Using Deep Learning for X-Ray Image Classification

Michael Harrison

Submitted for the Degree of Master of Science in

Machine Learning

Department of Computer Science
Royal Holloway University of London
Egham, Surrey TW20 0EX, UK

August 30, 2018

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count:

Student Name: Michael Harrison

Date of Submission:

Signature:

Acknowledgement

I would like to thank Alberto Paccanaro for giving me the initial spur for this project, and for providing help and support along the way.

Abstract

This report covers the use of Deep Learning to perform X-ray image classification. In particular, on the MURA (muskuloskeletal radiographs) dataset - a collection of over 40,000 X-rays of various parts of the upper extremity such as hand, shoulder or elbow. The classification task was to distinguish normal from abnormal X-rays. The Deep Learning aspect was to build and train a convolutional neural network to do this. We trained a model that achieved 77% accuracy on the validation data. We then submitted this model to a competition held by the MURA authors, and it will be assessed against an unseen test set. However at the time of submitting this project, we are awaiting this result.

Contents

1 Introduction **1**

2 Background **2**
2.1 Deep Learning . . . 2
2.2 Artificial Neural Networks . . . 2
2.2.1 Basic Structure . . . 2
2.2.2 Activation Functions . . . 4
2.3 Training . . . 6
2.3.1 Loss Functions . . . 7
2.3.2 Backward Pass . . . 8
2.3.3 Weight Update . . . 8
2.3.4 Optimisers . . . 8
2.3.5 Weight Initialisation . . . 9
2.4 Regularisation . . . 10
2.4.1 Dropout . . . 10
2.4.2 Batch Normalisation . . . 11
2.5 Convolutional Neural Networks . . . 12
2.5.1 Image Representation . . . 13
2.5.2 Convolution Layers . . . 13
2.5.3 Padding . . . 15
2.5.4 Pooling Layers . . . 16
2.5.5 Prediction . . . 17
2.5.6 Training . . . 18
2.6 Performance Measures . . . 19
2.6.1 Confusion Matrix . . . 19
2.6.2 Cohen's Kappa . . . 20
2.6.3 ROC Curve . . . 20

3 State of the Art **22**
3.1 VGG . . . 23
3.2 GoogLeNet . . . 24
3.3 ResNet . . . 25
3.4 GoogLeNet-v4 . . . 27

4 Data **30**
4.1 Data Collection . . . 31
4.2 Abnormality Analysis . . . 32
4.3 Data Breakdowns . . . 33

4.3.1 Numbers of Studies 33
4.3.2 Numbers of Images 33
4.3.3 Studies of Multiple Body Parts 34
4.3.4 Consecutive Studies 36
4.3.5 Alternative Data 37

5 Model Development **39**
5.1 Working Environment 39
5.2 Data Preprocessing 39
5.3 Model 40
5.3.1 Simple Baseline 40
5.3.2 Pre-Built Architectures 41
5.3.3 Training 42
5.4 Predictions 42

6 Results **44**
6.1 Benchmark - MURA Authors 44
6.2 Baseline Performance 46
6.2.1 Vanilla Model 46
6.2.2 Regularisation 47
6.2.3 Learning Rate Decay 48
6.2.4 Weight Initialisation 49
6.2.5 Vanilla Model Reruns 50
6.2.6 Alternative Architectures 51
6.3 Pre-Built Models 53
6.3.1 VGG16 53
6.3.2 Inception-ResNet-v2 55
6.4 MURA Competition Submission 56

7 Professional and Ethical Issues **58**

8 Self-Assessment **60**
8.1 Project Next Steps 60
8.2 Learning Outcomes 61

References **62**

A How To Use My Model **65**
A.1 Stucture 65
A.2 Dependencies 66
A.3 Packages 67

A.4 Other Notes 67

B Background - Further Details **68**

B.1 Linear Activation Functions 68

B.2 Calculating Gradients in Backward Pass 68

B.3 Gradient Descent 71

B.4 Stochastic Gradient Descent 72

B.5 RMSProp 72

B.6 Adam 73

B.7 Batch Normalisation 74

C Pre-Built Model Architechtures **76**

C.1 VGG16 76

C.2 Inception-ResNet-v2 77

D Image Examples **78**

D.1 Elbow 78

D.2 Finger 79

D.3 Hand 80

D.4 Humerus 81

D.5 Forearm 82

D.6 Shoulder 83

D.7 Wrist 84

List of Figures

1 Example of a simple artificial neural network 3
2 Confusion Matrix Example 19
3 Example of a ROC Curve plot 21
4 ImageNet Classification Winners 22
5 Inception Module 24
6 Skip Connection 26
7 Error rates of plain networks on CIFAR image dataset 26
8 Bottleneck Block 27
9 Efficient Dimension Reduction Block 29
10 Studies Breakdown by Part 33
11 Images Breakdown by Part 34
12 Patients by Number of Body Parts 35
13 Breakdown of body part co-occurence 35
14 Numbers of studies per patient's body part 36
15 Results of Consecutive Studies 37
16 Breakdown of study outcomes 37
17 Simple Baseline Structure 40
18 Alternative Baseline Models 41
19 MURA Authors Results - Test Data 44
20 MURA Leaderboard Top 10 - as of 30th August 2018 45
21 Small Baseline Model - Image Size 48×48 46
22 Studywise Performance - Vanilla Model 47
23 Vanilla Model with Batch Normalisation 47
24 Vanilla Model with Dropout 48
25 Vanilla Model with Learning Rate Decay 49
26 Vanilla Model with Alternative Weight Initialisation 50
27 Vanilla Model Reruns 51
28 Vanilla Model (32 vs 128 initial filters) 52
29 Vanilla Model - Shallower and Deeper Versions 52
30 VGG16 53
31 Inception-ResNet-v2 55
32 ROC Curves - Model Selection 56
33 Results by Site - VGG16 & Inception-ResNet-v2 57
34 VGG16 Architecture 76
35 Inception-ResNet-v2 Architecture 77
36 Inception Block with Skip Connection 77
37 Patient 05926; Study 1; Elbow; Normal 78
38 Patient 00234; Study 1; Finger; Abnormal 79

39 Patient 09836; Study 1; Hand; Abnormal 80
40 Patient 11225; Study 1; Humerus; Normal 81
41 Patient 11417; Study 1; Forearm; Abnormal 82
42 Patient 00001; Study 1; Shoulder; Abnormal 83
43 Patient 01521; Study 1; Wrist; Abnormal 84

1 Introduction

The aim of this project is to explore the use of Deep Learning in a practical situation. Deep Learning techniques have been used in a range of problems, such as image recognition, natural language processing, playing games and self-driving cars, among others. These techniques have often achieved cutting-edge performance on these tasks - for instance, surpassing human performance in the ImageNet task, or beating the world champion Lee Sedol at the game Go. As such, Deep Learning is an exciting branch of Machine Learning with the potential to make significant impacts in many areas of life. Indeed Deep Learning is one of the largest areas of active research in Machine Learning at the time of writing.

This project focuses on image recognition, as this is one of the areas where Deep Learning has seen its biggest successes. In particular, we will be applying Deep Learning to the MURA (**mu**sculoskeletal **ra**diographs) dataset, published by the Machine Learning Group at Stanford University [1]. This is a collection of 40,005 X-ray images of a part of the upper extremity - comprising the arm, shoulder, wrist, hand etc. Each image is from one of 14,656 studies of an individual patient, performed at a particular point in time on one of these parts of the upper extremity. Each study was labelled as normal or abnormal at the time of clinical interpretation by a radiologist from Stanford Hospital - and these labels have been published alongside the images themselves. Therefore the aim of this project is to use Deep Learning to perform binary image classification on this set of X-rays - classifying them as either normal or abnormal. This will be a supervised learning task, since we have clear labels for our data, and our aim is to predict them.

While any model developed as part of this project can only represent a toy solution to this problem, the principle of combining Deep Learning and medicine seems to be a good one. Indeed this too is an area of active research, both within radiology, medical imaging, and medicine more widely. Thinking optimistically, Deep Learning can help improve patient outcomes, produce results more quickly, alleviate pressure off medical pracitioners, reduce medical mistakes and improve medical decision-making. As such, this represents good motivation for me to dip my toe into this area as part of my project.

2 Background

This section describes the background of Deep Learning, and the methodology behind the techniques used in this project.

2.1 Deep Learning

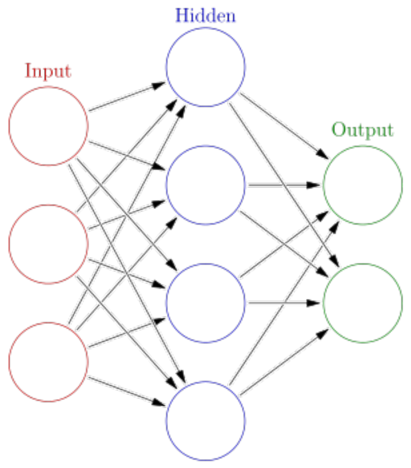
The main idea behind Deep Learning is for a system to learn representations of the data given to it. While these representations can start out simple, they are organised into a hierarchical sequence of layers so that simpler representations from earlier layers can be composed and built up into higher-level, more complex representations. This is repeated over many layers so that the final representations have sufficient information and detail for the task at hand. This process of building up complex representations from simpler parts means that feature engineering - the choice of which aspects of the data to include in the model, how they should be transformed or combined etc - is performed automatically by the system, as part of training. This is one of the great strengths of Deep Learning since feature engineering as a manual process can be difficult and time-consuming, especially when the data is very high-dimensional (as with images).

2.2 Artificial Neural Networks

2.2.1 Basic Structure

More specifically, these Deep Learning systems tend to be artificial neural networks (ANN's), of various forms depending on the nature of the task. A simple version of such a network [2] is shown in Figure 1 below. These consist of a set of "neurons" (the circles in the image below) organised into layers, with each neuron from one layer connected to all neurons in the next layer. Each neuron has an associated activation, typically just a real number, which is derived from the activations of the neurons feeding into it from the previous layer. The input layer is a special case, where the neuron activations represent values from the item of data being fed into the network. For instance, for greyscale images, the input neurons correspond to each pixel in the given input image, and the input activations are simply each pixel's value. The activations of the final output layer may represent different things, depending on the task - for instance, the final learned representation of the data item when doing dimension reduction. Or, for classification, the probability distribution over the set of classes that captures the network's prediction of which class the given data item is in.

Figure 1: Example of a simple artificial neural network



The connections between the neurons from one layer to the next represent the network's weights, each weight again usually just a real number. In addition to these weights between layers, each neuron (aside from the input layer) has an associated bias, once again usually just a real number. Taken across the whole network, these weights and biases form the set of parameters that define the model. They dictate how the activations from neurons in the previous layer should be used to calculate the activations of neurons in the next layer. For instance, let the activations of the neurons in layer i be given by the vector $a^{[i]}$, and the weights connecting neurons in layer i to those in layer $i+1$ be given by the matrix $\mathbf{W^{[i+1]}}$. So if the numbers of neurons in layers i and $i+1$ respectively are n_i and n_{i+1}, then $a^{[i]}$ will have dimension $n_i \times 1$, and $\mathbf{W^{[i+1]}}$ dimension $n_i \times n_{i+1}$[1]. Then the activations for the neurons in layer $i+1$ are given by:

$$a^{[i+1]} = f_{i+1}(\mathbf{W^{[i+1]}}^\top a^{[i]} + b^{[i+1]})$$

where:

- $b^{[i+1]}$ is the vector of biases for the neurons in layer $i+1$ (so of dimension $n_{i+1} \times 1$).
- f_{i+1} is the activation function for layer $i+1$, that applies element-wise

[1]Each of the n_i neurons in layer i connects to each of the n_{i+1} neurons in layer $i+1$, making $n_i \times n_{i+1}$ connections in total.

to the vector on which it operates, and hence returns a vector. The choice of activation functions is part of the design of the network.

