.

3.4.1 Principal Component Analysis

Principal Component Analysis (PCA) (first introduced by Karl Pearson in 1901 and then improved on by Hotelling in 1933) is a linear algorithm used to simplify high dimensional data whilst retaining trends and important patterns present in the data. PCA is a form of feature extraction where a set of variables are reduced down into a new set of independent features known as Principal Components (PCs). The PCs are found by conducting a linear orthogonal transformation which fits a k-dimensional ellipsoid to the data, this means geometrically projecting the data onto lower dimensions.

For this transformation we need to compute a series of mathematical products from the data which are vectors $\mathbf{x}_i \in X$. For indexing purposes, we let d be the number of numerical dimensions of the data (excluding any non-numerical labels present) and D = [1, d]. In image processing this is the size of the tensor which contains the RGB values for each pixel in an image for a number of images. For each $i \in D$ we calculate the mean μ_i of each observation $\mathbf{x}_i$ and then centre the data at 0 by subtracting the mean from each observation.

$$\mathbf{x}_i - \mu_i$$

Then we can calculate the covariance matrix Σ and find its eigendecomposition, from which we obtain the eigenvectors $\mathbf{v}_i$ and corresponding eigenvalues λ_i. Note that the eigendecomposition is used since it produces orthogonal vectors which optimally represent the information of a dataset in a smaller number of vectors. These eigenvectors are then sorted into order by decreasing eigenvalue as follows.

$$\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_\mathrm{d} \geq 0$$

The eigenvalues λ_i are ranked here in order of importance, by the amount of information they represent. We choose a target dimension k with $k < d$ and keep k eigenvectors of largest eigenvalue to capture a desired amount of variance e.g. 90%. These eigenvectors $\mathbf{v}_i$ correspond to the k principal components we wish to find [4]. By choosing eigenvectors of largest eigenvalue, we maximise the total variance captured for each successive PC. We have that the sum of eigenvalues is equal to the sum total of the variance or equivalently, the trace of the covariance matrix Σ.

$$\sum_{i=1}^{d} \lambda_i = \sum_{i=1}^{d} \mathrm{Var}(X_i) = tr(\Sigma)$$

Then we have that the percentage of variance captured in our k Principal Components can be summarised as the proportion of the sum of eigenvalues.

$$\frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{d} \lambda_i}$$

Note that this is an orthogonal transformation to new basis vectors, so the correlation eigenvectors must be normalised so that they have unit length which is achieved by dividing $\mathbf{v}_i$ by its absolute value.

$$\hat{\mathbf{v}}_i = \frac{\mathbf{v}_i}{|\mathbf{v}_i|}$$

These $\hat{\mathbf{v}}_\mathrm{i}$ for $i \in D$ are the new bases in the transformed system. Then we can form a $k \times d$ matrix, W which contains the k chosen eigenvectors concatenated into this matrix. This is the instructions for the linear orthogonal transformation. We take the transpose of W, so that the column eigenvectors, $\hat{\mathbf{v}}_\mathrm{i}$ become the rows in the new matrix W^T. Then in matrix multiplication the row eigenvectors in W^T will be correctly multiplied by the data column vectors $\mathbf{x}_i$. We perform the principal component transformation using the formula

$$X_{PCA} = W^T \cdot X$$

where X_{PCA} is the dataset X in the transformed subspace with the k dimensions (PCs) [5]. This transformation ensures that the first principal component, PC_1 is chosen to minimise the squared residual between the original point $\mathbf{x}_1$ and its dimension reduced approximation $\mathbf{x'}_1$ [4]. Further components are chosen in the same way but with the condition that they are uncorrelated with previous PCs, meaning geometrically that the PCs are orthogonal to each other. We have that the maximum number of PCs is either the number of samples in the dataset or the number of features (variables), whichever number is smaller. The desired result of this analysis is to find the best summary of the data using as few principal components as possible.

3.4.2 t-Distributed Stochastic Neighbour Embedding (t-SNE)

t-Distributed Stochastic Neighbour Embedding (t-SNE) (Maaten and Hinton, 2008) is an extension of the Stochastic Neighbour Embedding algorithm (Hinton and Roweis, 2002). It is a non-linear algorithm for dimensionality reduction, in contrast to the linear technique PCA. The technique visualises high-dimensional data $X = \{x_1, x_2, ..., x_n\}$ by giving each data point a location in a lower-dimensional space $Y = \{y_1, y_2, ..., y_n\}$. This could be into two or three dimensions, in which case they can be visually displayed in a scatter plot. The algorithm begins by calculating the probability that points x_i and x_j are similar in high-dimensional space, p_{ij}. As well as the probability of the lower dimensional projections y_i and y_j being similar in a low-dimensional space q_{ij}. The similarity is calculated as the conditional probability $p_{j|i}$ that x_i would choose point x_j as its neighbour, if neighbours were chosen in proportion to their probability density under a Normal Distribution (Gaussian) centred at x_i. Similarly this is the case for $p_{i|j}$ with i and j were swapped around. Also, note that $p_{i|i}$ (and $q_{i|i}$) is set to zero as we are modelling pairwise similarity only [6]. Here the similarity calculation for $p_{j|i}$ is shown below.

$$p_{j|i} = \frac{\exp(-\|x_i - x_j\|^2/2\sigma_i^2)}{\sum_{k\neq i}\exp(-\|x_i - x_k\|^2/2\sigma_i^2)}$$

Where σ_i is the variance of the Gaussian that is centred at x_i. Then the average of the two conditional probabilities is taken.

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}$$

For the lower dimensional projections, the propability of similarity between y_i and y_j is given by.

$$q_{j|i} = \frac{\exp(-\|y_i - y_j\|^2)}{\sum_{k\neq i}\exp(-\|y_i - y_k\|^2)}$$

We also take an average of the lower dimensional conditional probabilities $q_{j|i}$ and $q_{i|j}$ to obtain q_{ij}. Note that $q_{j|i}$ is not divided by the variance of the Gaussian. In the high-dimensional space, close data points x_i and x_j will have a high conditional probability $p_{j|i}$. Whereas, for distant points $p_{j|i}$ is very small and the same is true for close and distant low-dimensional projections. This means that by minimising the difference between the two measures of similarity p_{ij} and q_{ij}, the low-dimensional space Y will be the best representation of the data into the given lower dimension. The standard SNE algorithm will attempt to do this by minimising the sum of Kullback-Leibler (KL) divergence of overall data points using a gradient descent method. KL divergence a loss function 'L' used as a measure of how one probability distribution diverges from a second one [6]. In this case, it is how the high dimensional distribution P differs from the low-dimensional distribution Q.

$$L = D_{KL}(P||Q) = \sum_{i\neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

The issues with the original SNE algorithm are that it can be computationally inefficient to optimise this loss function and also we are faced with the ‘crowding problem’. This is a problem that arises because the area in a low dimensional space that represents the low-dimensional projections of distant data points will not be big enough relative to the area representing nearby data points. These issues are improved upon in t-SNE where the algorithm will compute a symmetric version of D_{KL}. The new version has simple gradients and it applies a heavy tailed Student-t distribution with one-degree of freedom (which is equivalent to a Cauchy distribution), rather than a Gaussian to compute similarity in lower-dimensional space q_{ij}. This alleviates both the crowding problem and the optimisation problems of SNE [7]. Applying this distribution, the formula to calculate low dimensional similarity, is redefined as.

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq i}(1 + \|y_i - y_k\|^2)^{-1}}$$

The gradient of the symmetric KL divergence between P and the Student-t based joint probability distribution Q is calculated as.

$$\frac{\partial L}{\partial y_i} = 4\sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + \|y_i - y_j\|^2)^{-1})$$

The gradient of the symmetric KL divergence shown here is easier to minimise than the original KL divergence in the SNE algorithm. The full algorithm and derivations of these formulas can be found at [6]. In this report, we utilise t-SNE as a data visualisation and exploration technique. After this algorithm as has been applied you cannot infer the distances between or density of the points in X from their projections in Y. That is the high-dimensional input features are no longer identifiable. t-SNE will be used for finding patterns and clustering points with multiple features to indicate how the data is distributed in its original dimension.

3.5 Cluster Analysis

Clustering is the task of finding structure between variables within a given number of dimensions e.g. a cluster of jam in a doughnut in 3-dimensions would have central x, y, z values. Cluster analysis is an unsupervised learning method and is used in identifying groups in unlabelled (unstructured) input data. Clustering algorithms work to find similar attributes between data points, this is also a form of feature extraction where new independent classes are created. The data is partitioned using a measure of similarity and is done to allow supervised learning to be conducted on the newly labelled data. There are multiple clustering algorithms which can be used to perform this analysis, however we will consider Agglomerative Hierarchical Clustering (Ward, 1963) and K-means Clustering (MacQueen, 1967).

3.5.1 Hierarchical Clustering

In hierarchical clustering, a network is created with the nodes representing each data point and then each node is iteratively merged based on pairwise distance. The two most efficient algorithms for calculating this are complete linkage and single linkage clustering which return the maximum or minimum pairwise distances respectively [8]. These methods for calculating similarity are optimally efficient and have time complexity of O(n^2). The algorithm starts by assigning each data point to its own cluster and then considers each cluster in turn and merges it to its closest neighbour in the data space. This process is continued until all the data is in one cluster. This form of clustering is hierarchical in nature because each successive parse of the algorithm creates taller bars representing lower levels of similarity between data within the clusters. The output is a tree structured graph of the clusters which is called a dendrogram.

Similarities between data can be calculated using the Euclidean distance measure

$$d_E = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

which derives from the Pythagorean theorem. This is just one of multiple similarity calculations which can be employed. Once a dendrogram plot is produced, it's up to the model author to find the balance between level of similarity and number of classes they wish to use. To partition the data, a cut can be taken at a fixed height on the dendrogram and the resulting subtrees below the cut are the chosen clusters. This cut is taken to spilt the largest vertical distance between horizontal lines. It's important to note that conclusions are based on the vertical (similarity) y-axes exclusively rather than the symmetry or the structure of the tree. The x-axis shows the data observations ordered so that each point is adjacent to its closest neighbour. The order of the datapoint and thus the appearance of the dendrogram can change based on the calculation used for similarity [9].

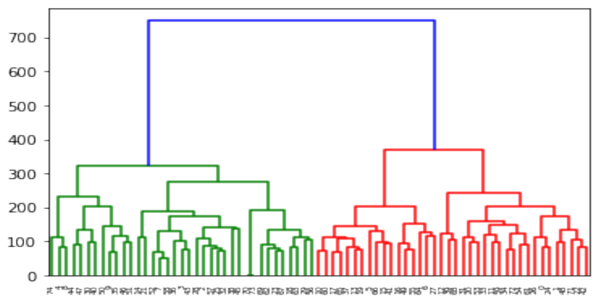


Figure 2: Dendrogram produced from X-ray data

In Figure 2 we have the dendrogram I produced using the X-ray images used in the binary classification model in section 4.1. Looking at this this graph, the highest vertical distance that doesn't intersect with any clusters is the top blue one. Given that there are 2 vertical lines which cross this threshold, the optimal number of clusters is 2 which supports the use of a binary classification model on this data.

3.5.2 K-means Clustering

Another method of clustering we will consider is k-means clustering which is easy to implement and is more computationally efficient than hierarchical clustering. The time complexity of k-means is linear with O(n), while hierarchical is cubic time complexity O(n^3) - though with the use of optimally efficient methods this can be reduced to O(n^2). Consequently, the time taken to carry out cluster analysis grows at an increasing rate for large datasets (large values of n). Thus, it can be more practical to use the k-means algorithm for big data samples instead of more computationally complex techniques. The methodology of k-means clustering is as follows; firstly choose the desired number of clusters k and then randomly select k values to be the 'centroids', these are the initial cluster means μ_j. The centroids may be either randomly selected observations from the dataset or randomly generated within the range of the data [8]. Secondly, calculate the similarity between each data point x_i and each centroid μ_j, for $i \in [1, n]$ (the number of observations) and $j \in [1, k]$ (the number of clusters). The similarity is usually calculated using Euclidean distance d_E as in hierarchical clustering. Thirdly, all the points are grouped with their nearest centroid to form clusters, where the proximity is based on the similarity calculation. K-means clustering aims to minimise inertia, the sum of squared error between data values and the centroid within each cluster. This is the sum

$$\sum_{i=1}^{n} \min_{\mu_j \in C}(||x_i - \mu_j||^2)$$

On this basis, the algorithm next reassigns the position of the k centroids to the middle of each cluster. It does this by calculating the mean of the observations in each cluster and replacing each centroid with the cluster means for all j. The clustering algorithm iterates this step until the centroids which minimise inertia have been found, i.e. when the centroid values no longer change from running this calculation. Finally, cluster quality can then be checked using two measures: 'within cluster similarity' and 'between cluster similarity'. Within cluster similarity should be high since the clusters should group highly-similar data and between cluster similarity would ideally be low so that there is minimal overlap between groups [8]. The most important factor in cluster quality is the number of clusters chosen, k. Since there is no exact method for determining the optimal k it's important to test a range of k values and use quality measures to compare the resulting clusters produced. There are a number of intuitive ways to attempt to find an optimal k, for example plotting the mean distance to the centroid as a function of k. This distance will decrease as number of clusters k increases. We can then locate the 'elbow point' of this graph, where the rate of decreasing distance to the mean shifts sharply, this point on the graph can be used to roughly determine the optimal k [10].

3.6 Neural networks

One branch of deep learning concerns itself with multi-layered artificial neural networks (ANNs), these are networks built up of layers of nodes or 'neurons' connected by weighted edges or 'synapses'. They are used for classification or regression modelling purposes where predictions are formulated on data. A basic ANN is characterised as an algorithm which models the behaviour of biological neurons in a human brain. That is, a network which can process information through the combined output of individual neurons connected by adjustable synapses. Also, neural networks can learn through iterative training which can be thought of as experience. ANNs learn a dataset better as the optimisation algorithm changes the weights of the connections between the nodes (the synapse), which gives the model a more accurate prediction of the outputs. A simple 'feed-forward' ANN is comprised of a network containing an input layer, multiple 'hidden' layers and an output layer.

The input layer is simply the x amount of input variables represented as x amount of nodes. In the hidden layers each node of the network has an 'activation' and each edge which connects these nodes have parameters with weight and bias which are applied to an input variable. Mathematically, the parameters $\Theta = \{W, B\}$ are such that W is the set of weights which are applied to their corresponding nodes and B is the set of biases which are applied to their corresponding layers. Each node in the hidden layers of the network can be said to be active or inactive. An active node is one that will pass information forward to the next layer after information reaches it and an inactive node will stop the information from passing on. Whether the node is activated or not is determined by an activation function $a(x)$. The activation function transforms the information input and determines whether information is fed forward or not based on a loss function. Information that passes through all the hidden layers reaches the output layer at the end of the network. At this final layer, all information is fed into a final node which formulates a prediction on the input data.

In this report, we focus on deep learning for classification modelling, where a neural network algorithm is used in supervised learning to classify inputs by attaching a label to them with a stated level of confidence. The supervised learning or training of an ANN consists of; exposing the network to known labelled training data, using the ANN to predict the labels and then improving the accuracy by optimising its parameters using an optimisation function. This process is iteratively carried out on the whole training set a number of times (epochs) and a loss function is implemented to quantify the level of error which can then be back propagated toward a minima to reduce the error. ANNs can come in a variety of structures which can perform classification tasks, we first consider the Single-Layered Perceptron (SLP), the characteristic form of an ANN. Then we will discuss the Multi-Layered Perceptron (MLP) which is a typical ANN with added hidden layers. We will also formally define the process of back propagation where the gradient of the loss function is used to iteratively update and optimise the weights of the nodes in ANNs. Lastly, we will consider the use of Convolutional Neural Networks (CNNs) in the context of medical image classification. CNNs are a form of ANN, most suited to image classification tasks, uniquely they employ the use of convolutional filters to extract spatial information from inputs. This is achieved through building up composite feature representations of image data or 'feature maps', which can be used to recognise similar spatial patterns in other images across the domain.

3.6.1 The Perceptron

The perceptron is a supervised learning procedure which classifies output by performing an algorithmic calculation at each node. In its characteristic form, the perceptron is a binary classifier, known as the Single-Layered Perceptron (SLP). Named so on account of having one layer of links, between the input and output. The SLP decides whether an input belongs to a certain class or not by outputting a 1 or a 0. The values produced from the first layer of nodes are fed forward into the output layer via links known as weighted synapses. These edges come with the parameter set $\Theta = \{\theta, b\}$ of weights $w_i \in \theta$ and a singular bias b. There is also a threshold activation function which decides whether each node in the layer is activated or deactivated. This determination is similar in biology to whether the neuron is said to fire, based on if the values are above a certain threshold.

The perceptron formalises the 'learning' process previously described; approximating a mapping function which maps the data X to a label in Y.

$$f : X \to Y$$

Each perceptron takes a input vectors $x_i \in X$ for $i \in [1, n]$ and assigns weights $\theta = (w_1, w_2, ..., w_n)$ to each input. Next, a combination function $\sum_{i=1}^{n} x_i w_i$ is used to produce a weighted sum of the input values. The formula also has a bias b added, which shifts the activation function to the left or right to fit the data better. The bias b is usually initialised at 0 and then adjusted automatically along with the weights during the tuning of the model. This value 'α' is then interpreted by an activation function which decides whether a node should be activated. Mathematically, we can describe this as a function a with the threshold '$fire$' that decides whether or not the information α is fed forward.

$$a(\alpha) = \begin{cases} \alpha & \alpha \geq fire \\ 0 & otherwise \end{cases}$$

All together we have

$$\begin{aligned} y &= \sum_{i=1}^{n} a(w_i^T \cdot x_i + b) \\ &= a(W^T \cdot X + b) \\ &= a(\alpha) \end{aligned}$$

In Figure 3 below, we can see a generic structure of an SLP with three input variables showing graphically the order of this formula. Thus, the perceptron approximates the mapping function f, that produces the value y which is a label in Y.

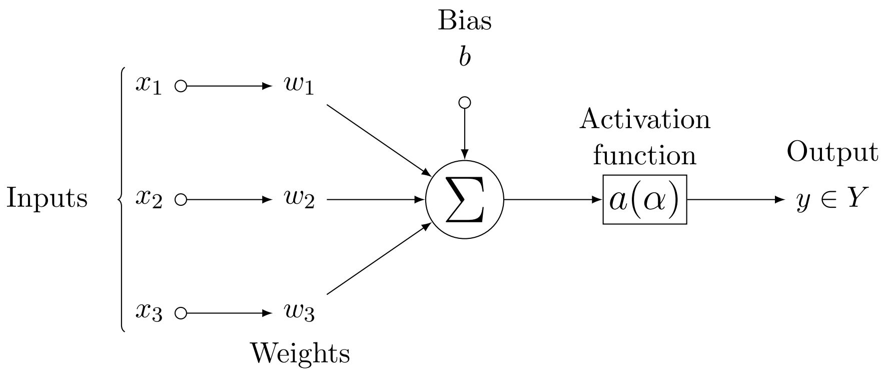


Figure 3: Diagram of a Single Layered Perceptron (SLP)

3.6.2 Multi-Layered Perceptron

The MLP is an extension of the SLP where it is subject to several layers of these transformations. It has the parameter set $\Theta = (W, B)$, where the set W contains the set of weights for each layer and B contains the biases for each layer. Here $W = (\theta_1, \theta_2, ..., \theta_l)$ is the set of layer weights where l is the number of layers in the network and each θ_i contains the weights w_j for each node in layer i. Each layer has a unique bias b_i added from the set $B = (b_1, b_2, ..., b_l)$ as well. In the MLP an activation function is applied at each layer which forms a combination of functions to produce a label in Y. To use the MLP for a binary classification model it's suitable to apply the sigmoid function for the activation on the output layer.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

This maps the data to values between 0 and 1 which can be used as probability or confidence score that an image falls into one of two categories. With the use of the sigmoid activation for all layers here, this calculation is formalised as follows [11].

$$f(x; \Theta) = \sigma(\theta_l \sigma(\theta_{l-1} ... \sigma(\theta_1 x + b_1) + b_{l-1}) + b_l)$$

The intermediate layers between the input and output are called the hidden layers as the values of these transformations are hidden. For example, at the second layer the value of

$$\sigma(w_1 \sigma(w_0 x + b_0) + b_1)$$

is not shown to the model author. Neural networks with multiple hidden layers are considered 'deep' networks as the analysis is conducted through a long string of transformations. These intermediate layers in the MLP provide the model with more flexibility as there are more trainable parameters which can be utilised in optimisation to give more accurate predictions.

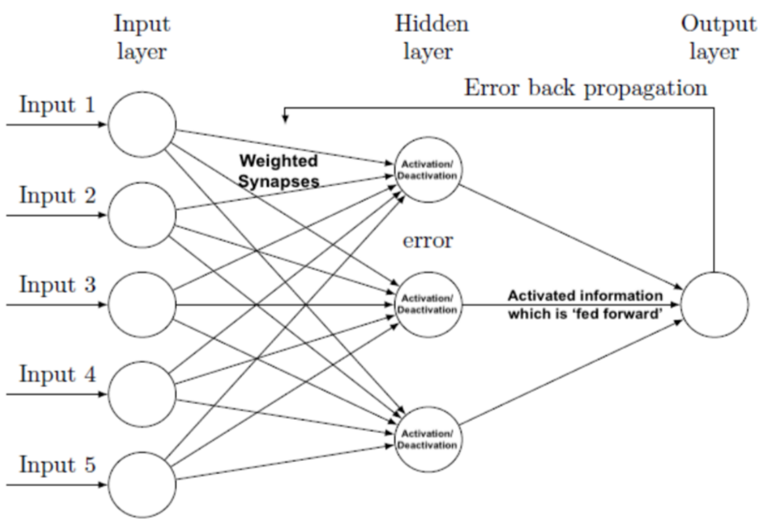


Figure 4: Structure of a generic feed-forward Multi-Layered Perceptron

Figure 4 above shows a simple MLP with one hidden layer, it represents graphically the learning process of feed-forward neural networks. After one full parse of the data, the weighted synapses are error adjusted using back propagation which is a supervised optimisation algorithm. This supervised training process is carried out iteratively until the error is suitably reduced. This is when the outputs labels Y_1 closely resemble the labels of X_{train}. As discussed before, it is important to not over or under fit the MLP mapping function f as it leads to higher levels of error on X_{test}, which indicates that it will perform poorly on unseen data. If the training phase is successful, the model will perform accurately on both the training and testing data which means it has generalised the domain and so the model can used for the desired purpose it was built for.