This process is repeated, with activations from the input layer feeding into activations of the first hidden layer; activations from this hidden layer feeding into the next hidden layer; etc until the final activations in the output layer are produced. Note that while Figure 1 above showed only one hidden layer between input and output, in practice there can be arbitrarily many hidden layers - each taking as input the activations from their previous layer, and sending their own activations as output to the next layer. The choice of the numbers of hidden layers, and the numbers of neurons in each of these layers is again part of the design of the network.

Such networks are called fully-connected networks, and theirs layers with every neuron connecting to every neuron in the next layer are called *fully-connected* or *dense* layers.

2.2.2 Activation Functions

The activation functions are usually chosen to be non-linear - since otherwise, composing several layers of neuron activations reduces to taking a linear function of the original input neurons, and the network itself simply becomes a linear function. See Appendex B.1 for the mathematical argument. Training the network is effectively equivalent to just performing linear regression. This is undesirable since such a network cannot learn more complicated, non-linear features of the data.

Therefore non-linear activation functions are usually preferred. In this project we have used the following activation functions, in line with standard practice for image classification.

ReLU
The Rectified Linear Unit (ReLU) is a scalar activation function that takes a linear function of its argument above a certain threshold m, and is constant otherwise:

$$f(x) = max\{x, m\}$$

Note that m is a hyperparameter, specified in advance of training, rather than learned during it. Typically it is just set to 0 - as otherwise this implies some sort of prior knowledge about the scale of the "pre-activation" values $\mathbf{W^{[i+1]}}^\top a^{[i]} + b^{[i+1]}$ for all the various layers and neurons in the network (since m could theoretically be set on a neuron-by-neuron basis). This would be difficult to justify, especially since adjusting the level of these pre-activation values is already handled by the bias term $b^{[i+1]}$, a parameter that is learned during training.

This activation function introduces a non-linearity at the point $x = m$, since its slope jumps from 0 to 1 as x increases beyond m. While the function is linear above m, this non-linearity at m has proven sufficient in practice to obtain good results with ANN's.

Softmax
The Softmax can be regarded as a vector-to-vector function, that takes as input a vector v, of length (say) n, and produces an output vector w with the same length as v. The j^{th} component of w is given by:

$$w_j = f(v)_j = \frac{e^{v_j}}{\sum_{k=1}^{n} e^{v_k}}$$

where v_j is the j^{th} component of v. Since the Softmax involves exponentiation, it is clearly a non-linear function of its inputs, as desired. Now $e^x \geq 0 \ \forall \ x \in \mathbb{R}$, so we have $w_j \geq 0$. And, by definition:

$$\begin{aligned}\sum_{j=1}^{n} w_j &= \sum_{j=1}^{n} \frac{e^{v_j}}{\sum_{k=1}^{n} e^{v_k}} \\ &= \frac{\sum_{j=1}^{n} e^{v_j}}{\sum_{k=1}^{n} e^{v_k}} \\ &= 1\end{aligned}$$

Hence $w_j \in [0, 1] \ \forall \ j = 1, \ldots, n$ and so w represents a probability distribution over the set of values $1, \ldots, n$. This is true regardless of the scale or values of the input v.

This is extremely useful for classification, since we can set the final layer of our ANN to have the same number of neurons as numbers of classes, and let it use the Softmax activation function. Then each neuron would correspond to one of the w_j above, and the activations across the whole output layer would form a probability distribution over the set of classes. The network then provides a means to go from, say, an image to predicted probabilities of which class the image belongs to. More concretely, for our purposes we set the final layer to have 2 neurons - corresponding to normal or abnormal respectively.

2.3 Training

After choosing the structure of the ANN - i.e. the numbers of layers, numbers of neurons, and activation functions for each layer/neuron - it remains to train the network. The act of training the network boils down to setting appropriate values for all the weights and biases across the whole network. While this could be done by hand - this would be very difficult and, for larger networks, the numbers of weights and biases can be several million and hence infeasible to set by hand anyway. Instead, the weights[2] are initialised to random values and then automatically updated through a process called *Backpropagation*. This involves the following steps:

- **Forward Pass:** data items are input into the network (via the input layer). The activations of each of the subsequent layers are then calculated based on this input, up to the final output layer. Then (since our problem is supervised learning) the output is compared to known, true label for the given data item. This comparison is quantified by a loss function, some chosen function that measures how close the prediction is to the true label.

- **Backward Pass:** Since the value of the loss function depends on the network's weights (via the predicted label), the gradient of the loss function with respect to each of the weights can be calculated. These gradients then tell us how to adjust (in what direction, by how much) each of the weights in order to reduce the loss - and hence move the prediction closer to the true label.

- **Weight Update:** It remains to make these changes to the weights. Typically we only make a small change vs that indicated by the gradient - since the gradients themselves will be subject to a lot of random variation from data item-to-data item. So making the full change will result in noisy, erratic weight changes, and hence also erratic changes in the predictions with each weight update. So instead we multiply all of the gradients by some small number η (e.g. $\eta \approx 0.001$) called the *learning rate*. This has the effect of slowing down the weight updates, and hence allowing a more gradual progression towards a minimum of the loss function.

There are in fact a number of different training algorithms, which vary in the details, but all of the ones we consider follow this basic pattern.

[2]We'll use "weights" as a shorthand for "weights and biases" - i.e. the set of model parameters - except where a distinction between the two is necessary.

2.3.1 Loss Functions

As mentioned, loss functions are used to measure how far away predicted labels, generated by the network, are from the true label. Predictions that are "close" to the true label should have a smaller loss than predictions that are further away. For any item of data, both the activations in the network's input layer, and the item's true label are fixed. So the only aspect of a given network that we are free to vary are its weights, since these determine how the prediction is generated for an item of data. Therefore we can regard the loss as a function of the network's weights - and the goal of training is to set these weights so as to minimise the expected loss over a collection of data items.

The loss function we have used in this project is the cross-entropy loss, again in line with standard practice for image classification.

Cross-Entropy Loss
The cross-entropy loss assumes that the true label y (i.e. the class to which the data item belongs) is coded as a "one-hot" vector. If there are n classes, y is a vector of length n, with each component corresponding to one of these classes. All components of y are set to 0, except for the component corresponding the class to which y belongs - which is set to 1.

So for our x-ray data we code the two classes as follows:

- Normal: $[1, 0]$
- Abnormal: $[0, 1]$

The predicted label $\hat{y}$, per the Softmax activation function in the output layer, represents a probability distribution over the n classes - and so has the same dimension as y.

The cross-entropy loss is then given by:

$$L(y, \hat{y}) = -\sum_{j=1}^{n} y_j log(\hat{y_j})$$

Note that since y has all components 0, except for a 1 corresponding to the true label, the loss above reduces to taking the logarithm of the predicted probability of the true label. The sign of the logarithm is negated, so that smaller probabilties, which would give larger negative values of log, give larger positive loss values. The loss is therefore monotonically decreasing with the predicted probability of the true label - which is as desired, since predicting a high probability for the true label should have a smaller loss than predicting a lower probability.

2.3.2 Backward Pass

After performing a forward pass and calculating the loss for a given data item, we now calculate the gradients of the loss with respect to all the weights and activations in the network. This is called a backward pass since the gradients in one layer depend on those in the subsequent layer. Hence the process starts with calculating gradients in the output layer, then the gradients for the previous layer, and so on all the way back through the whole network. While the weight update step of the Backpropagation algorithm only uses the gradients of the weights, these gradients themselves depend on the gradients of the activations of the neurons which depend on these weights. Thus the backward pass involves calculating the gradients of activations, even though these are not strictly used in the weight update step. For the mathematical details of calculating gradients during the backward pass, see Appendix B.2.

2.3.3 Weight Update

The gradients calculated in the backward pass above give the change in weight to make, in order to increase the loss function. Since we aim to decrease our loss function, we subtract the gradients from the current values of all the weights. However, as mentioned, we first multiply the gradients by a hyperparameter η, called the learning rate, in order to avoid large, erratic changes in weights (and hence predictions) after each weight update. That is, for a weight (or bias) w in the network, with gradient $\frac{\partial L}{\partial w}$ we apply the following update:

$$w = w - \eta \frac{\partial L}{\partial w}$$

2.3.4 Optimisers

The following algorithms are all variations on the general theme of Backpropagation described above. That is, they all concern how to set the weights in order to minimise the loss function, based on the gradients of the loss function (hence the algorithms are called "optimisers").

RMSProp
One of the issues with ordinary gradient descent or stochastic gradient descent (see Appendix B for information about these) is that the learning rate η is a global hyperparameter that is fixed for all the weights in the network, and for all weight update iterations. However in practice, some weights in the network will require larger or smaller updates than others and, over the course of training, the weight updates will usually need to get smaller and smaller for successive iterations, as we approach a minimum of the loss function. RMSProp aims to solve these problems by effectively allowing the learning rate to vary during training and for each weight. It does this by taking a moving average of the gradients during training, and dampening the learning rate by this factor. See Appendix B.5 for the mathematical details.

Adam
The Adam (”Adaptive Momentum”) optimiser is a variation on RMSProp that takes account of momentum in the gradients from previous epochs. That is, large gradients on several previous iterations should not be ignored if the current gradient (by chance) happens to be very low, or vice versa. The momentum effect produces smoother and less erratic weight updates than RMSProp. To this end, Adam maintains an moving average of the gradient, as well as the squared gradient of RMSProp. This has the same benefits as RMSProp of dampening the learning rate η during training, but the weight update uses the smoothed rather than the raw gradient at each iteration. See Appendix B.6 for the formulas.

2.3.5 Weight Initialisation

Prior to training, all weights and biases in the network need to be initialised to some reasonable values. For this project we have used the following weight initialisation scheme, since that is the default in the software (Keras) we used to train our models.

Glorot Uniform
This initialisation scheme was introduced by Xavier Glorot and Yoshua Bengio in 2010 [3]. It initialises biases to 0, and the weights $w_{i,j}$ in each layer randomly according to the distribution:

$$w_{i,j} \sim U[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}]$$

where $U[a, b]$ is the continuous random uniform distribution over the interval $[a, b]$, and n is the number of neurons in the previous layer.

In fact, examining the Keras documentation and source code [4], Glorot Uniform is implemented slightly differently to how it was introduced by Glorot & Bengio. Biases are still initialised to 0. However weights are drawn randomly from $U[-limit, limit]$ with $limit = \sqrt{\dfrac{6}{fan\_in + fan\_out}}$ where fan_in is the number of input neurons to the weight tensor, and fan_out the number of output neurons. In our terms, if $\mathbf{W}$ is the $n_i \times n_{i+1}$ matrix of weights connecting layers i and $i + 1$, then each element w of $\mathbf{W}$ will be initialised as above, with $fan\_in = n_i$ and $fan\_out = n_{i+1}$.