3.6.3 Activation Function

It is also important to discuss the use of different activation functions and their relative effect on the training process. Machine learning models without activation function are essentially just linear regression models, the purpose is to introduce non-linearity into the output of a node. This makes the model capable of learning a more complex task as its gradient can be back propagated towards a minima in the model tuning. There are a variety of activations which exist, each with their own benefits and drawbacks, examples include; the sigmoid function, softmax and ReLU activations. The sigmoid $\sigma(x)$ and softmax are typically used only in the output layers of ANNs as the values they produce can be used as a confidence score that an image falls into a certain category. The sigmoid function as stated before maps values to between 0 and 1, which can be used to distinguish between two classes in a binary classification model. Whereas the softmax function can be used to represent a categorical probability distribution which distributes data into k given classes in a categorical classification model.

$$Softmax(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{k} \exp(x_j)}$$

Further, we can use different activations for the hidden layers of an ANN which can increase accuracy and speed of training. One such activation which can train at a faster rate than the sigmoid function is a ReLU activation which takes the maximum of 0 and x.

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

Layers with this non-linear activation are known as Rectified Linear Units (ReLUs). They train faster than sigmoid activation since for $x > 0$ the gradient is constant. Whereas in sigmoid activation, the gradient converges to 0 for large absolute values of x, since $\sigma(x) \to 1$. This means that ReLUs have a faster learning rate $\eta_{ReLU} > \eta_{\sigma(x)}$. However, ReLU activation should only be applied to the hidden layers of a neural network because it can produce dead neurons. This is where the output will always be 0 and not update during the training phase which can hinder the model's performance. Thus, it's most practical to use a combination of activation functions and also to test different types to see which work best to achieve the most accuracy.

3.6.4 Optimisation Algorithm

An optimisation algorithm has the task of reducing the level of error in the predictions of an ANN. This process is iteratively carried out by updating the model parameters by subtracting a scalar multiple of the gradient of the loss function. The scalar is the 'learning rate' η and is used to put a limit on how fast the parameters are updated, or in other words how fast the model learns. Updating the parameters in this way is known as the back propagation of the loss function, where the changes to the parameters move the loss function toward a minima. This has the effect of reducing the level of error in the output variables (predictions). The loss function measures the correctness of outputs and several different loss functions can be implemented. For example, the binary cross-entropy (BCE) and mean squared error (MSE) are two which we will discuss. The equations for the two loss functions are shown below, where $\hat{y}_i$ is the predicted label of input x_i and y_i is its true value.

$$BCE : L(\Theta) = -\sum_{i=1}^{n}(y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i)$$

$$MSE : L(\Theta) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

The BCE is a log loss function which assumes that the data come from a Binomial distribution where there are two classes and each random variable follows a Bernoulli distribution. This has the assumption that each random variable has the label

$$y_i = \begin{cases} 1 & \text{with probability } p, \\ 0 & \text{with probability } 1 - p, \end{cases}$$

The BCE is preferred for binary classification tasks because the loss function is convex over the range [0, 1] which means there is a unique minimum to back propagate towards. On the other hand, the MSE loss function is preferred for regression tasks as is assumes a normal (Gaussian) distribution which expects values in the range of all real numbers. However a binary classification outputs bounded probabilities between 0 and 1 through a sigmoid function, which might lead to a non-convex problem in this range. This means that an optimisation algorithm back propagating over the MSE could get stuck in a local minima, meaning that the loss is not properly minimised.

We assume that Y' is the set containing the predicted labels of X and Y is the true labels. The MLP mapping function is $f(x, \Theta)$, with parameter set $\Theta = \{W, B\}$ that includes the weight and bias parameters of the model. This function f finds the predicted labels for X by calculating $Y' = \Theta \cdot X$, note that this is a simplified format of this operations detailed in 3.6.2. Gradient decent is applied to Y' to adjust the node weights in the perceptron in order to optimise the model's accuracy. As training data is fed through the network we calculate the gradient of the loss function $\nabla L(\Theta)$ with respect to every weight, w in the parameter set Θ using the chain rule as follows.

$$\nabla L(\Theta) = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial w} = (\Theta \cdot X - Y) \cdot (X^T)$$

This is then iteratively minimised by changing the parameters $\theta \in \Theta = \{W, B\}$ as follows:

$$\theta^{(i+1)} = \theta^{(i)} - \eta \nabla L(\theta^{(i)})$$

where η is the learning rate and i is its iteration index [12]. The learning rate controls the speed of training, a large learning rate η will give more weight to $\nabla L(\theta)$ which means that the weights are updated at a faster rate. Different activation functions can alter the learning of a model regardless of the learning rate η. Therefore it's important to try a variety of different activation functions and learning rates to find the optimal combination.

3.6.5 Convolutional Neural Network

The next logical step in improving the performance of the model is to let the model learn the features that optimally represent the data, that is to incorporate feature extraction in an ANN. This concept is used in many deep learning algorithms where at each layer the model learns increasingly higher level features. The base feature representation of an image is the raw intensity (RGB or Greyscale) values at each pixel. Individual feature vectors can represent anything in the image ranging from one pixel to the whole image, for example the model might encode edges into feature vectors then create higher level features representing objects made up of these edges. A CNN will take image data inputs which are arranged in a grid structure to create a matrix with numbers denoting pixel values, the input is a 4-dimensional (4D) *'tensor'* with shape (number of images, image width, image height, image depth). A convolutional layer in neural network, contains filters created using a set of parameters $\Theta = \{W, B\}$ of weights and biases which are common for the whole layer. The number of filters applied to the image is also specified, this is the number of feature map channels. The input tensor is passed through the convolutional filters, with minimum kernel size 3 x 3, denoting the height and width of the filter. The filter height and width must be odd numbers and so larger filters such as 7 x 7 can be used. Larger filters will learn higher level features which recognise larger objects as individual features. The *'kernel'* is a matrix of a given size which is moved iteratively across the image and is multiplied with the input.

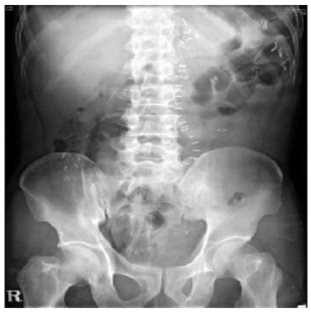
The algorithm calculates the dot product (matrix multiplication) with the first block, at the first placement of the filter on the image and stores its output in a tensor. Next, the convolutional filters are iteratively passed over the whole image storing the output from each placement of the filter, the image x becomes abstracted into a feature map of the image x_k. The feature map x_{k_i} for each image is then fed through a non-linear activation for example the sigmoid function $\sigma(x)$ as before. The output is then combined in a 4D tensor which has the shape (number of images, feature map width, feature map height, feature map channels). At the end of the convolutional stream, each feature map is pooled using a pooling layer which performs the process of taking a small grid or neighbourhood in the feature map and combining it into a single output. This usually via calculating the maximum number or the average number in the grid, which transforms the shape of the data into a 2D tensor with shape (number of images, feature map channels). This transformation is done so that it can be fed forward into the typical dense layers of an ANN which accept 2D tensors. Convolutional layers work similarly to fully connected dense layers, here the filters are the 'neurons' of the layer. They convolve the input and apply weight w, bias b and activation $\sigma(x)$ to the value and then feed forward the result to the next layer. This process is repeated for every convolutional layer 'l' [11].

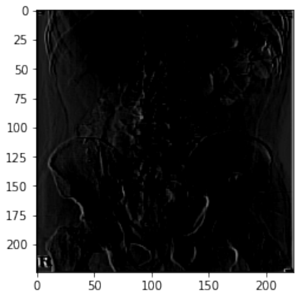
$$x_k^l = \sigma(w_k^{l-1} \cdot x^{l-1} + b^{l-1})$$

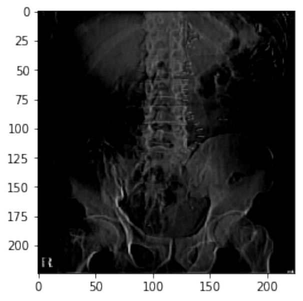
Here the feature map for the current layer l is x_k^l and is generated from the previous layer using the same process as in the MLP. The data from the previous layer (layer $l-1$) is x^{l-1} and it is multiplied by its weights w_k^{l-1} and has bias b^{l-1} added, then some activation is applied (sigmoid in this case).

CNNs are better designed for use on image classification, as the structure is better at recognising and utilising spatial information in images that may be lost in traditional ANNs. This is because in a regular ANN the image is flattened into a vector before being processed by the network, which means that statistical properties such as mean or variance have more weight than geometric properties. Whereas, in a CNN the convolutional filters keep the original structure of the image and apply filters to blocks of the image where the pixels are spatially-neighbouring. This allows the network to generate the feature map x_k which will recognise geometric patterns such as the edges and objects as well as areas with differences in the colour intensity which may indicate lighter and darker areas of the image. These features may be common across the whole training set and so help the model classify images into different categories based on this information.

The images below in Figure 5 are example features maps of (a), an abdominal X-ray taken from the X-ray data. These feature maps were obtained using kernel size 3 x 3 and weight parameters learned by a simplistic model with one convolutional layer and multiple filters. The model's parameters produce a spectrum of feature maps each with a unique representation of the image based on the learned parameters to extract different features. The feature map (b) shows a lower level representation where only some of the most prominent outside edges are encoded and the rest is dark. In (c) the image produced highlights the bone structure of the abdomen with the rest of the image blacked out, this could be especially important as this structure would be very similar for all abdominal X-rays. Lastly, (d) is a representation with the lightest areas of the image highlighted only.


(a) Abdominal X-ray Image


(b) Lower level feature map of edges


(c) Feature map representing the bones

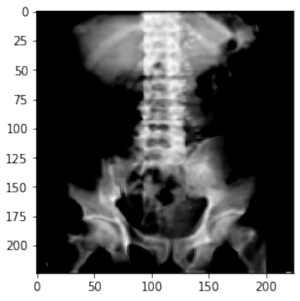

(d) Feature map of the lightest areas

Figure 5: Feature maps of image (a) from X-ray data

Each one of these images contains useful information about the visual structure of this data. Thus, it is important to convolve inputs with multiple filters as each feature map produces an individual representation of the data, preserving valuable spatial information from the data which is lost in traditional deep learning methods.