2.4 Regularisation

Regularisation is used to prevent overfitting by neural networks - i.e. the fact that, while the network may perform well on the data on which it was trained, its performance on another set of data is poor. This usually occurs because, while training, the network captures too much of the random noise present in the training data, and therefore does not transfer well to another data set where that noise is not present (instead having its own random noise). So regularisation aims to ensure that training leads networks to capture genuine features of the data that will be present in other data sets, rather than that random noise. We describe some regularisation techniques below:

2.4.1 Dropout

Dropout was introduced by Srivastava et al in 2014 [5]. The idea is, for a given network layer, to randomly remove some neurons from that layer. At each training iteration, e.g. for each batch in SGD, each neuron and its associated weights and biases are retained in the network with some chosen probability p, and removed with probability $1 - p$. This random selection is performed fresh for each iteration, so that each iteration effectively uses a slightly different network architecture. That means that the paths for information to travel through the network - the sequence of neurons and connections between them - vary. This introduces redundancy in that the same genuine features of the data are learned by several routes through the network, since the available routes change with each iteration. This reduces the likelihood that parts of the network just start capturing noise, as for any given iteration those parts of the network may or may not be present. Further, for any given epoch, each neuron will only see an expected fraction p batches of training data, and is therefore exposed to less noise in the data. So the neuron's weights will be less likely to capture noise.

Dropout also reduces the effective capacity of the network layer at each iteration by the factor p. A layer with 10 neurons and dropout probability 70%, would only use an expected 7 neurons. This reduces the scope of the layer to capture noise in the data, since there are fewer effective parameters. So there is less "room" for the network layer to pick up noise.

After training, say when testing the network on a validation set, we revert to the full original network architecture - neurons are no longer dropped out. Instead, a trained weights w in a given dropout layer are replaced with pw. A forward pass can then be conducted as normal.

2.4.2 Batch Normalisation

Batch normalisation was intoduced by Ioffe and Szegedy in 2015 [6]. It involves normalising the activation values for each neuron in a given layer across all data items in each SGD batch. The normalisation is done with respect to the mean and variance of the activation values for the batch - so that the activations after the batch normalisation will have mean 0 and variance 1 across the batch. These values are then scaled and shifted by two parameters, learned separately for each neuron during training. For further information, see B.7.

2.5 Convolutional Neural Networks

Convolutional neural networks (CNN's) are a variant of artificial neural networks described above, that are often used for image processing tasks such as classification. They are organised as a sequence of "convolution" layers, where the first layer operates directly on an image to produce a representation of it comprised of various low-level features, such as edges or patches of light or dark etc. The next layer then operates on this representation, producing another representation that captures slightly higher-level features, based on compostions of the low-level features from the first layer. This process is repeated for a chosen number of layers, with each layer outputting a representation based on the one from the previous layer. The final output from the sequence of convolution layers is then some representation of the original image built up from the low- and mid-level features from past layers. This final representation can then be used for the task at hand - for instance fed through a fully-connected layer, into a Softmax output layer for classification.

The strength of CNN's is that the representations they produce can pick up features regardless of where they appear in the image. So if a representation captures the shape and colour patterns of eyes (say), it will "recognise" eyes that appear at any position in the image. This property is called *translational invariance.* By contrast, if the original image was fed through a sequence of fully-connected layers, different weights in the network would correspond to different positions in the image. So that any feature learned by some subset of the weights would only be recognised if it appeared at the corresponding position in the image.

Another strength is that the numbers of weights in convolution layers are not tied to the size of the image, whereas a fully-connected layer would be. Therefore the number of weights in a given layer can be significantly smaller than for a corresponding fully-connected layer. This makes training the network easier, since there are fewer weights to set. For instance, consider a 100×100 image fully-connected to a hidden layer of 10 neurons - this would have $100 \times 100 \times 10 = 100,000$ weights. Meanwhile a convolution layer might have, say, 30 3×3 filters - making $30 \times 3 \times 3 = 270$ weights. This independence from the size of images means that previously-trained CNN's can be used on different sets of images (potentially of entirely different dimensions) from those it was originally trained on, and still identify the same features in both sets of images. This makes CNN's very convenient for transferring from one image processing task to another - so-called *transfer learning.*

2.5.1 Image Representation

Throughout this section, we represent an image as either a 2D or 3D grid of pixel values, where the height and width of the image are the first two dimensions, and the channels of the image are the third dimesion. For greyscale images, there is only one channel, so we regard the image as being 2D. For colour images coded as RGB (say), there are three channels (red, green, blue) so we regard the image as 3D with depth 3. The height and width are sometimes called the spatial dimensions, to distinguish them from the channels.

2.5.2 Convolution Layers

A convolution layer represents a collection of *filters* of a chosen dimension - say, 3×3. Each element of a filter is a real number, and is one of the weights of the convolution layer to be learned during training. The filter will operate on all channels of its input image or representation (from previous convolution layers). So a 3×3 filter working on an RGB image will have $3 \times 3 \times 3 = 27$ weights, since the image has 3 channels. The output of each filter in a convolution layer is passed to the next convolution layer as a separate channel - so that if the first layer has (say) 32 filters, the filters in the next layer will have a depth of 32. If the next layer uses 5×5 filters, each one will have $5 \times 5 \times 32 = 800$ weights[3].

Each filter performs a convolution operation between its input and its weights. For a one-channel input, this is defined as follows. Let $\begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix}$ be a 2×2 filter and b be its associated bias. Let $\begin{bmatrix} p_{1,1} & p_{1,2} \\ p_{2,1} & p_{2,2} \end{bmatrix}$ be the values for some contiguous 2×2 region of the input. The convolution is then:

$$z = \sum_{i=1}^{2} \sum_{j=1}^{2} w_{i,j} p_{i,j} + b$$

$$a = f(z)$$

[3]In fact, each filter has a single associated bias term (regardless of depth), so the filters mentioned would have 28 and 801 parameters respectively.

That is, we take the element-wise product of the corresponding weights and input values, then sum all of them. The function f is some chosen activation function (e.g. ReLU). For a multiple-channel input, the filter and input will be 3-dimensional, and the operation would instead apply to a contiguous 3D volume of the input. For instance, if the input were an RBG image and the filter were now of dimension $n \times n$, the convolution would look like:

$$z = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{3} w_{i,j,k} p_{i,j,k} + b$$

$$a = f(z)$$

This convolution operation is repeated multiple times, over many positions in the input. It first applies at the top-left corner of the input - for the one-channel 2×2 filter, say this is $\begin{bmatrix} p_{1,1} & p_{1,2} \\ p_{2,1} & p_{2,2} \end{bmatrix}$ - with result $a_{1,1}$. Then the filter will shift to the right some chosen number of columns, called the *stride*, and perform the convolution operation again on this new region of the input - with result $a_{1,2}$. However for convolution layers, the stride is usually just set to 1 - so that for our 2×2 filter, the second position is $\begin{bmatrix} p_{1,2} & p_{1,3} \\ p_{2,2} & p_{2,3} \end{bmatrix}$. This continues on until the right-hand edge of the input is reached, at which point the filter moves down some number of rows, again given by the stride. So if the input has width w, the 2×2 filter with stride 1 would now apply to the region $\begin{bmatrix} p_{2,w-1} & p_{2,w} \\ p_{3,w-1} & p_{3,w} \end{bmatrix}$ - with result $a_{2,w-1}$. Then filter would then move from right-to-left over this new row, before moving down to the next row. This continues until the filter has performed a sweep over the whole input, applying the convolution to the corresponding input values at each position.

This produces a set of values $\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots \\ a_{2,1} & a_{2,2} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$ which we call the representation of the input, produced by the filter. These are organised spatially into a 2D grid based on the position on the input to which they correspond - i.e. $a_{1,1}$ corresponds the to top-left position etc. If the convolution layer consists of several filters, each one will produce one of these 2D grids, and we stack the set of these grids into a 3D volume - with each grid considered a separate channel. This 3D volume is then the layer's output representation, on which the next convolution layer will operate.

The set of weights that consitutes a filter are kept the same for every convolution operation at every position on the input. This explains the translational invariance property, since the filter will pick up the same feature, wherever in the input it appears.

Filters are usually chosen to be square - i.e. to have the same height and width - so as not to bias them in advance towards picking up features of one orientation over another. A filter that was taller than it was wide would be more likely to pick up vertically-oriented features. Instead we would rather the filter to be initially "neutral" in that regard, and learn to pick up vertical features during training if appropriate.

Filters are also usually chosen to have odd height and width, rather than even (e.g. 3×3 rather than 2×2). This is so that the filter has a well-defined center, and we can think of the position of the filter on the input as that corresponding to the central weight in the filter. So if a 3×3 (one-channel) filter covers the input region $\begin{bmatrix} p_{x-1,y-1} & p_{x,y-1} & p_{x+1,y-1} \\ p_{x-1,y} & p_{x,y} & p_{x+1,y} \\ p_{x-1,y+1} & p_{x,y+1} & p_{x+1,y+1} \end{bmatrix}$, we can call the result of the convolution operation $a_{x,y}$. This produces a convenient alignment between the input and the representation produced from it, and in the absence of a compelling reason for an even-dimension filter we might as well use it.

2.5.3 Padding

Padding allows for the representation produced from an input to have the same height and width as the input. Note that if we position a 3×3 filter in the top-left corner of the input, its central weight would not correspond to the top-left pixel[4] - instead to the pixel down-and-to-the-right by one. And similar when the filter is postioned in the top-right, bottom-left etc. This means that there would be no positions in the output representation corresponding to the top or bottom rows of the input, nor the rightmost or leftmost columns. So an $h \times w$ input would reduce to a $h - 2 \times w - 2$ representation. Padding inserts values of 0 around the edge of the input - effectively increasing its dimension to $h + 2 \times w + 2$ - so that the top-left position can correspond to the actual top-left pixel of the input. So the

[4]We'll use "pixel" loosely to mean either one cell in the original image (thought of as a 2D or 3D grid), or one cell in the representation produced by a convolution layer.

resulting output representation would then be $h \times w$ as desired. The amount of padding layers will depend on the dimension of the filter - a 5×5 filter would have 2 such padding layers, and so on.

2.5.4 Pooling Layers

Pooling layers are used to reduce the spatial dimensions of a representation, while still retaining its key features. This is also called downsampling. It is useful when using the final representation produced by a sequence of convolution layers for further tasks. For example, feeding it through a fully-connected layer in order to perform classification. If we had used padding, the final representation would have the same height and width as the original image, and potentially many more channels depending on the number of filters used in the final convolution layer. The fully-connected layer would therefore be extremely large, and have a very large number of parameters. Reducing the spatial dimension will also reduce the computational cost of a convolutional layer, since the number of positions on which needs to perform a convolution operation is reduced.

Moreover, downsampling in between convolution layers allows the subsequent layer to more easily capture spatial combinations of the features created in the previous layer. In particular doing so using smaller filter sizes (and hence fewer trainable weights) than would otherwise be necessary. That is, if the key features of an $h \times w$ representation are compressed into a smaller, say, $\frac{h}{2} \times \frac{w}{2}$ representation - an $n \times n$ filter would now be able to capture combinations of features that were further than n pixels apart in the original representation, if they are now within n pixels in the new representation. As this process continues over several layers, with representations getting progressively smaller, the convolution operations are better able to progress from isolated low-level features of the original image, to mid- then high-level combinations of such features that span the whole image.

The pooling layers themselves again involve a window of some chosen dimension, again usually square, and stride. However the stride is now usually (though not necesarily) set equal to the height/width of the window. The window will move across the input in the same manner as the filter in a convolution layer - in increments of the chosen stride. However instead of performing the convolution operation, it will instead perform some aggregation of the pixel values corresponding to its position on the input. For instance taking the maximum, the average or the sum of these values. This will then form the value for one cell in the represention produced by the pooling layer. Since it is simply aggregating pixel values, the pooling layer has no associated weights, and so does not need to be trained. Also, the pooling layer only operates on the two spatial dimensions - so for a multi-channel input, it will operate on each channel separately. The representation produced by the pooling layer will therefore have the same number of channels as its input, but a smaller height and width.

Consider a 2×2 window with stride 2. This will start in the top-left corner of the input and produce a single value. Then it will move two columns to the right to produce the next value. This means that if the input was dimension $h \times w$, the representation from the pooling layer will have dimension $\frac{h}{2} \times \frac{w}{2}$. Taking the max or summing the pixel values means that the key information is retained from the input representation - since large pixel values correspond to the presence of the feature that the filter (whose convolution operation produced the pixel values) has been trained to capture.

2.5.5 Prediction

In the context of image classification, the predictions we make from a convolution net are a probability distribution over a given set of classes. This is done using a Softmax as described above. The network itself will comprise of a sequence of convolution and pooling layers that produce some final representation of the input image. While this final representation is typcially a 3D volume, we introduce a "flatten" layer that treats each cell in this 3D volume as a neuron. So if the final output of the convolution layers is, say, a $10 \times 10 \times 5$ 3D volume, after flattening we get a 500 neuron layer. This flatten layer is then connected to an intermediate, fully-connected hidden layer, which is then connected to the output Softmax layer. So the connections from the flatten layer to the intermediate hidden layer form another set of of weights (and biases) to be learned, as do those from the hidden to the output layer. Of course, we could introduce further hidden layers prior to the output layer, if necessary. In effect, we can treat the flatten layer as the input layer of a separate, fully-connected network. We can think of this as the "head" of the convolutional network, while the preceding convolution and pooling layers form its convolutional "base".

This separation allows for transfer learning as described above. We can replace simply the head, while retaining the convolutional base - both its structure, and the weights and biases of its filters. This means that the base will still pick out the same features, combinations of features etc, in the same way when applied to a new set of images. We can then use these as inputs for the new network head, which may need to have a different design than the head with which the base was originally trained with. For instance, if our new image classification task involves a new (smaller or larger) set of classes than those originally trained on - the output Softmax layer would have to change to match the new number of classes.

One slight downside of using a fully-connected head, is that the size of the flatten input layer is dependant on the size of the input image - say all our filters use padding, and we have two pooling layers of window size 2×2 with stride 2. Then the final representation will have one quarter the height and one quarter the width of the original image. This means, if we're performing training over a set of images, each image in the set must have the same dimension. Otherwise, the flatten layer would vary in size with each image, and so the number of weights between the flatten and hidden layer in the head.

2.5.6 Training

Convolutional networks can be trained in the exactly the same way as for general artificial neural networks described above- that is, by Backpropagation via algorithms such as SGD or Adam. As mentioned, the weights for convolutional layers are the values in the filters. We can perform a backward pass through both the head and convolutional base to calculate the gradients of the loss with respect to the weights and biases in all filters, throughout the whole network.

However in transfer learning, the weights in the convolutional base are often left unchanged. Instead only the weights in the head are trained on the new set of data - they generally can't be reused (even if the head has the same structure) since they will be specific to the set of labels originally trained on. Whereas the convolutional base captures fairly generic features that will have relevance across a range of image processing tasks.

2.6 Performance Measures

We now introduce the measures we use to asses the performance of our trained network. While the network is trained using the cross-entropy loss, as described above, we can report other measures that capture other aspects of performance more explicitly e.g. the false positive or false negative rates. Typically we are interested on the performance of the network on an alternative data set to that on which it was trained - a so-called valiation set.

2.6.1 Confusion Matrix

The confusion matrix decomposes the correct and incorrect predictions for each class. An example for our two classes *Normal* and *Abnormal* is shown below:

Figure 2: Confusion Matrix Example

		True Label		
		Normal	**Abnormal**	**Total**
Predicted Label	**Normal**	a	b	$a+b$
	Abnormal	c	d	$c+d$
	Total	$a+c$	$b+d$	$a+b+c+d$

That is, the network predicted a examples with true label *Normal* as *Normal*, etc. So the accuracy rate would be $\dfrac{a+d}{a+b+c+d}$.

Now if we consider *Normal* to be "negative" and *Abnormal* to be "positive", we can define:

- **True Negative Rate** Proportion of true *Normal* labels predicted correctly: $\dfrac{a}{a+c}$
- **False Negative Rate** Proportion of true *Abnormal* labels incorrectly predicted as *Normal*: $\dfrac{b}{b+d}$
- **True Positive Rate** Proportion of true *Abnormal* labels predicted correctly: $\dfrac{d}{b+d}$
- **False Positive Rate** Proportion of true *Normal* labels incorrectly predicted as *Abnormal*: $\dfrac{c}{a+c}$

2.6.2 Cohen's Kappa

Cohen's Kappa was introduced by Jacob Cohen in 1960 [7]. It is a measure of agreement between two raters - so when applied to a corresponding set of true and predicted labels, it measures the accuracy of the predictions. Its advantage over simple accuracy is that is measures the extent of agreement over and above that which would be expected by chance. This is useful for instance when the class frequencies in the underlying data are naturally skewed. Consider data comprised of two classes, 80% A and 20% B - a classifier that simply always predicted A would have 80% accuracy, but can not be considered a very good classifier. Cohen's Kappa accounts for this underlying skew, and would score the classifier poorly. For the confusion matrix in Figure 2 above, it is defined as follows:

Let $p_o = \dfrac{a+d}{a+b+c+d}$ be the observed overall accuracy.

Now let $p_e = p_{normal} + p_{abnormal}$ be the probability of agreement at random, where:

$$p_{normal} = \frac{a+b}{a+b+c+d} \, \frac{a+c}{a+b+c+d}$$

$$p_{abnormal} = \frac{c+d}{a+b+c+d} \, \frac{b+d}{a+b+c+d}$$

Cohen's Kappa is then:

$$\kappa = \frac{p_o - p_e}{1 - p_e}$$

Cohen's Kappa usually ranges from 0, when the observed accuracy is simply equal to that expected at random (i.e. $p_o = p_e$) - to 1, when the observed accuracy is 100% (i.e. $p_o = 1$), regardless of the extent of expected random agreement p_e[5]. It can be negative, when the observed accuracy is *below* that expected by chance, but this should be rare in practice.

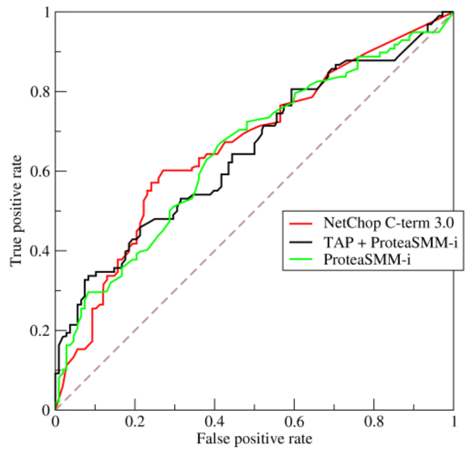
2.6.3 ROC Curve

The ROC curve is a way to illustrate the performance of one or more binary classifiers, allowing for convenient comparison. An example [8] is shown below:

It plots the true positive rate of the classifier against its false positive rate. A single run of the classifier on a set of data would only represent a single

[5]Ignoring the degenerate case $p_e = 1$.

Figure 3: Example of a ROC Curve plot



point on the plot. However, in practice, we vary the discrimination threshold of the classifier, e.g. the predicted probability for which it predicts a "positive" label, from 1 to 0. For a given classifier this then produces a series of points on the plot, which we connect to form the ROC *curve*. If the threshold probability is 1, all examples are classified as negative and hence both the true and false positive rates are 0. Reducing the threshold probability means more of the true positive labels are correctly predicted as positive - increasing the true positive rate. But this can introduce a higher false positive rate, since more of the true negative labels are predicted as positive. When the threshold reaches 0, all examples are classified as positive, so the true positive rate is 1, but the false positive rate is also 1 since all true negatives are predicted as positive.

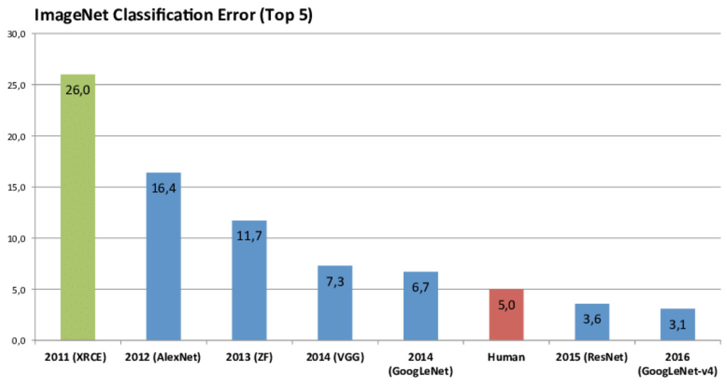
However, intermediate points on the ROC curve will indicate the extent to which the classifier can correctly classify positive examples (high true positive rate), while avoiding misclassifying negative examples (low false positive rate). Clearly, a classifier that is better at doing this is preferred, so we prefer those with ROC curves towards the top-left corner of the plot. And a classifier whose ROC curve is strictly above another's in the plot is a strictly better classifier (at least on the given data set).

We can summarise this over a whole set of threshold probabilities by calculating the area under the ROC curve, called the *AUC*. A classifier whose ROC curve is strictly above another's will have a larger AUC - so a larger AUC represents a better classifier.

3 State of the Art

In this section we review the current state of the art Deep Learning models used in image classification. For this we look to the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [9], a machine learning competition held each year between 2010 and 2017 to assess the performance machine learning systems on a variety of computer vision tasks. The competition is held on a subset of ImageNet, a database of 10,000,000 labelled images depicting over 10,000 object categories, published by Deng et al. in 2009 [10]. For instance, for the 2017 object classification and localisation tasks, the test and validation sets together comprised 150,000 photographs hand-labelled with the presence or absence of 1000 object categories. Each algorithm submitted to the competition was required to output 5 class labels, in decreasing order of confidence. The error was then based on matches between the true object categories and these top-5 predicted labels - giving the so-called top-5 error. Winners of the ILSVRC classification task, and their associated top-5 error rates are shown below (taken from [11]).