4 Methodology

Deep learning has a lot of potential for use in medical research and especially for medical image classification. One such use is a binary classification, where deep learning algorithms can be built to perform simple diagnostic classification such as positive or negative diagnosis. This can be performed on various forms of medical images including MRI, X-ray and CT scans. We will construct a binary classification model to allocate X-ray images into two defined classes; chest and abdominal X-ray. The model will take *'.png'* image inputs and will output a confidence score between 0 and 1, where a score close to 0 predicts an image is an X-ray of a chest and a score near 1 is an abdominal X-ray.

This section will detail the computational application of the mathematical techniques introduced in the background section. We will first outline the deep learning model framework which will be built to classify a medical image dataset of 75 X-ray images. Secondly, we will detail the methods used for the image preprocessing achieved by scaling and applying real-time data augmentation to diversify the dataset. Thirdly, we will discuss the suitability of PCA for reducing the dimensionality of this input data and also detail how a t-SNE model will be used provide lower dimensional visualisations of the data. Finally, we highlight how transfer learning will be used to evaluate the feasibility of using a pre-trained source model from a non-medical domain in a medical image classification model.

4.1 Constructing the Deep Learning Neural Network

4.1.1 Deep Learning Framework

Constructing a deep neural network from scratch requires an understanding of complex concepts on top of having computational power greater than most standard home computers and laptops can provide. In 2015, Google released an open source software library called *TensorFlow* which contains methods that can be combined to form the building blocks of many deep learning algorithms including multi-layered neural networks. TensorFlow provides an engine to perform the sequential and parallel operations used in deep learning algorithms. We access these methods via the *Keras* library which itself is a high-level Application Programming Interface (API) that makes the methods of TensorFlow more user friendly.

For TensorFlow to use images as data inputs it first stores the image data as a cube of numbers of shape width x height x depth, which is called a 3-dimensional (3D) *tensor*. TensorFlow then takes these tensor inputs and creates a *DataFlow* graph, which uses a network graph structure to represent the flow of individual and parallel operations in an algorithm. These networks consist of nodes which are units that perform the computations and edges which carry forward the information to the next layer. The DataFlow graphs can be constructed to produce a neural network with an input layer which accepts data, multiple hidden layers and an output layer which formulates a prediction on its input data. The edges are the *'weighted synapses'* that were discussed in section 3.6 where the required manipulations are performed with weight, bias and activation applied. The specifics of the DataFlow graph and network built for this analysis are detailed in section 4.1.2 and were carried out using Python written in a Jupyter Notebook. The source code is listed in the Appendix and the full notebook with printed results is available online at: `https://github.com/benkeel/benrepository`.

4.1.2 Building the (Sequential) Network Model

We will construct our neural network architecture using a variety of Keras' inbuilt modules that work on the backend of TensorFlow. Firstly, we initialise the model parameters $\Theta = (W, B)$ using a source model - the Inception V3 model, which has been widely used for image classification purposes. Outside of this research, its parameters have been pre-trained on *ImageNet* which is a large and diverse visual database of more than 15 million hand-annotated images with 22,000 different classes. Google's Inception V3 [13] is a 42-layered convolutional neural network which achieved 78% accuracy on a subset of the ImageNet data. Although Inception V3 was initially trained to classify of non-medical images, it will be shown that via transfer learning it can be repurposed for classifying images in a medical domain.

On top of the Inception V3 model we can define our own network architecture called the 'top model', optimised for the X-ray dataset. We will use a pre-defined Python object called a *'Sequential'* object, we can compile the model we will build with the Inception V3 model using the *.compile()* method. Keras allows us to add layers piecewise to the top model, using the *'.add()'* method. A multi-layered CNN will be constructed using a combination of convolutional and traditional dense layers as well as other techniques to prevent over-fitting. CNNs are highly suited to processing image data as they preserve the spatial structure of the data. A 2D convolutional layer accepts 4D tensor inputs and outputs representative feature maps of the data. This information is then processed by fully connected layers of nodes which are compiled in the final output layer to produce the final prediction (classification).

4.1.3 Convolutional layers

After defining the sequential model, we add 2D convolutional layers to extract this spatial information from the images. These layers contain multiple filters with a given kernel size per layer. This is the size of the matrix which is slid over the image to calculate a dot product with each placement of the filter. Kernel sizes of increasing magnitude in different layers can be used to allow the feature maps to represent different sized objects in the image. Example feature maps obtained from this data are shown in Figure 5. Feature maps can be used for instance to identify edges which could then be encoded into the bones. Then, this spatial information about the bone structure may be recognised across images in the domain which could play a role in the classification of an image. Convolutional layers add more trainable parameters which increases the degrees of freedom of the model. Multiple convolutional layers with a high number of trainable parameters will increase the computational work load of the model. So to help reduce this only two *'Conv2D'* layers will be used. For these layers we will choose 16 sliding filters each and small kernels of sizes of (3, 3) and (5, 5) respectively and ReLU activation.

4.1.4 Pooling layer

Next, we transform the shape of the network with a *'GlobalMaxPooling2D'* layer to reduce the dimensionality of the convolutional layer output from a 4D tensor of shape (batch size, map rows, map columns, number of filters) into a 2D tensor with shape (batch size, number of filters). Global maximum pooling typically performs better in image classification than other pooling methods. This operation 'down samples' each feature map to a single maximum value to summarise the presence of a feature in an image. Global pooling calculates only a single value which is different to normal pooling layers as they calculate a value at each patch of each feature map. A normal pooling layer will produce a 4D tensor with pooled rows and columns values replacing the map rows and columns. Further, max pooling only extracts the most important features of each map, taking a maximum value rather than taking an average across all features in each map. The resulting feature summaries may or may not be important for detecting objects, taking an average can mean losing the most important features. In summary, a global maximum pooling layer is added after the convolutional layers in order to down sample the feature maps into a 2D tensor, to be accepted by the fully connected dense layer.

4.1.5 Dense layers

Finally, we add fully connected layers of nodes which are defined as *'Dense'* layers, this type of layer is similar to the layers of neurons in a multi-layered perceptron. The layer is fed the 2D tensors produced by the pooling layer for each batch and outputs a value at each node to be fed forward to the next layer. Since our image data is of high dimensionality a large number of neurons are required to obtain a high level of accuracy. We add two dense layers each with 256 nodes (or 'neurons') each with ReLU activation. The final layer is another fully connected layer with 1 node only, and is used to calculate a weighted sum from the outputs of the 256 nodes in the penultimate layer. This singular value can be interpreted as a prediction on what class an image is. The final layer has sigmoid activation which normalises the value between 0 and 1 so can be seen as a confidence score that an image falls into a certain class. A score close to 0 classifies the image as a chest X-ray and a score close to 1 indicates an abdominal X-ray.

To help reduce the risk of over-fitting we add a dropout layer after each dense layer which randomly deactivates a percentage of nodes and their connections from the dense layer. In this model we will choose 25%, which means setting the output of 64 randomly selected nodes to 0 in each dense layer. Using a *'Dropout'* layer prevents the nodes from co-adapting too much, which means that the model is forced to generalise the data. To further reduce over-fitting we add *'BatchNormalization'* layers which normalises the output of the previous layer at each batch. This has the effect of stabilising the learning process, where the activations from the previous layer are standardised. It finds a transformation of the data so that it has a mean close to 0 and a standard deviation close to 1. This ensures that the subsequent layer can have the same assumption that the data is normally distributed. Additionally, batch normalisation helps to reduce the sensitivity of the parameters to their initial state. This is because the normalised values smooth the loss function which makes it easier to update the model parameters in back propagation. Thus, batch normalisation along with dropout layers help to improve the overall performance and speed of training in the model.

In summary, we have 2 dense layers of 256 nodes with ReLU activation. After each of these layers there is a dropout layer with 25% of nodes to be deactivated, and batch normalisation layers. The final layer has 1 node and sigmoid activation which can be interpreted to give a binary classification of this data. Once this model has been constructed it can be compiled and fed batches of training and testing data. These data are processed by the model and the predictions at each epoch are used to automatically tune the parameters using the back propagation process detailed in 7.3. The model is trained on 20 epochs of the 75 images.

4.1.6 Activation in each layer

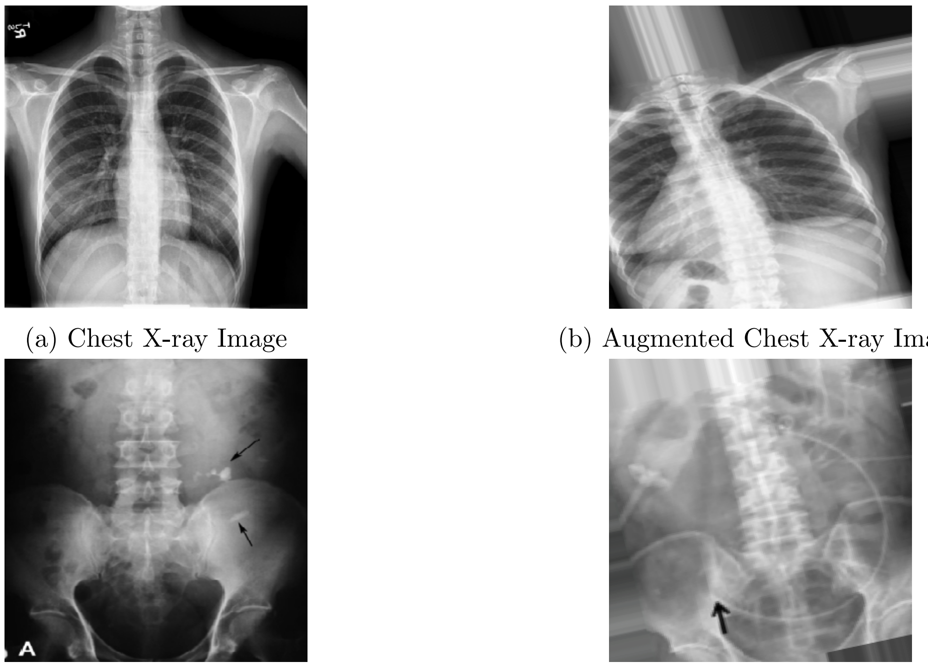
The convolutional and fully connected dense layers have ReLU activation applied which takes the maximum of the value $\alpha = W^T \cdot x + b$ produced from each node and 0.

$$f(\alpha) = max(0, \alpha)$$

As stated before, deep neural networks with hidden layer ReLU activation can train several times faster than sigmoid units and have little overall effect on model performance. This is because when $\alpha > 0$ the gradient of the activation is constant whereas the gradient of the sigmoid activation converges to 0 as the absolute value of α increases. The constant gradient of ReLU results in a faster learning rate η. The ReLU activation is also less computationally complex than sigmoid which reduces time it takes to complete the training phase. The output layer has a sigmoid activation since it normalises the prediction to a value between 0 and 1. This is required for binary classification as it can be used as a confidence score that the images fall into a certain class.