Figure 4: ImageNet Classification Winners



AlexNet in 2012 was the first winner to use Deep Learning with convolutional neural networks, and as can be seen achieved a considerable improvement in performance over the previous year's winner. Since then, all winners have used variations or extensions of the Deep Learning/convolutional network idea. Note that in 2014, the VGG team achieved first place on the localisation task and second place on the classification task, which was won by GoogLeNet. Below we describe some of these winning networks in more detail.

3.1 VGG

VGG [12] explored the idea of using very deep convolutional networks, with up to 19 weight layers. The networks were made up of a set of successive "blocks", each comprising successive 3×3 convolutional layers (with padding, ReLU activaiton function, and stride of 1) followed by one 2×2 max-pooling layer with stride of 2. The output of each block was the input to the next block. The convolutional layers within each block contained the same number of filters, but the number of filters increased with each block.

For instance, the VGG19 model contains five blocks, the first consisting of two convolutional layers each with 64 filters, plus the final max-pooling layer. The second block contains two convolutional layers each with 128 filters. The three remaining blocks all have four convolutional layers with, respectively, 256, 512 and 512 filters. This convolutional base then feeds into a head comprising a flatten layer, then two fully-connected layers and finally a Softmax layer. This gives it $2 + 2 + 4 + 4 + 4 + 3 = 19$ trainable layers, as the name suggests. The VGG16 model is similar, but the final three blocks have only three convolutional layers each - giving 16 trainable layers.

Note that one pixel in the output of two successive 3×3 convolutional layers (with stride 1) would correspond to the same region of the original input as the equivalent pixel in the output of one 5×5 convolutional layer (with stride 1)[6]. However the VGG team preferred to use stacks of smaller convolutional filters in order to keep the number of weights small (e.g. 18 vs 25), and increase the extent of non-linearity in the network - the two 3×3 layers each apply separate activation functions where the 5×5 layer applies just one. The idea is that this should increase the discriminative power of the network, allowing better performance.

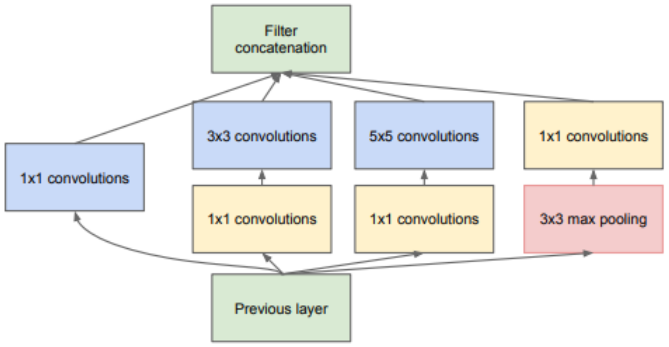
A schematic of the VGG16 architecture is shown in Appendix C.

[6]The latter would correspond to, say, $\sum_{i=1}^{5}\sum_{j=1}^{5} w_{i,j}p_{i,j}$ (ignoring bias and activation function). While the former would correspond to $\sum_{x=0}^{2}\sum_{y=0}^{2} w'_{x+1,y+1}(\sum_{i=1}^{3}\sum_{j=1}^{3} w_{i,j}p_{x+i,y+j})$. Both cover the same range of pixel values $p_{i,j}$.

3.2 GoogLeNet

GoogLeNet [13] introduced the idea of ”Inception” modules, illustrated in below (image from [13]):

Figure 5: Inception Module



The initial layer of 1×1 filters (with stride 1) performs dimension-reduction by reducing the number of channels in the input layer - the spatial height and width will be unchanged, but the number of channels will reduce to the chosen number of 1×1 filters. That is, they perform an aggregation across the channels for each pixel in the input, namely a weighted sum across the channels weighted by the filter's weight in each channel. This lets certain channels be prioritised or deprioritised as appropriate. It also reduces the number of channels that the subsequent 3×3 and 5×5 operations have to work on, reducing their numbers of parameters and computation time. For example, a 5×5 operation on an input with 100 channels would involve $5 \times 5 \times 100 = 2500$ parameters. Whereas using, say, 50 1×1 filters to first reduce the depth would involve in total $100 + (5 \times 5 \times 50) = 1350$ parameters. The number of computational operations required would also be reduced similarly.

The 1×1, 3×3 and 5×5 (all with stride 1) operations are performed in parallel, so that the module can pick up different features at different spatial scales simultaneously. Subsequent modules can then more easily capture combinations of highly local features (from the 1×1 filters) and more global features spanning a larger part of the spatial range (from the 5×5) filters). There is also a parrallel max pooling layer added, according to the authors, due to the success of pooling operations in other convolutional networks. Note it uses a stride of 1 so as not to change the spatial dimension, but is followed by a 1×1 convolutional layer to reduce the number of channels - since pooling operates on each channel separately and so would retain the original number of channels. Note that padding is used on all appropriate layers, including the pooling layer, so as to maintain the original spatial dimension, and to ensure all their outputs' spatial dimensions match. The results of all of these layers are finally concatenated, with each forming different channels of the module's final output representation.

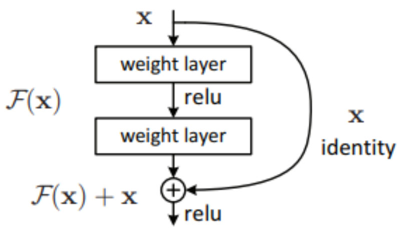
GoogLeNet itself consists of a stack of 9 of these inception modules chained together, each working on the output of the previous module. There are some preliminary convolutional and pooling layers before the first inception module (called the network's "stem"), and some intermediate max-pooling layers with stride 2 in the middle of the inception stack, to perform spatial dimension reduction. The output from the final inception module is then fed through an average-pooling layer, followed by a fully-connected and Softmax layers for the final classification. In addition, at earlier points in the stack there are two "auxiliary" classifiers parallel to the main inception stack, consisting again of average-pooling, fully-connected then Softmax layers. The loss from these auxiliary classifiers is scaled by 0.3 then added to the loss from the main stack. The idea is that these will help to improve the disciminative power of the earlier parts of the network - since during training any weight updates in these earlier layers will include terms in the gradient corresponding to these auxiliary classifiers, and so should improve their loss. This also helps the gradients from becoming progressively smaller, to the point of vanishing, as we move back through the main stack. GoogLeNet has in total 22 trainable layers - a similar order of magnitude to VGG.

3.3 ResNet

ResNet [14] introduced a technique called *residual* or *skip connections* that allowed networks of unprecedented depth to be trained successfully. The winning submissing to ILSVRC 2015 was a residual network with 152 layers, which was the deepest network ever presented on ImageNet at the time. An example of a skip connection is shown below (image from [14]). The identity connection simply carries forward the original input X and adds it element-wise to the output of the weight layers, $F(X)$.

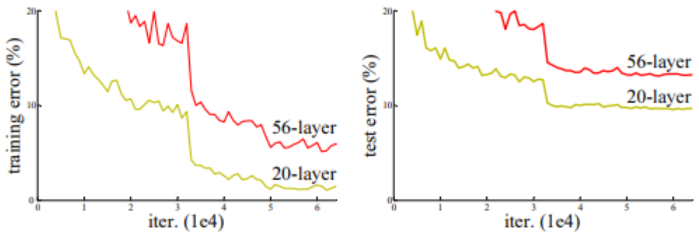
The idea is that, in this scheme, it is much easier for the network to learn the identity mapping $H(X) = X$, since it requires only learning weights

Figure 6: Skip Connection


x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

that make $F(X) = 0$ (i.e. weights and biases 0) if $H(X) = F(X) + X$. Whereas learning $H(X) = X$ directly is much more difficult if $H(X)$ is simply the output of some weight layers. The motivation behind easier identity mappings is the observation by the ResNet authors that, beyond a certain number of layers, the performance of plain convolutional network begins to degrade, even on the training set - and so is not simply due to overfitting. See the image below, again from [14].

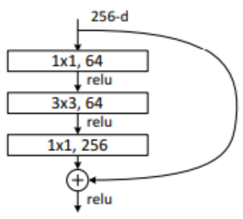
Figure 7: Error rates of plain networks on CIFAR image dataset



The authors hypothesise that this is due to the difficulty in finding good solutions (i.e. minima of the loss function) for such deep networks using current optimisation techniques (i.e. Backpropagation and its variants). Whereas, if later layers in the network simply perform identity mappings, then extending the network depth must perform as well as the smaller network since the output representations will be the same. Adding skip connections allows the later layers in deeper networks to learn the identity mapping, or close to it, if necessary - or not, if later non-identity layers do indeed improve performance.

The ResNet architechtures used the idea of skip connections in the form of a ”bottleneck” block, shown below:
The initial 64 1×1 filters perform dimension-reduction in the manner of

Figure 8: Bottleneck Block



GoogLeNet, to reduce the computation time of the 3×3 layers. The subsequent 1×1 filters project the output of the 3×3 back up to 256 channels, to match the block's input and so enable the skip connection. The spatial dimensions remain unchanged throughout the block, so the skip connection can perform element-wise addition $F(X) + X$ as required - and involves no parameters. Note that the core of feature detection is done by the 3×3 convolutional layer. These were kept small (as opposed to, say, 5×5 etc) in line with the philosophy of the VGG authors.

The winning ResNet-152 architecture then involved stacking these bottleneck blocks. The number of filters inside their 3×3 layers began at 64, and was periodically doubled after every few bottleneck blocks. To offset this doubling in terms of computation time, the spatial dimensions were also halved by using a stride of 2 in the 3×3 layer of the first subsequent bottleneck block. However this introduces a mismatch in the dimensions of the block's input and output, and hence in the skip connection. To overcome this, the skip connection was altered to be $F(X) + \mathbf{W_s}X$, where $\mathbf{W_s}$ is a matrix of trainable weights that projects X to the same dimension as $F(X)$. However this was only done intermittently; most bottleneck blocks use the parameter-free skip connection as described above.

3.4 GoogLeNet-v4

GoogLeNet-v4 [15] incorporated the idea of skip connections from ResNet into the Inception-based architecture of GoogLeNet, as well as other techniques from the -v2 and -v3 reviews of the original GoogLeNet architecture [16]. The Inception-v4 model used a number of different Inception blocks, all variations of the theme of the original block. The following good practices were identified in these reviews:

- Factorise large filter size convolutions: This seems to follow the insight from VGG that larger filters can be decomposed into successive 3×3 filters that, in combination, cover the same spatial range as the larger filter while having fewer parameters and being cheaper computationally. For instance, 5×5 can be decomposed into two 3×3 filters, 7×7 into three 3×3 filters, and so on. In particular, the 5×5 block in the original Inception module was replaced in this manner.

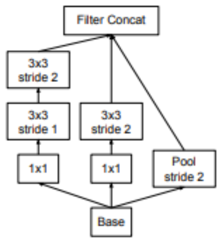
- Asymmetric convolutions: An alternative approach to factorising convolutions decomposes an $n \times n$ filter into two $1 \times n$ and $n \times 1$ filters.. The authors seem to organise these either in sequence or in parallel - for instance a 1×7 followed by a 7×1 in one of the inception blocks of Inception-v4, but parallel 1×3 and 3×1 in another. For the parallel version, horizontal and vertical features would feed forward as separate channels, but subsequent 1×1 layers in the following block should combine these by aggregating across channel. Whereas for the successive version, the vertical $n \times 1$ layer operates on the output of the horizontal $1 \times n$ layer, so capturing spatial features more directly. The rationale behind this design other than reduced computational cost and numbers of parameters is not clearly explained by the authors, beyond mentioning that they achieve "very good" results with successive 1×7 and 7×1 layers.

  Note that the authors use both the asymmetric and successive 3×3 forms of factorisation, in different blocks - with the former used earlier in the network and the latter used later, presumably simply as a result of experimentation since they mention the asymmetric approach does not work well on earlier layers. Although they do not hypothesise as to why.

- Efficient dimension reduction: the authors reccomend to avoid reducing the spatial dimension too much, too early in the network, as this introduces a bottleneck in the amount of information that can flow through to later layers. They claim that the method of inserting a pooling layer to reduce dimension prior to a convolution layer (or, say, Inception block) violates this principle - since the subsequent convolutions work on a compressed representation. Conversely, pooling after the convolutional layer is much more computationally costly, since it will many more convolution operations.

  The authors introduce a more computationally efficient alternative, that avoids the bottleneck issue. They build a "reduction block" using parallel pooling and convolution layers, both with stride 2 to reduce the spatial dimension, and concatenate the results. The convolutional layer therefore works on the original-dimension input, avoiding information bottleneck, while still reducing spatial dimension. For instance, the block in Figure 9 below (from [16]) has parallel 3×3 convolutional and pooling layers, and a parallel 5×5 convolutional layer decomposed into successive 3×3 layers. In GoogleLeNet-v4 the authors also did not use padding in these layers, so as to further reduce dimension.

Figure 9: Efficient Dimension Reduction Block



- Residual connections: residual connections were put across inception blocks, with the output of the block added element-wise to its input. This performs the same function as in ResNet, where the block and skip connection together can more easily learn the identity mapping, if required, and so enable larger stacks of inception blocks. For instance, the Inception-ResNet-v2 architecture comprised a stack of 40 inception blocks (of various types). One module from this architecture is shown in Appendix C below, showing the skip connection.

The winning GoogLeNet-v4 model was an ensemble of the Inception-v4 architecture - which incorporated the above factorisation and spatial reduction ideas, but did not use skip connections - and three Inception-ResNet-v2 architectures - which also included skip connections as well as the other ideas.

A schematic of the Inception-ResNet-v2 architecture is also shown in Appendix C.

4 Data

The data we will use in this project is the MURA (**mu**sculoskeletal **ra**diographs) dataset, published by the Machine Learning Group at Stanford University [1]. This is a collection of 40,005 X-ray images of a part of the upper extremity - comprising the arm, shoulder, wrist, hand etc. Each image is from one of 14,656 studies of 11,967 individual patients, performed at a particular point in time on one of these parts of the upper extremity. Example images for each part of the upper extremity are shown in Appendix D. Each study was labelled as normal or abnormal at the time of clinical interpretation by a radiologist from Stanford Hospital - and these labels have been published alongside the images themselves.

The data was published as part of a competition, allowing participants to submit their trained classifiers. These will then be assessed on a test set of a further 556 X-ray images from 207 studies of 206 patients - however this test set has not been published alongside the other images. The authors split the published data into training and validation sets, to help participants train their classifiers on this specific data. They say that they ensured no overlaps in patients between any of the data sets - i.e. that all studies and images for a given patient belong only to one of the training/validation/test sets - and I verified this when analysing Patient ID's in the data.

Table 1: MURA data breakdown

Type	Images	Studies	Patients
Training	36,808	13,457	11,184
Validation	3,197	1,199	783
Test	556	207	206
Total	40,561	14,863	12,173
Total (exc Test)	40,005	14,656	11,967

Alongside the images, the authors published which of seven parts of the upper extremity each study was performed on - Elbow, Finger (though not which finger), Hand, Humerus, Forearm, Shoulder or Wrist. They also published Patient ID to distinguish the different patients, and a study number to distinguish multiple studies of the same patient on the same site (presumably performed at different times, for instance an initial and follow-up study). It's not clear if consecutive study numbers imply the studies were performed consecutively in chronological time, i.e. that study

1 was always performed before study 2 etc. Note that some Patient ID's appeared for multiple upper extremity parts - e.g. they had studies of both, say, the wrist and forearm. However the different studies for different body parts are numbered only within each body part - e.g. a patient may have study 1 for wrist, and studies 1 and 2 for forearm (vs, say, study 1 for wrist and studies 2 and 3 for forearm). So we can't infer that studies 1 for both parts were performed at the same time.

As noted, a label "normal" or "abnormal" was also published for each study - and we have assumed this label applies for all images in the study. However beyond this no further information was released - for instance, date of the study (which would let us identify follow-up studies, or if studies of different parts were performed at the same time and so likely due to the same injury or condition), demographic information about the patients (age, sex etc), the nature of any abnormalities (e.g. fracture vs lesion), the nature of the treatment or the patient's medical condition (e.g. emergency room vs out-patient) etc, all of which would be relevant in a real, clinical setting. The corresponding radiologist's report for each study was also not published, presumably due to privacy concerns. As such, the data available to build a classifier is really just the images themselves, their study's label, and which part of the upper extremity the images are of. Therefore we treat this task as an exercise in image classification, despite the clear relevance of other information (if it were available).

Most of the studies in the data are made up of several X-ray images, each of the same part of the upper extremity, but taken from a different angle. For instance, a frontal view of the hand and a side-profile view. However information about this directionality was not published by the authors.

4.1 Data Collection

As noted by the authors, each study was performed at Stanford Hospital between 2001 and 2012. The labelling was performed manually by a board-certified radiologist, using [1]: "*DICOM*[7] *images presented on at least 3 megapixel PACS*[8] *medical grade display with max luminance 400 cd/m*^{2} *and min luminance 1 cd/m*^{2} *with pixel size of 0.2 and native resolution of* 1500×2000 *pixels*".

[7]Digital Imaging and Communications in Medicine

[8]Picture Archiving and Communication System

4.2 Abnormality Analysis

The authors also manually investigated the radiologist reports for a sample of 100 abnormal studies to get an understanding of the types of abnormalities present in the data. From this they found [1]: "*53 studies were labeled with fractures, 48 with hardware, 35 with degenerative joint diseases, and 29 with other miscellaneous abnormalities, including lesions and subluxations*". Where "hardware" presumably refers to orthopedic hardware such as screws, plates, pins etc, and a "subluxation" refers to the partial dislocation of a joint. The authors did not give a breakdown of which body parts they surveyed, although we assume it was representative, or how this abnormalities compared across body parts. It is also not clear how representative the sample of 100 is of the full data. However the high numbers of hardware is interesting and of relevance to image classification, since such hardware will be apparent in the images and a classifier may learn them to be strong predictive of abnormality. Note that beyond this survey no other informaton about the type or nature of abnormality was given with the data.

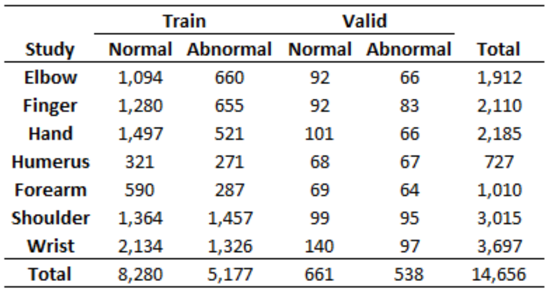
As a side note, the authors do not mention any similar analysis of normal studies. The overall proportion of normal studies in the published dataset is about 60% (see breakdowns below), and it is not clear why so many patients are having X-rays that do not find any abnormality. I would not expect a patient to get an X-ray of their upper extremity without prior cause, and so would have expected a higher overall rate of abnormality. The explanation may come down to the clinical context - for instance, a large proportion of these studies may be follow-up X-rays post-treatment. Or even X-rays taken during treatment where "normal" might mean "treatment progressing normally" even if the original abnormality (say a fracture) is present. However this is entirely speculation on my part.

4.3 Data Breakdowns

Below we present some breakdowns of the data.

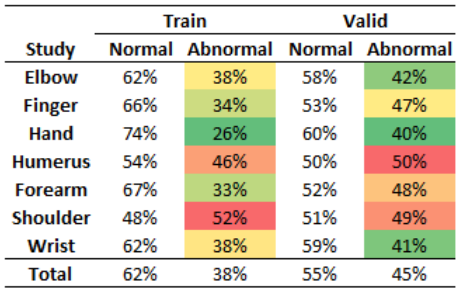
4.3.1 Numbers of Studies

Figure 10a below is a recreation in my own work of the table in [1] showing the same breakdown - which acts as a validation that I am analysing the data correctly. The percentage breakdown shows the proportion of Normal studies is around 60%, as mentioned. The variation in abnormal rate does not look too drastic, although 26% for Hand in the training data perhaps seems low. Shoulder has the highest abnormality rate in the training data, which seems reasonable given that a dislocated shoulder is a commom sports injury. Stanford Hosptial is close to Stanford University, and so may have an unusal number of such injuries (e.g. due to student atheletics, or American college football)



	Train		Valid		
Study	Normal	Abnormal	Normal	Abnormal	Total
Elbow	1,094	660	92	66	1,912
Finger	1,280	655	92	83	2,110
Hand	1,497	521	101	66	2,185
Humerus	321	271	68	67	727
Forearm	590	287	69	64	1,010
Shoulder	1,364	1,457	99	95	3,015
Wrist	2,134	1,326	140	97	3,697
Total	8,280	5,177	661	538	14,656

(a) Numbers of Studies



	Train		Valid	
Study	Normal	Abnormal	Normal	Abnormal
Elbow	62%	38%	58%	42%
Finger	66%	34%	53%	47%
Hand	74%	26%	60%	40%
Humerus	54%	46%	50%	50%
Forearm	67%	33%	52%	48%
Shoulder	48%	52%	51%	49%
Wrist	62%	38%	59%	41%
Total	62%	38%	55%	45%

(b) Percentage Normal/Abnormal

Figure 10: Studies Breakdown by Part

4.3.2 Numbers of Images

Below is an analogous breakdown to Figure 11, except by numbers of images rather than numbers of studies.

	Train		Valid		
Study	Normal	Abnormal	Normal	Abnormal	Total
Elbow	2,925	2,006	235	230	5,396
Finger	3,138	1,968	214	247	5,567
Hand	4,059	1,484	271	189	6,003
Humerus	673	599	148	140	1,560
Forearm	1,164	661	150	151	2,126
Shoulder	4,211	4,168	285	278	8,942
Wrist	5,765	3,987	364	295	10,411
Total	21,935	14,873	1,667	1,530	40,005

(a) Numbers of Images

	Train		Valid	
Study	Normal	Abnormal	Normal	Abnormal
Elbow	59%	41%	51%	49%
Finger	61%	39%	46%	54%
Hand	73%	27%	59%	41%
Humerus	53%	47%	51%	49%
Forearm	64%	36%	50%	50%
Shoulder	50%	50%	51%	49%
Wrist	59%	41%	55%	45%
Total	60%	40%	52%	48%

(b) Percentage Normal/Abnormal

	Train		Valid		
Study	Normal	Abnormal	Normal	Abnormal	Total
Elbow	2.7	3.0	2.6	3.5	2.8
Finger	2.5	3.0	2.3	3.0	2.6
Hand	2.7	2.8	2.7	2.9	2.7
Humerus	2.1	2.2	2.2	2.1	2.1
Forearm	2.0	2.3	2.2	2.4	2.1
Shoulder	3.1	2.9	2.9	2.9	3.0
Wrist	2.7	3.0	2.6	3.0	2.8
Total	2.6	2.9	2.5	2.8	2.7

(c) Average Number of Images Per Study

Figure 11: Images Breakdown by Part

The above shows no strong bias in numbers of images per study, however there are slightly more on average for abnormal studies than normal - which stands to reason. This is reflected in the percentage abnormal by image being slightly higher than the by study equivalent. Humerus and forearm have noticable fewer average images per study than the other body parts - most likely because they do not need to be viewed from as many angles to make an assessment.