4.2 Data Preprocessing

The data we use is obtained from Paras Lakhani's GitHub repository, available online at `https://github.com/paras42/Hello_World_Deep_Learning`. It is a set of 75 X-ray images which we will partition into the training set X_{train} of 65 images with 33 chest X-ray images and 32 abdominal images and the testing set X_{test} of 10 images with 5 of each class. In their raw format, many of the images have different size dimensions which need to be regularised. We note that the dataset is comparatively small which is usually an issue in training deep learning models as they often require high volumes of data. The image data will be regularised and preprocessed using Keras' image preprocessing modules. Firstly, we can set a 'target size' of (299 x 299) pixels so that when when the data is generated from the images, the dimensions are uniform. Further, we can normalise the pixel values to between 0 and 1 and also apply random transformations, which is known as data augmentation. We apply data augmentation to increase the diversity of the dataset which can help a deep learning model to generalise the domain, in this case the domain is solely chest and abdominal X-rays. The Keras module $ImageDataGenerator()$ can be used to perform real-time data augmentations, by taking the input set and generating batches of tensor image data. The pixel RGB values are normalised to between 0 and 1 by dividing each value by 255, then a series of random transformations can be applied. These include; rotations, resizing, changing brightness, horizontal and vertical flips. This process replaces the training data with tweaked images, which are known as 'synthetic' data and a large amount of synthetic (augmented) images can be produced from the original set. Data augmentation is performed to help prevent the over-fitting of a model by exposing it to a greater amount of synthetic data which forces it to generalise the data (or its domain). In Figure 6 below, we have sample images taken from the dataset and example synthetic versions of these images.


(a) Chest X-ray Image (b) Augmented Chest X-ray Image

(c) Abdominal X-ray Image (d) Augmented Abdominal X-ray Image

Figure 6: Example of training images and augmentations from X-ray data

We can see that the synthetic chest (b) and abdominal (d) X-rays are similar but also different from images the model may encounter in real life data. Augmentation is applied so that the model does not just learn the training data specifically and so not perform well on unseen images. It increases the flexibility of the model so that it is able to correctly classify a broader range of images it might encounter, meaning that the model generalises the domain. This is equivalent to the polynomial mapping function of model having a good/robust fit (in Figure 1) where it is not ‘forced’ through all the known data points. Finally, it’s important to discuss potential problems with the data, for example some added features that are unique to this set and may play a significant role in classifying the images. We have that in each chest X-ray images there is an L or R printed on each image which are placed to indicate which side of the body the letter is on (see Figure 6 (a)). Further, in the abdominal X-ray images there are arrows which point out areas of interest identified in their analysis (see Figure 6 (c)). These arrows exist in all the abdominal X-rays and could be seen as another tell-tale sign of which type of image it is. Data augmentation may help to alleviate these concerns as it will make the patterns less consistent across the range of the data. For instance, in the image (b) the letter L (or R) has been cropped out of the image and so would not influence its classification. In summary, this dataset does present some issues as the size of the data used will not be sufficient to generalise the data and that there is a potential for over-fitting of the model. However, the use of data augmentation as well as other dropout and batch regularisation layers in the model are included to reduce over-fitting.

4.3 Dimensionality Reduction

The full dataset of the X-ray images produces a tensor of the shape (75, 299, 299, 3) which holds the 75 images each with 299 x 299 pixels and 3 colour channels. This equates to 20,115,255 values that the model has to process at each iteration of the full dataset (epoch). To train a deep learning model on this data we parse the entire set through the deep learning algorithm a number of times and error adjust the model parameters using back propagation. As a result, the training phase of the X-ray model is relatively complex and takes a considerable amount of time to run on a low powered machine. This leads to the question of how to reduce the computational complexity of the model without jeopardising the accuracy of predictions.

4.3.1 Application of PCA

Dimensionality reduction techniques can be applied to high dimensional datasets such as this, to reduce the overall workload of the algorithm but still produce accurate classification predictions. PCA is a technique which achieves this by finding a new set of independent features which describe the most important patterns and structure in the data. Through the procedure stated in 3.4.1, this is achieved by finding the eigendecomposition of the data space and retaining k eigenvectors of largest eigenvalue. A new k-dimensional space is then constructed using the chosen orthogonal eigenvectors as its basis. The geometric projections of the data onto this space retain a percentage of the variance between the observations in high dimensional space, this calculated as a proportion of the sum of all eigenvalues. Eigenvectors with low eigenvalue can be seen to hold insignificant or redundant information about the data and so can be disregarded without a significant loss of information.

To carry out this operation we use the PCA module from Python’s machine learning library *Scikit-learn*. It performs this dimensionality reduction technique cleanly and allows you to specify the amount of variance you wish to retain. Firstly, to use this module the data is converted from generator form to *numpy* arrays using the *‘.next()’* method to concatenate data from all the 75 images into a single tensor. Then, it was found that to explain 90% of the variance it required 23 Principal Components (PCs) which are the eigenvectors of largest eigenvalue. The cumulative variance represented by the chosen components is visualised below in Figure 7.

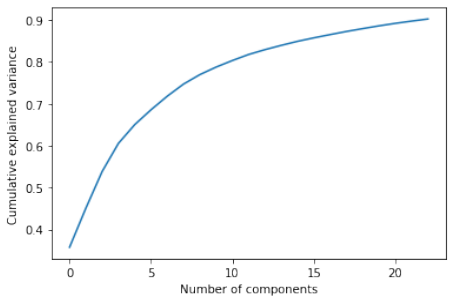


Figure 7: Cumulative explained variance from 23 PCs

This linear orthogonal transformation converts our combined image dataset of shape (75, 299, 299, 3) into the shape (75, 23) which means that the model only processes 1725 values at each epoch. As a result, there is a 99.99% reduction in the number of values the model needs to process, significantly reducing its complexity. This data can be used to train a deep learning algorithm for the same purpose of classifying X-ray images. A Keras sequential model is constructed with similar architecture to the model in 4.5 and its composition is detailed in the Appendix. Further, the reduced complexity of the data allows us to make the neural network 'deeper' by adding more hidden layers which can improve the performance. However, since the transformation involves flattening the 4D tensor into a 2D tensor (matrix), we are unable to convolve inputs with convolutional layers in the model and spatial information is lost.

The training process of the PCA model was completed in a fraction of the time it took to train the full model on the high dimensional version of this data. The model achieved 98.7% accuracy on this data which higher accuracy than the accuracy that high-dimensional model achieved. This suggests that this classification task can be simplified using dimensionality reduction.

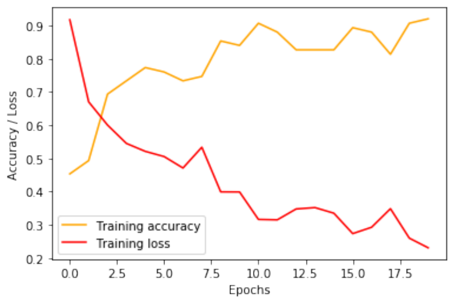


Figure 8: PCA Accuracy Curve

The performance graph in Figure 8 indicates that the accuracy increases linearly and the loss decreases at an identical rate, which is expected. The speed of training and accuracy suggests that the model can easily learn this data. It could be the case that this model is over-fitted due to simplicity of the data, where the mapping function is recognising each image individually leading to false performance readings. This model requires all the data to be combined into one set to perform PCA which means that the model is not subjected to any testing data. This makes it difficult to determine whether the model has just simply learned the training data very well and therefore will not perform well on new data. PCA is mainly suitable for exploratory data analysis, where it is used to find structure in databases that contain images with no class labels. Since this data is already classified, this analysis can be primarily used as a measure of the complexity of this task. We have demonstrated that this classification task could be carried out from a lower dimension and that the internal structure of the data supports an accurate binary separation.

4.3.2 Application of t-SNE

As a further exploration of the data, t-SNE was applied as a visualisation technique for the high dimensional X-ray data into a 2-dimensional scatter plot which is shown below in Figure 9.

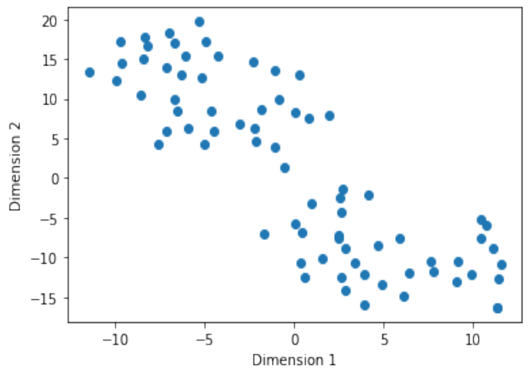


Figure 9: t-SNE Visualisation

This method, as described in 3.4.2 calculates the conditional probability that one point would choose another as its neighbour in high and low dimensional space. Then the difference between these two measures of similarity is calculated and minimised using the sum of KL divergence of these two probability distributions. The result is a more accurate lower dimensional projection of high dimensional data. In Figure 9 we can observe clear separation of the data into two clusters. In the scatter plot, images from one class take on mostly positive values in one dimension and negative values in the other which indicates that the data is separated this way in its origin high dimension. This supports the use of a binary classification model as the plot confirms that the data is truly binarily separated.

4.4 Transfer Learning Application

Keras allows to you to apply transfer learning directly by importing the model architecture (Figure 10) and parameters Θ of the Inception V3 model as the source model.

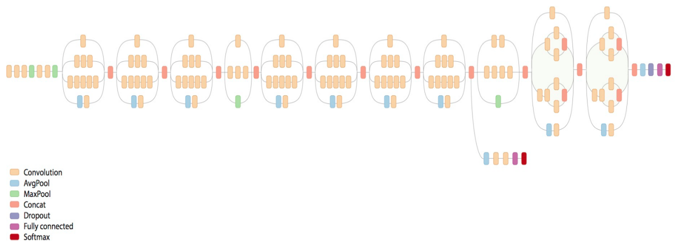
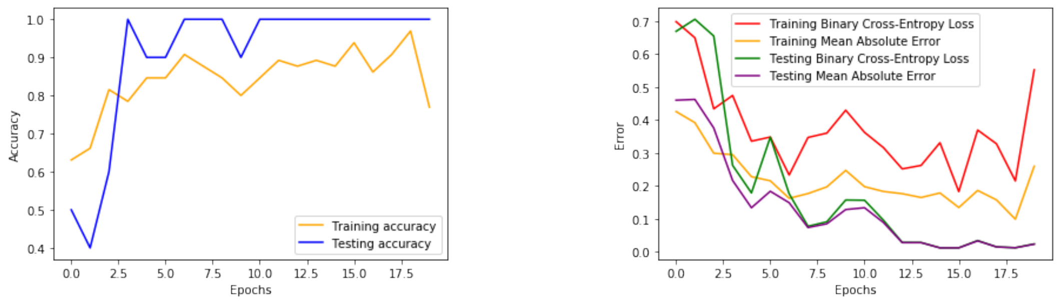


Figure 10: Architecture of Inception V3 from [14]

This source model is compiled with the model we defined in section 4.1 and together they form final neural network architecture. Transfer learning utilises the knowledge gained in the source model in the form of parameters and learned feature maps, which can be transferred to a new model. Transfer learning can improve a deep learning algorithm's performance in a number of ways. Firstly, the new model will have higher initial skill as the source model will already be adept to recognising generalised features. Furthermore, it can increase the rate at which the model learns the data which increases the gradient of the training accuracy curve. Lastly, transfer learning can also increase the limit in accuracy it converges to, giving the new model a higher overall accuracy after training. This is because some of the learned features in Inception V3 may be applicable to classifying X-rays. Multiple research papers cite the use of transfer learning of models with non-medical domains for use in medical image classification. In particular, Chang and Park [14] presented the feasibility of the Inception V3 model for the classification of histopathological images of breast cancer and achieved over 83% accuracy. In this report we will apply the transfer learning of the Inception V3 model to improve the performance of the chest and abdominal X-ray classification model. Additionally, we will explore further applications of this model by repurposing the Inception V3 model again to be able to give an informal diagnosis from the X-rays of potential COVID-19 patients.