4.3.3 Studies of Multiple Body Parts

Below we show the numbers of patients by how many body parts they have a study for. This reason for examining this is that it implies a correlation in the data - an abnormal study of the hand, say, might make a study of the wrist more likely to be abnormal if they are both the result of the same injury. While our method image classification doesn't really require independant data, this correlation is still worth noting.

Figure 12: Patients by Number of Body Parts

Number of Sites	Train	Valid	% Train	% Valid
1	9717	535	87%	68%
2	1231	174	11%	22%
3	194	63	2%	8%
4	37	9	0%	1%
5	5	2	0%	0%
Total	11,184	783	100%	100%

So this says that 11% of patients in the training data had at least one study each on two parts of the upper extremity. It is interesting that the validation data is skewed towards patients with studies of multiple parts, although this is probably a random effect due to small numbers, rather than being intentional by the MURA authors. Below we break down the co-occurence of pairs of body parts in the training data.

	Elbow	Finger	Hand	Humerus	Forearm	Shoulder	Wrist
Elbow	-	108	113	31	56	142	166
Finger	108	-	108	40	53	157	158
Hand	113	108	-	30	51	150	196
Humerus	31	40	30	-	16	55	58
Forearm	56	53	51	16	-	70	78
Shoulder	142	157	150	55	70	-	249
Wrist	166	158	196	58	78	249	-
	616	624	648	230	324	823	905

(a) Co-occurence by numbers of patients

	Elbow	Finger	Hand	Humerus	Forearm	Shoulder	Wrist
Elbow	-	5%	5%	1%	3%	7%	8%
Finger	5%	-	5%	2%	3%	8%	8%
Hand	5%	5%	-	1%	2%	7%	9%
Humerus	1%	2%	1%	-	1%	3%	3%
Forearm	3%	3%	2%	1%	-	3%	4%
Shoulder	7%	8%	7%	3%	3%	-	12%
Wrist	8%	8%	9%	3%	4%	12%	-

(b) Percentage Co-occurence

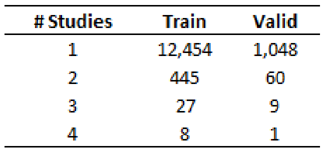
Figure 13: Breakdown of body part co-occurence

So for instance 113 patients had studies of both their hand and elbow. Note this table double-counts patients with studies of more than two parts (e.g. hand/wrist/shoulder would be counted in all pairs hand/wrist, hand/shoulder, shoulder/wrist) so totals won't match Figure 12 above. However producing the full multi-way occurence breakdown is cumbersome, and not very relevant since only a small proportion of patients had studies of more than two parts. The breakdown above shows that shoulder and wrist are the most common pair of parts, which seems plausible assuming they were studied together in the same period of treatment. Humerus and forearm have only small numbers of co-occurences, suggesting they tend to be studied in isolation.

4.3.4 Consecutive Studies

Now we examine the cases of the same patient having multiple studies on the same body part. The distribution of numbers of studies per patient's body part is shown below:

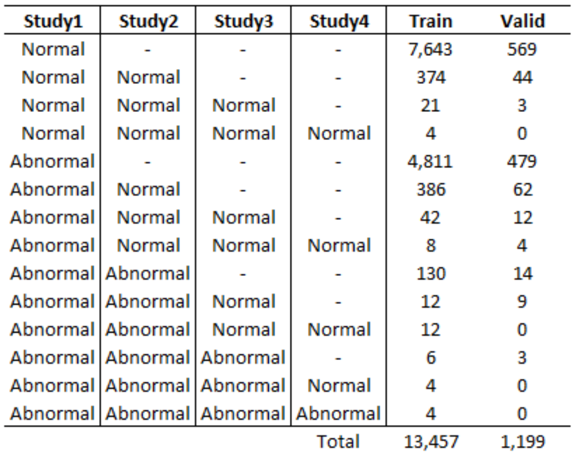
Figure 14: Numbers of studies per patient's body part



# Studies	Train	Valid
1	12,454	1,048
2	445	60
3	27	9
4	8	1

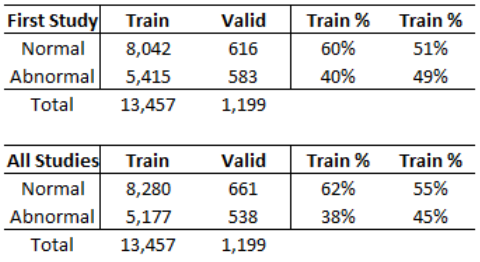
So there were 8 patients in the training data with 4 studies of the same body part (ignoring studies of other parts). We speculated above that multiple studies likely related to follow-up reviews after initial treatment. This *appears* to suggested by the data, since analysing the study results over consecutive studies (which were named study1, study2 etc in the raw data) shows only progressions from abnormal to normal, and none at all from normal to abnormal anywhere in the data:

Figure 15: Results of Consecutive Studies



Study1	Study2	Study3	Study4	Train	Valid
Normal	-	-	-	7,643	569
Normal	Normal	-	-	374	44
Normal	Normal	Normal	-	21	3
Normal	Normal	Normal	Normal	4	0
Abnormal	-	-	-	4,811	479
Abnormal	Normal	-	-	386	62
Abnormal	Normal	Normal	-	42	12
Abnormal	Normal	Normal	Normal	8	4
Abnormal	Abnormal	-	-	130	14
Abnormal	Abnormal	Normal	-	12	9
Abnormal	Abnormal	Normal	Normal	12	0
Abnormal	Abnormal	Abnormal	-	6	3
Abnormal	Abnormal	Abnormal	Normal	4	0
Abnormal	Abnormal	Abnormal	Abnormal	4	0
			Total	13,457	1,199

This led to the thought that the rate of normal studies in the whole data is being inflated by the follow-up studies. Examining the results of the first study only shows this is true, although the effect is too small to be relevant as the numbers of these apparent follow-up studies are small. The tables below show the first study has a slightly higher rate of abnormality than the rate of abnormality across the whole data.



First Study	Train	Valid	Train %	Train %
Normal	8,042	616	60%	51%
Abnormal	5,415	583	40%	49%
Total	13,457	1,199		

All Studies	Train	Valid	Train %	Train %
Normal	8,280	661	62%	55%
Abnormal	5,177	538	38%	45%
Total	13,457	1,199		

Figure 16: Breakdown of study outcomes

4.3.5 Alternative Data

Prior to deciding to work with the MURA data, we also considered the ChestX-ray14 data [17] - another set of radiology images, of the chest, published by the National Institutes of Health (NIH). However this data was more complicated to work with since instead of being labelled normal or abnormal, each image was labelled with up to 14 (not mutually exclusive) pathology labels - such as "Pneumomnia", "Nodule", "Effusion". The task would therefore have been to output 14 different binary predictions for each image as to the presence of each pathology label.

Further, these labels were not directly given by a radiologist, but were instead generated by text-mining of the original radiologist's report associated with each image. This text-mining included attempts to identify and account for negations (such as "no effusion"), uncertainty, handle synonyms etc. As such, the labels seemed to be less relaible, for a more complicated task, than for MURA. This was corroborated in a commentary by Luke Oakden-Rayner, a radiologist unaffiliated with NIH, who manually reviewed several hundred of the images and their associated labels [18]: *"Compared to human visual assessment, the labels in the ChestXray14 dataset are inaccurate, unclear, and often describe medically unimportant findings."*. While his word is not necessarily final, we opted to work with the MURA data instead for simplicity.

5 Model Development

5.1 Working Environment

The working environment for this project is primarily a Google Cloud virtual machine, which I have rented (using free credit) in order to get access to GPU, namely an NVIDIA Tesla P100. This is needed because training large a neural network would be infeasible without a GPU, which are optimised to perform the types of calculations required (e.g. convolution operations). The GPU will also handle parallelisation, e.g. of each image in an SGD batch, to further improve computation time. The machine I used for training had 2 P100 GPU's and 16 7.5GB Intel Sandy Bridge CPU's, and ran the Ubuntu 17.10 operating system.

Beyond that, I used the Python library Keras to build, train and test my neural networks. This is a subset of the Tensorflow (version 1.9) library, which also handles assignment of computation to the GPU. As such, all my scripts are written in Python (version 3.6.5). I installed this set-up myself on the virtual machine, including Tensorflow dependencies such as NVIDIA's CUDA toolkit (version 9.2) and cuDNN neural network library (version 7.2) which enable it to work with the GPU.

5.2 Data Preprocessing

The images in raw data come in a variety of resolutions and aspect ratios. However for the purposes of image classification with convolutional networks, all images in the input must be the same size - as otherwise the sizes of the final representation from the convolutional base would differ, and hence the number of neurons after flattening it.

Therefore we resized the images to common height and width using an in-built Tensorflow function. This uses the method called *bilinear interpolation*. Following wikipedia [19] this works by first linearly interpolating pixel values in the x-direction, and then performing a second interpolation on the results, in the y-direction. Note that our images are greyscale, and so have only one channel - hence interpolation is only required in the spatial dimension.

In addition, prior to inputting into our models, the pixel values in each image were normalised to have mean 0 and standard deviation 1. This was done for each image individually, rather than across the whole dataset. Not doing this resulted in networks failing to train, likely because large pixel values (the original png encodings ranged between -128 and +128) lead to large activations, and hence large gradients and so too-large weight updates.

5.3 Model

In building a model to submit to the MURA competition, I took the approach of first constructing a relatively simple baseline model by myself, then moving on to more sophisticated pre-built architectures designed by others which should outperform what I can come up with in the time allowed for this project.