5 Model Evaluation

At the end of the training phase the X-ray model achieved 90.8% accuracy which is equivalent to correctly predicting 64/65 on the training set. In Figure 11 (a) below we can see that the accuracy improved for both training and testing sets as the number of epochs increased. The training data reached a limit in accuracy of around 90% after the 10 epochs and the testing accuracy reached 100% consistently after the first 5 epochs. The accuracy represents the learning of the model and the number of epochs is the number of passes of the X-ray data; which is the experience of the model. Thus, we can determine as the model gained more experience its accuracy improved.



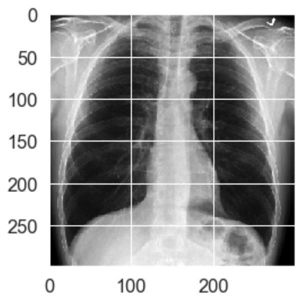
(a) Accuracy Curve (b) Error Curve

Figure 11: X-ray Performance Measures

In (b) two different loss functions are used to evaluate the performance, a binary cross-entropy (BCE) and mean squared error (MSE). As detailed in the optimisation algorithm section 3.6.4, the BCE is preferred for binary classification as the loss function is convex over the range [0, 1]. Thus, the back propagation of the BCE optimises the weights towards a clear minimum. The error curves show a negative trend for both sets, the testing errors follow a very similar trend whereas these two measures vary more for the training data. Also in Figure 11 (b) the cross-entropy loss for the training data is more 'noisy' and is higher than its mean squared error. In the error curve above both the testing losses were below 0.1 at the end of the training phase whereas the training losses were consistently higher. The BCE curve for the training data had a mean average value of 0.374 which is higher than the average testing BCE of 0.187.

The results from this model suggest that this is not an optimal fit for this data because the loss curves would be expected to show minimal differences in trend and value between the training and testing data. But both of the training loss functions have higher values than the testing loss with a difference of at least 0.1 past the 5th epoch. This may be a consequence of a small dataset that does not fully represent the probability distribution of the data. Additionally, there is a large number of parameters with 22,141,121 trainable parameters from the combined model with the Inception V3 model base. This large number of degrees of freedom gives the model more flexibility but can also mean that groups of nodes can co-adapt which can lead to over-fitting. It may also be the case that the classification problem is too simple for the model leading to high levels of accuracy with over 90% for both sets. Therefore we may be able to deduce more with a larger training sample size. The training accuracy is lower which may be a consequence of data augmentation in the preprocessing of the image data where the training data had already had several transformations applied to it such as random flips and rescaling. As a result, it may have influenced the differences in accuracy between the two sets as the testing set did not undergo any augmentation.

Next, to see if the data has only learned the similar images from the small train and test sets we use the model to predict the class of images obtained externally from [15]. We do this to see if the learned features can classify a different style of X-rays from its experience and thus 'transfer' its learning to another purpose. In Figure 12 we have the images taken from the external source and the predictions produced by this model. The model gives correct predictions on both these images, but it is more confident about (a) with a score of 0.0194 than (b) with a score of 0.342. In (b) we have a side view of a chest X-ray which the model was able to correctly classify, even though this type is different to the front facing view on all the data in the training set. Clearly this model has generalised the domain well and is capable of performing accurately on unseen data.


(a) Predicted: [[0.01937448] Chest X-ray

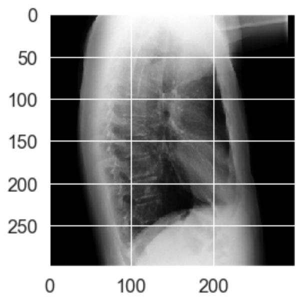

(b) Predicted: [0.34242734] Chest X-ray

Figure 12: Chest X-ray images

Lastly, we should note that without the use of the Inception V3 source model, the network we constructed on top is not very effective at classifying inputs. To test this, the top model alone (without compiling with the Inception V3 model) was trained on the same data. It achieved 50% accuracy, which is equivalent to blindly guessing the class of the images, thus we should note that the success of this model depends greatly on the source model for its learned parameters for general image classification. It is important however to also build this model on top as only the top model parameters are changed in the training process. Therefore transfer learning hinges on both the source and the top models in combination to achieve the optimal results. This model is proof of principal that transfer learning can be somewhat effectively utilised to build accurate classification models on small datasets.

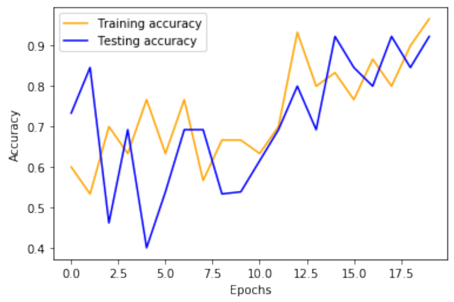
5.1 Transferring the learning of the model to COVID-19

To further explore the applications of binary classification models, I will train a model for the purpose of diagnosing COVID-19. In late 2019 a virus named SARS-CoV-2 or Coronavirus which causes the disease COVID-19, has gripped the world in one of the the deadliest pandemic in recent times. Deep learning methods to help combat this disease have recently begun appearing in academic papers. To explore the effectiveness of deep learning methods in this context, I will build a binary classification model to attempt to diagnose COVID-19 from chest X-ray images. The new model will make use of the same algorithm I built to classify between chest and abdominal X-rays (detailed in section 4) but will be trained on a new dataset containing

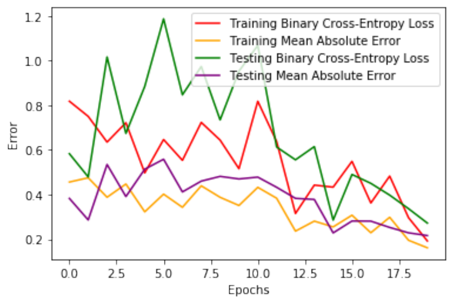
X-ray images of known COVID-19 cases. This is done to demonstrate the feasibility of using transfer learning of wider image domains for specific medical image analysis.

This new model is built for an accurate but informal diagnosis of COVID-19 as it would still need to be laboratory tested to be sure. It will be trained on a new dataset containing a mix of COVID-19 confirmed cases by pathological laboratory testing and negative cases. These images were obtained from Joseph Paul Cohen's GitHub repository at `https://github.com/ieee8023/covid-chestxray-dataset` and includes 50 chest X-ray images. The dataset is equally split with 25 patients with COVID-19 positive and 25 negative cases. The negative cases include images of completely healthy patients with no underlying conditions as well as those with bacteria or viral based pneumonia and other repository conditions unrelated to COVID-19. The class of negative cases is constructed to include suspected cases with similar symptoms as well as completely healthy patients. This is done to be representative of those who might be tested for the condition. Once trained, this model can give a predicted diagnosis of the condition from a chest X-ray almost instantly.

To evaluate the performance of this model from a deep learning perspective, we reference the results from the model in Figure 13. It's clear that the model has overall performed well on this classification task, achieving 96.7% accuracy at the end of the training phase. The accuracy curve (a) shows that both training and testing accuracy followed a similar trend and both had accuracy of over 90%. Also the error curve (b) shows that the error was suitably reduced to around 0.3 during the training phase of this model. The mean average values of BCE for training data was 0.552 and the testing was higher with 0.671. Thus, the model here could be expected to perform quite accurately if put into use. However, as with the chest and abdominal X-ray model there is consistently higher training loss throughout the training phase which indicates there may be over-fitting. Additionally, the overall accuracy of the model is less than the previous model however this is expected as the learning task is more complex. This is because all the images are of the same type (chest X-rays) and the model is looking for inflammation in the lungs among other tell-tale features it might recognise to be a result of COVID-19.



(a) Accuracy Curve



(b) Error Curve

Figure 13: COVID-19 Model Performance Measures

The applications of the models built in this analysis and others similar extend widely to help reduce the impact of various disease on heavily burdened healthcare systems. For instance, a deep learning algorithm trained on COVID-19 data could be used to prioritise highly suspect cases where the model believes its highly likely that the person has the condition. Laboratory testing can take over 48 hours to complete which is a significant time lag in the system. It presents a risk where people with the condition may still be interacting with others which could lead to more cases. This model can provide instant results in situations where it may be beneficial to find out sooner. Further, this model could be used to highlight cases where there is potential for misclassification. Where the virus may not show up in laboratory testing but is deemed highly likely to be present based on an X-ray.

6 Conclusion

In conclusion, we presented an in-depth analysis of deep learning methods for use in medical image classification. We found that the applications of deep learning can extend to various realms of medical image analysis and classification. In this report we explored the mathematical theory behind how deep learning algorithms can learn from labelled data by approximating a mapping function to predict the labels of unseen data. The theory discussed includes the process and applications of dimensionality reduction and clustering; for visualisation and to reduce the complexity of datasets. Additionally, the training and optimisation of Artificial Neural Networks and more specifically Convolutional Neural Networks for use in image classification. Finally, we provided a proof of principal of using the transfer learning of a model with a non-medical domain in application to a medical domain. This was carried out in the framework of the TensorFlow library and the Keras API with the transfer learning of Google's Inception V3 model. A high-performance binary classification model was constructed to classify chest and abdominal X-rays. Further, this model was repurposed to diagnose COVID-19 from chest X-ray images with comparable accuracy. In summary, we have demonstrated the theory and practical application of deep learning methods in a medical domain, showing that they can offer high flexibility in the scope of analysis which a model is intended for.

References

[1] A. Wade, "*How AI is powering a revolution in medical diagnostics.*" Available from: https://www.theengineer.co.uk/ai-medical-diagnostics/, 2018. [Accessed: 24/3/2020].

[2] A. Bhande, "*What is underfitting and overfitting in machine learning and how to deal with it.*" Available from: https://medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76, 2018. [Accessed: 11/12/2019].

[3] Y. P. Lin and T. P. Jung, "*Improving EEG-based emotion classification using conditional transfer learning,*" *Frontiers in Human Neuroscience*, vol. 11, p. 3, 2015. [Accessed: 8/3/2020].

[4] N. Kambhatla and T. K. Leen, "*Dimension Reduction by Local Principal Component Analysis,*" *Neural Computation*, vol. 9, no. 7, pp. 1493–1494, 1997. [Accessed: 18/12/2019].

[5] A. Dubey, "*The Mathematics Behind Principal Component Analysis.*" Available from: https://towardsdatascience.com/the-mathematics-behind-principal-component-analysis-fff2d7f4b643, 2018. [Accessed: 18/12/2019].

[6] L. van der Maaten and G. Hinton, "*Visualizing Data using t-SNE,*" *Journal of Machine Learning Research*, vol. 9, pp. 2579–2906, 2008. [Accessed: 18/12/2019].

[7] J. Jaju, "*Comprehensive Guide on t-SNE algorithm with implementation in R and Python.*" Available from: https://www.analyticsvidhya.com/blog/2017/01/t-sne-implementation-r-python/, 2017. [Accessed: 18/2/2020].

[8] N. Altman and M. Krzywinski, "*Clustering,*" *Nature Methods*, vol. 14, pp. 545–546, 2017. [Accessed: 19/12/2019].

[9] S. Kaushik, "*An Introduction to Clustering and different methods of clustering.*" Available from: https://www.analyticsvidhya.com/blog/2016/11/an-introduction-to-clustering-and-different-methods-of-clustering/, 2016. [Accessed: 19/12/2019].

[10] A. Trevino, "*Introduction to K-means Clustering.*" Available from: https://blogs.oracle.com/datascience/introduction-to-k-means-clustering, 2016. [Accessed: 20/12/2019].

[11] B. E. B. e. a. Geert Litjens, Thijs Kooi, "*A survey on deep learning in medical image analysis,*" *Medical Image Analysis*, vol. 42, p. 62, 2012. [Accessed: 1/2/2020].

[12] K. S. Zhou, H. Greenspan, and D. Shen, "*Deep Learning for Medical Image Analysis,*" *Deep Learning for Medical Image Analysis*, p. 5, 2017. [Accessed: 1/2/2020].

[13] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "*Rethinking the Inception Architecture for Computer Vision,*" 2015. [Accessed: 7/3/2020].

[14] J. Chang and E. Park, "*A Method for Classifying Medical Images using Transfer Learning: A Pilot Study on Histopathology of Breast Cancer,*" 2017. [Accessed: 10/3/2020].

[15] F. Gaillard, "*Normal chest x-ray.*" Available from: https://radiopaedia.org/cases/normal-chest-x-ray. [Accessed: 11/3/2020].

Appendices

```
#As stated before, the Jupyter Notebook which details the procedure of the analyses
    carried out in this report is available online at
    \url{https://github.com/benkeel/benrepository}

#Here I have imported the necessary modules to generate the image data and apply
    real-time augmentation
#Code referenced in Methodology: Keras Sequential Model 4.1, Preprocessing 4.2 and
    #Dimensionality Reduction 4.3
from keras import backend
import keras as K
from keras import applications
from keras.preprocessing.image import ImageDataGenerator
from keras import optimizers, layers
from keras.preprocessing import image
from keras.models import Sequential
from keras.layers import Input, Dropout, Flatten, Dense, BatchNormalization,
    GlobalMaxPooling2D
from keras.models import Model
from keras.optimizers import Adam
from keras.layers.convolutional import Conv2D
from keras.regularizers import l1, l2
from keras import metrics
from keras.utils import plot_model
import pydot as pyd
from keras.applications.inception_v3 import InceptionV3
import numpy as np
from numpy import *
import matplotlib.pyplot as plt
import pandas as pd
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import array_to_img
from numpy import concatenate
from PIL import Image

#Dimensions of our images
img_width, img_height = 299, 299

#Location of training data
train_data_dir = r"C:\Users\benke\Documents\Year 3\Deep Learning\Xray
    classification\Open_I_abd_vs_CXRs\TRAIN"

#Location of validation (testing) data
validation_data_dir = r"C:\Users\benke\Documents\Year 3\Deep Learning\Xray
    classification\Open_I_abd_vs_CXRs\VAL"

#Number of samples used for determining the samples per epoch
nb_train_samples = 65
nb_validation_samples = 10
epochs = 20
batch_size = 5

# Using ImageDataGenerator() to rescale images and apply real time augmentation
# Training data with augmentation
train_datagen = ImageDataGenerator(
       rescale=1./255,         # normalize pixel values to [0,1]
       shear_range=0.2,
       zoom_range=0.2,
       rotation_range=20,
       width_shift_range=0.2,
       height_shift_range=0.2,
       horizontal_flip=True)

# Validation Data without augmentation
val_datagen = ImageDataGenerator(
       rescale=1./255)     # normalize pixel values to [0,1]

# Producing batches of image data to be accepted by Keras Seqential Model using
    'model.fit_generator()'
# Training generator
train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=(299,299),
    batch_size=5,
    class_mode='binary')

# Validation generator
validation_generator = val_datagen.flow_from_directory(
    validation_data_dir,
    target_size=(299,299),
    batch_size=5,
    class_mode='binary')

# Training data without augmentation (for use in PCA)
train_datagen2 = ImageDataGenerator(
       rescale=1./255)    # normalize pixel values to [0,1]

train_generator2 = train_datagen2.flow_from_directory(
    train_data_dir,
    target_size=(299,299),
    batch_size=5,
    class_mode='binary')

#PCA: see Section 4.3.1
# Convert non-augmented training and validation data into numpy arrays for use in PCA
# Data is converted from train_generator using '.next()' method to iterate through
    batches of 5

# Produced in batches of 5 so range(13) produces 5x13 = 65 training images
train_images = np.concatenate([train_generator2.next()[0] for i in range(13)])
train_labels2 = np.concatenate([train_generator2.next()[1] for i in range(13)])
train_labels = np.reshape(train_labels2,(65,1))

# range(2) produces 5x2 = 10 validation images
val_images = np.concatenate([validation_generator.next()[0] for i in range(2)])
val_labels2 = np.concatenate([validation_generator.next()[1] for i in range(2)])
val_labels = np.reshape(val_labels2,(10,1))

# Check images match up to label: not matching up correctly
# However since the labels of the data are produced from a generator it is combined
    correctly in sequential model
print(display(array_to_img(train_images[24])))
print(train_labels[24])
```

```
# Check shape of data
print(train_images.shape)
# (65, 299, 299, 3)
print(val_images.shape)
# (10, 299, 299, 3)

# Flatten the images in 2D tensor (matrix) using (65,299x299x3) = (65,268203)
train_images = np.reshape(train_images,(65,268203))
val_images = np.reshape(val_images,(10,268203))

# Check shape is correct
print(train_images.shape)
# (65, 268203)
print(val_images.shape)
# (10, 268203)

# Concatenate (combine) images and labels from both training and validation data
# This is because with less that 75 images we need 23 PCs for 90\% variance
# and PCA cannot produce 23 PCs on 10 validation images so we produce 23 PCs on 75
    combined images
images = np.concatenate((train_images,val_images))
labels = np.concatenate((train_labels2,val_labels2))

print(images.shape)
# (75, 268203)
print(labels.shape)
# (75,)

# Conducting PCA on numpy arrays of X-ray data
# Used 'n_components=0.9' to capture 90\% of data variance which produces 23
    Principal Components (eigenvectors)
# from combined dataset of 75 images
time_start = time.time()
pca = PCA(copy=True, iterated_power='auto', n_components=0.9, random_state=None,
    svd_solver='auto', tol=0.0, whiten=True)
pca.fit(images)
pca_images = pca.transform(images)

# Produce graph of cumulative explained variance from 23 PCs (Figure 7)
# and print eigenvalues as 'Variance explained per principal component'
print('PCA done! Time elapsed: {} seconds'.format(time.time()-time_start))
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('Number of components')
plt.ylabel('Cumulative explained variance')
print('Variance explained per principal component:
    {}'.format(pca.explained_variance_ratio_))

# Check tensor shape
print(pca_images.shape)
# (75,23)
#Conduct PCA on numpy arrays of X-ray data: see section 4.3
from sklearn.decomposition import PCA
from PIL import Image
import os
import time

#PCA done! Time elapsed: 3.478677272796631 seconds
```

```
#Variance explained per principal component:
#[0.35792896 0.09307032 0.08756059 0.06722919 0.04470531 0.03544107
#0.03251029 0.02894042 0.02251312 0.01800812 0.0156776 0.01442149
#0.01145873 0.01030645 0.00957115 0.00832101 0.00770599 0.0075008
#0.00680107 0.00658203 0.00599622 0.00534516 0.00500807]

#Sequential Model on PCA data
modelPCA = Sequential()
#hidden layer 1
modelPCA.add(Dense(1024, activation='relu', input_shape=(23,)))
modelPCA.add(Dropout(0.25))
modelPCA.add(BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001))
#hidden layer 2
modelPCA.add(Dense(1024, activation='relu'))
modelPCA.add(Dropout(0.25))
modelPCA.add(BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001))
#hidden layer 3
modelPCA.add(Dense(512, activation='relu'))
#hidden layer 4
modelPCA.add(Dense(256, activation='relu'))
#output layer
modelPCA.add(Dense(1, activation='sigmoid')) #used 'sigmoid' for the output layer as
    this is a binary classification.

#compile using Adam optimiser
modelPCA.compile(optimizer=Adam(lr=0.0001, beta_1=0.9, beta_2=0.999,
    epsilon=1e-08,decay=0.0), loss='binary_crossentropy', metrics=['accuracy',
    metrics.mae])

# Train PCA model (no validation used here as train and validation sets are combined)
historyPCA = modelPCA.fit(pca_images,
    labels,batch_size=batch_size,epochs=epochs,verbose=1,
                  validation_data=None)

# Produce PCA performance graph (Figure 8)
plt.figure()
plt.plot(historyPCA.history['acc'], 'orange', label='Training accuracy')
plt.plot(historyPCA.history['loss'], 'red', label='Training loss')
plt.legend()
plt.show()

# Produce accuracy measures
loss = modelPCA.evaluate(pca_images, labels)
print(modelPCA.metrics_names)
print(loss)
#['loss', 'acc', 'mean_absolute_error']
[0.1222726062933604, 0.9866666706403097, 0.10687879502773284]
print("Accuracy = ", loss[1])
#Accuracy = 0.9866666706403097

#TSNE: see section 4.3.2
# Import TSNE from sckit learn
from sklearn.manifold import TSNE
import time
time_start = time.time()

# Figure 9
tsne_images = TSNE(perplexity=15,learning_rate=1).fit_transform(images)
```

```
print('t-SNE done! Time elapsed: {} seconds'.format(time.time()-time_start))
tsne_images.shape
x = [tsne[0] for tsne in tsne_images]
y = [tsne[1] for tsne in tsne_images]
plt.scatter(x,y)
# Graph produced shows clear separation of data into clusters
# different 'perplexity' values produce different plots (some with less clear
    separation)

#Dendrogram produced of X-ray data, shown in Figure 2
dendrogram = sch.dendrogram(sch.linkage(images, method='ward'))
# Import PCA module from scikit learn and other used modules
from sklearn.decomposition import PCA
import os
import time

#X-ray Sequential Model: see section 4.1
source_model = applications.InceptionV3(weights='imagenet', include_top=False,
    input_shape=(img_width, img_height, 3))

print(source_model.input_shape[0:])
#(None, 299, 299, 3)
print(source_model.output_shape[0:])
#(None, 8, 8, 2048)

#Define sequential model
model_top = Sequential()
#hidden convolutional layer 1
model_top.add(Conv2D(16, (3, 3), input_shape=(8,8,2048), activation='relu',
    padding='valid'))
#hidden convolutional layer 2
model_top.add(Conv2D(16, (5, 5), activation='relu', padding='valid'))
model_top.add(GlobalMaxPooling2D(input_shape=(299, 299, 3), data_format=None))
#hidden dense layer 3
model_top.add(Dense(256, activation='relu'))
model_top.add(Dropout(0.25))
model_top.add(BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001))
#hidden dense layer 4
model_top.add(Dense(256, activation='relu'))
model_top.add(Dropout(0.25))
model_top.add(BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001))
#output layer
model_top.add(Dense(1, activation='sigmoid')) #used 'sigmoid' for the output layer as
    this is a binary classification.

#Combine with Inception V3 (source) model
model = Model(inputs=source_model.input, outputs=model_top(source_model.output))

#Compile with Adam optimiser
model.compile(optimizer=Adam(lr=0.0001, beta_1=0.9, beta_2=0.999,
    epsilon=1e-08,decay=0.0), loss='binary_crossentropy', metrics=['accuracy',
    metrics.mae])

#Train model on train and validation generators
history = model.fit_generator(train_generator, steps_per_epoch=nb_train_samples //
    batch_size, epochs=20, validation_data=validation_generator,
    validation_steps=nb_validation_samples // batch_size)
```

```
#Figure 11 (a)
print(history.history.keys())
plt.figure()
plt.plot(history.history['acc'], 'orange', label='Training accuracy')
plt.plot(history.history['val_acc'], 'blue', label='Testing accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

#Figure 11 (b)
plt.figure()
plt.plot(history.history['loss'], 'red', label='Training Binary Cross-Entropy Loss')
plt.plot(history.history['mean_absolute_error'], 'orange', label='Training Mean
    Absolute Error')
plt.plot(history.history['val_loss'], 'green', label='Testing Binary Cross-Entropy
    Loss')
plt.plot(history.history['val_mean_absolute_error'], 'purple', label='Testing Mean
    Absolute Error')
plt.xlabel('Epochs')
plt.ylabel('Error')
plt.legend()
plt.show()

# Mean average values of BCE for training and testing respectively
print(sum(history.history['loss'])/20)
#0.3739572982088878
print(sum(history.history['val_loss'])/20)
#0.1865787618793547

#Figure 12 predictions
# Image locations
img_path= r"C:\Users\benke\Documents\Year 3\Deep Learning\Xray
    classification\other\76052f7902246ff862f52f5d3cd9cd_jumbo.jpg"
img_path2= r"C:\Users\benke\Documents\Year 3\Deep Learning\Xray
    classification\other\da9ebbb115dc5a6bc77a65541789eb_jumbo.jpg"

# Load images
img = image.load_img(img_path, target_size=(img_width, img_height))
img2 = image.load_img(img_path2, target_size=(img_width, img_height))

# Show image which is being classified: Chest X-ray
plt.imshow(img)
plt.show()

# Convert to array
img = image.img_to_array(img)
x = np.expand_dims(img, axis=0) * 1./255 # normalize pixel values to [0,1]
# Use model to classify image
score = model.predict(x)
# Round scores to give qualitative classification of image
print('Predicted:', score, 'Chest X-ray' if score < 0.5 else 'Abd X-ray')

# Side view Chest X-ray
plt.imshow(img2)
plt.show()

img2 = image.img_to_array(img2)
```

```
x = np.expand_dims(img2, axis=0) * 1./255
score2 = model.predict(x)
print('Predicted:', score2, 'Chest X-ray' if score2 < 0.5 else 'Abd X-ray')

# Produce accuracy measures
loss = model.evaluate_generator(train_generator, len(train_generator), workers = 1)
print(model.metrics_names)
print(loss)
print("Accuracy = ", loss[1])

# Sample Augmented images shown in Preprocessing (Section 4.2: Figure 6)
# Producing sample augmented images using '.next()' method and displaying transformed
    image
augmented_images = np.concatenate([train_generator.next()[0] for i in range(13)])
print(display(array_to_img(augmented_images[24])))

# Feature maps shown in Convolutional Neural Network (Section 3.7.4: Figure 5)
# Producing feature maps from first convolutional layer for given image
# Import VGG16 model and produce feature maps
from keras.applications.vgg16 import VGG16
from keras.applications.vgg16 import preprocess_input
from matplotlib import pyplot
from numpy import expand_dims
# load the model
model = VGG16()
# redefine model to output right after the first hidden layer
model = Model(inputs=model.inputs, outputs=model.layers[1].output)
model.summary()
# load the image with the required shape [(224,224) for VGG16 model]
img = load_img(r"C:\Users\benke\Documents\Year 3\Deep Learning\Xray
    classification\Open_I_abd_vs_CXRs\TEST\abd2.png", target_size=(224, 224))
# convert the image to an array
img = img_to_array(img)
print(img)
# expand dimensions so that it represents a single 'sample'
img = expand_dims(img, axis=0)
# prepare the image (e.g. scale pixel values for the vgg)
img = preprocess_input(img)
# get feature map for first hidden layer
feature_maps = model.predict(img)
# plot all 64 maps in an 8x8 squares
square = 8
ix = 1
for _ in range(square):
   for _ in range(square):
      # specify subplot and turn of axis
      ax = pyplot.subplot(square, square, ix)
      ax.set_xticks([])
      ax.set_yticks([])
      # plot filter channel in grayscale
      pyplot.imshow(feature_maps[0, :, :, ix-1], cmap='gray')
      ix += 1
# show the figure
pyplot.show()

# show individual feature maps
pyplot.imshow(feature_maps[0, :, :, 15], cmap='gray')
```

```
# COVID-19 Model: see section 5.2
# Dimensions of our images.
img_width, img_height = 299, 299

# Location of training data
train_data_dirCOVID = r"C:\Users\benke\Documents\Year 3\Deep
    Learning\COVID-19\dataset\Train"

# Location of testing data
test_data_dirCOVID = r"C:\Users\benke\Documents\Year 3\Deep
    Learning\COVID-19\dataset\Test"


# Number of samples used for determining the samples per epoch
nb_train_samplesCOVID = 30
nb_test_samplesCOVID = 18
epochs = 20
batch_size = 5

# Using ImageDataGenerator() to rescale images and apply real time augmentation
# Training data with augmentation
train_datagen = ImageDataGenerator(
       rescale=1./255,         # normalize pixel values to [0,1]
       shear_range=0.2,
       zoom_range=0.2,
       rotation_range=20,
       width_shift_range=0.2,
       height_shift_range=0.2,
       horizontal_flip=True)

# Validation Data without augmentation
test_datagen = ImageDataGenerator(
       rescale=1./255)     # normalize pixel values to [0,1]

# Producing batches of image data to be accepted by Keras Seqential Model using
    'model.fit_generator()'
# Training generator
train_generatorCOVID = train_datagen.flow_from_directory(
    train_data_dirCOVID,
    target_size=(299,299),
    batch_size=5,
    class_mode='binary')

# Validation generator
test_generatorCOVID = test_datagen.flow_from_directory(
    test_data_dirCOVID,
    target_size=(299,299),
    batch_size=5,
    class_mode='binary')

source_model = applications.InceptionV3(weights='imagenet', include_top=False,
    input_shape=(299, 299, 3))

# Define sequential model
model_top = Sequential()
# hidden convolutional layer 1
```

```
model_top.add(Conv2D(16, (3, 3), input_shape=(8,8,2048), activation='relu',
    padding='valid'))
# hidden convolutional layer 2
model_top.add(Conv2D(16, (5, 5), activation='relu', padding='valid'))
model_top.add(GlobalMaxPooling2D(input_shape=source_model.output_shape[1:],
    data_format=None))
#model_top.add(GlobalMaxPooling2D(input_shape=(299, 299, 3), data_format=None))
# hidden layer 3
model_top.add(Dense(256, activation='relu'))
model_top.add(Dropout(0.25))
model_top.add(BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001))
# hidden layer 4
model_top.add(Dense(256, activation='relu'))
model_top.add(Dropout(0.25))
model_top.add(BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001))
# output layer
model_top.add(Dense(1, activation='sigmoid')) #used 'sigmoid' for the output layer as
    this is a binary classification.

# Produce summary of model to be trained on X-ray data
model_top.summary()

# Combine with Inception V3 (source) model
modelCOVID = Model(inputs=source_model.input, outputs=model_top(source_model.output))

# Produce summary of full model
modelCOVID.summary()

# Compile using Adam optimisation algorithm
modelCOVID.compile(optimizer=Adam(lr=0.0001, beta_1=0.9, beta_2=0.999,
    epsilon=1e-08,decay=0.0), loss='binary_crossentropy', metrics=['accuracy',
    metrics.mae])

# Train model on train and validation generators
historyCOVID = modelCOVID.fit_generator(
           train_generatorCOVID,
           steps_per_epoch=nb_train_samplesCOVID // batch_size,
           epochs=20,
           validation_data=test_generatorCOVID,
           validation_steps=nb_test_samplesCOVID // batch_size)

# Produce accuracy curve (Figure 13 (a))
print(historyCOVID.history.keys())
plt.figure()
plt.plot(historyCOVID.history['acc'], 'orange', label='Training accuracy')
plt.plot(historyCOVID.history['val_acc'], 'blue', label='Testing accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Produce loss curve (Figure 13 (b))
plt.figure()
plt.plot(historyCOVID.history['loss'], 'red', label='Training Binary Cross-Entropy
    Loss')
plt.plot(historyCOVID.history['mean_absolute_error'], 'orange', label='Training Mean
    Absolute Error')
```

```
plt.plot(historyCOVID.history['val_loss'], 'green', label='Testing Binary
    Cross-Entropy Loss')
plt.plot(historyCOVID.history['val_mean_absolute_error'], 'purple', label='Testing
    Mean Absolute Error')
plt.xlabel('Epochs')
plt.ylabel('Error')
plt.legend()
plt.show()

# Mean average values of BCE for training and testing respectively
print(sum(historyCOVID.history['loss'])/20)
#0.5515911644945543
print(sum(historyCOVID.history['val_loss'])/20)
#0.6705862277593369

# Produce accuracy measures
loss = modelCOVID.evaluate_generator(train_generatorCOVID, len(train_generatorCOVID),
    workers = 1)
print(modelCOVID.metrics_names)
print(loss)
print("Accuracy = ", loss[1])
```