5.3.1 Simple Baseline

My simple baseline model had the following structure:

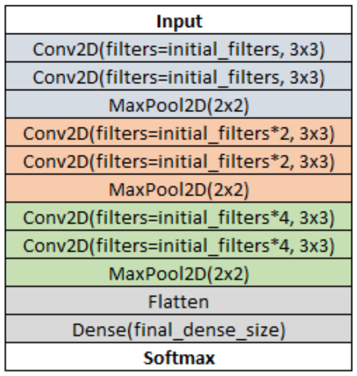

Input
Conv2D(filters=initial_filters, 3x3)
Conv2D(filters=initial_filters, 3x3)
MaxPool2D(2x2)
Conv2D(filters=initial_filters*2, 3x3)
Conv2D(filters=initial_filters*2, 3x3)
MaxPool2D(2x2)
Conv2D(filters=initial_filters*4, 3x3)
Conv2D(filters=initial_filters*4, 3x3)
MaxPool2D(2x2)
Flatten
Dense(final_dense_size)
Softmax

Figure 17: Simple Baseline Structure

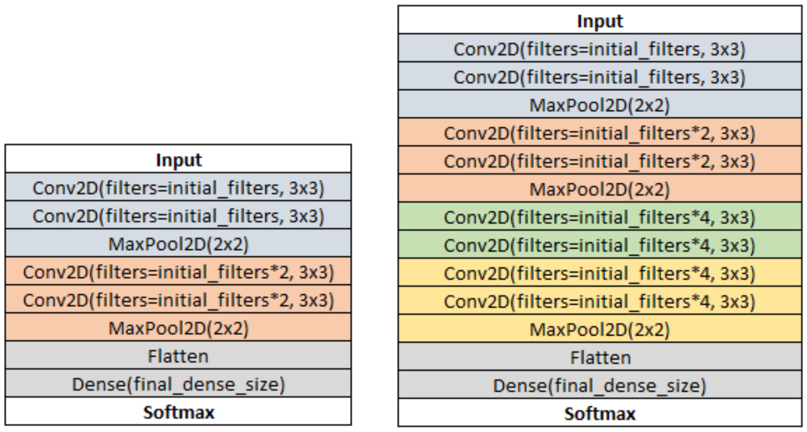
This architecture was inspired by VGG - I have consecutive 3×3 convolutional layers, followed by a 2×2 pooling layers to reduce the spatial dimension. The number of filters was doubled after reducing the spatial dimension. There are 3 of these ”blocks”, followed by a flatten layer and then a fully-connected and softmax layers to produce the final output.

There is flexibility in choosing the number of filters in the initial convolutional layer, and in the size of the final fully-connected layer. I experimented with 3 versions of this simple architecture:

- **Small:** 32 initial filters, 100 fully-connected neurons.
- **Medium:** 128 initial filters, 100 fully-connected neurons.

- **Big:** 256 initial filters, 256 fully-connected neurons.

The idea being that the bigger versions have more scope to capture more features of the images, and so should perform better than smaller versions. I also experimented with shallower and deeper versions of the baseline architecure:


Input
Conv2D(filters=initial_filters, 3x3)
Conv2D(filters=initial_filters, 3x3)
MaxPool2D(2x2)
Conv2D(filters=initial_filters*2, 3x3)
Conv2D(filters=initial_filters*2, 3x3)
MaxPool2D(2x2)
Flatten
Dense(final_dense_size)
Softmax
Input
Conv2D(filters=initial_filters, 3x3)
Conv2D(filters=initial_filters, 3x3)
MaxPool2D(2x2)
Conv2D(filters=initial_filters*2, 3x3)
Conv2D(filters=initial_filters*2, 3x3)
MaxPool2D(2x2)
Conv2D(filters=initial_filters*4, 3x3)
Conv2D(filters=initial_filters*4, 3x3)
Conv2D(filters=initial_filters*4, 3x3)
Conv2D(filters=initial_filters*4, 3x3)
MaxPool2D(2x2)
Flatten
Dense(final_dense_size)
Softmax

(a) Shallower Baseline (b) Deeper Baseline

Figure 18: Alternative Baseline Models

Note that in the deeper baseline, the pooling layer was omitted from the third "block" of convolutional layers, so as not to reduce the spatial dimension too far. As with the original baseline architecure, we can adjust the number of initial filters and size of the fully-connected layer.

The architecture was chosen to be simple so as to allow training from-scratch in a reasonable length of time - usually minutes rather than hours for more complicated architectures.

5.3.2 Pre-Built Architectures

The Keras library comes with a number of pre-built architectures, that can either be used with the weights as trained on the ImageNet data or weights trained from scratch. I opted for the latter as it seems unlikely that the features detected by the ImageNet weights would transfer well to the specialised case of X-ray images, since ImageNet covers everyday objects such as cats, dogs, cars etc. That is, while the architecture - the sequence of layers, numbers of filters in each layer and so on - remains the same, the associated weights and biases would differ.

Note that the architecture here refers only to the convolutional base, rather than the subsequent fully-connected and softmax head - since the pre-built heads are specific to ImageNet, and so output 1000 categories vs the 2 categories I need. So I replaced the ImageNet heads with heads of my own - a fully-connected layer (of chosen size) and a softmax layer with 2 neurons. The weights in these layers would of course also be trained from scratch, along with those in the pre-built convolutional base.

Specifically, I chose the following pre-built architectures:

- VGG16 - the 2014 ILSVRC winner
- Inception-ResNet-v2 (GoogLeNet-v4) - the 2016 ILSVRC winner.

5.3.3 Training

Unless stated otherwise, I trained all models using RMSProp with Keras' default parameters - learning rate 0.001, exponential smoothing parameter 0.9 and epsilon 1E-7. Training was ran for a pre-defined numer of epochs, usually 25, and the learning rate was left unchanged throughout unless stated otherwise. I typically used a batch size of 512 when training my simple baseline models. However for the larger pre-built models, large batch sizes resulted in out-of-memory errors so I reduced the batch size for these to 64.

5.4 Predictions

The models described above work on individual images - producing a predicted probability of abnormality for each image separately. However the original data is organised into studies consisting of multiple images, and the original normal/abnormal label was applied to the study, rather than to each image within. As mentioned, we have assumed during training that this label applies to all images in the study, and this seems reasonable. However when analysing the performance of our model after training, we should be more interested in its performance at the study level.

So we need to go from the model's image-wise predictions to study-wise predictions. To do this, we follow the MURA authors [1] who take the arithmetic mean of the predicted probabilities of abnormality across all images in the study. If this average exceeds 50%, then their study-wise prediction is "abnormal" and "normal" otherwise. After calculating study-wise predictions, we can then calculate other performance metrics, such as Cohen's Kappa, study-wise. Indeed, study-wise Cohen's Kappa assessed on the test set is the performance metric used in the competition held by the MURA authors.

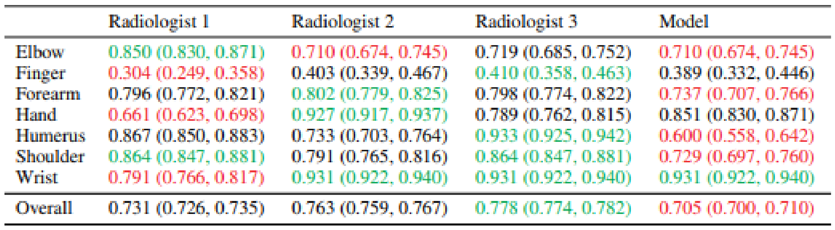
6 Results

6.1 Benchmark - MURA Authors

First we present the results given by the authors of the MURA paper [1]. They produced results using an ensemble of their own models - based on a 169-layer DenseNet architecture, trained end-to-end but with weights initialised from a model pre-trained on ImageNet.

They also created gold standard ground-truth labels for their test set by collecting normal/abnormal labels for each study from 6 board-certified radiologists, and randomly picking 3 of them. The gold standard was then based on the majority vote of these 3 radiologists. The other 3 radiologists were used to assess radiologists' performance in the task, against this gold standard. Note that we do not have access to this test data (either the images or the gold standard labels) - and so cannot do a like-for-like comparison with our own models. However these results still serve as a useful benchmark as to cutting-edge performance.

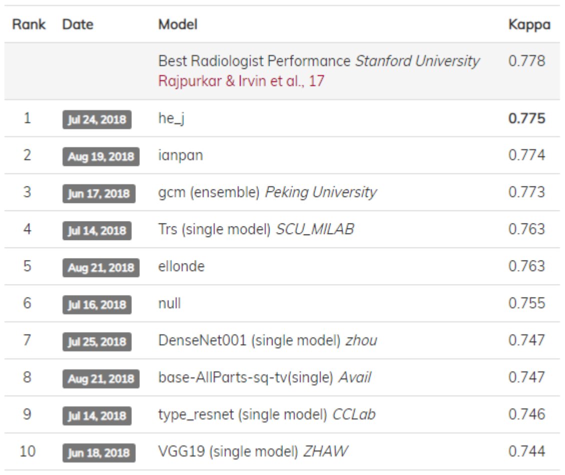
The performance, measured by Cohen's kappa, of the 3 radiologists and the authors' model ensemble on the test data is shown below (from [1]). For each study type, the best performance is highlighted in green and the worst performance in red. The figures in brackets are 95% confidence intervals based on the standard error of Cohen's kappa.



	Radiologist 1	Radiologist 2	Radiologist 3	Model
Elbow	0.850 (0.830, 0.871)	0.710 (0.674, 0.745)	0.719 (0.685, 0.752)	0.710 (0.674, 0.745)
Finger	0.304 (0.249, 0.358)	0.403 (0.339, 0.467)	0.410 (0.358, 0.463)	0.389 (0.332, 0.446)
Forearm	0.796 (0.772, 0.821)	0.802 (0.779, 0.825)	0.798 (0.774, 0.822)	0.737 (0.707, 0.766)
Hand	0.661 (0.623, 0.698)	0.927 (0.917, 0.937)	0.789 (0.762, 0.815)	0.851 (0.830, 0.871)
Humerus	0.867 (0.850, 0.883)	0.733 (0.703, 0.764)	0.933 (0.925, 0.942)	0.600 (0.558, 0.642)
Shoulder	0.864 (0.847, 0.881)	0.791 (0.765, 0.816)	0.864 (0.847, 0.881)	0.729 (0.697, 0.760)
Wrist	0.791 (0.766, 0.817)	0.931 (0.922, 0.940)	0.931 (0.922, 0.940)	0.931 (0.922, 0.940)
Overall	0.731 (0.726, 0.735)	0.763 (0.759, 0.767)	0.778 (0.774, 0.782)	0.705 (0.700, 0.710)

Figure 19: MURA Authors Results - Test Data

Also shown below is the top-10 of the leaderboard (from [20]) for MURA competition entries, showing that none of the submissions have yet surpassed the best radiologist's overall performance. Although they are getting impressively close, and out-perform the two other radiologists.



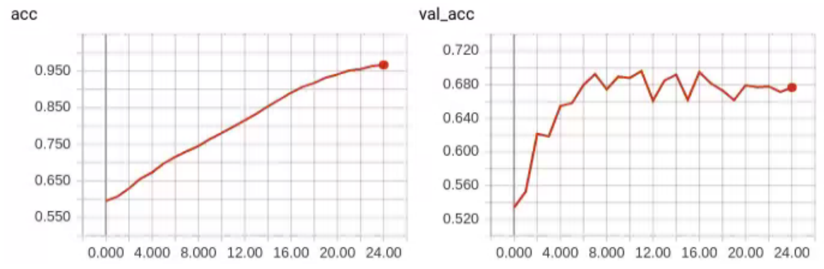
Rank	Date	Model	Kappa
		Best Radiologist Performance *Stanford University* Rajpurkar & Irvin et al., 17	0.778
1	Jul 24, 2018	he_j	**0.775**
2	Aug 19, 2018	ianpan	0.774
3	Jun 17, 2018	gcm (ensemble) *Peking University*	0.773
4	Jul 14, 2018	Trs (single model) *SCU_MILAB*	0.763
5	Aug 21, 2018	ellonde	0.763
6	Jul 16, 2018	null	0.755
7	Jul 25, 2018	DenseNet001 (single model) *zhou*	0.747
8	Aug 21, 2018	base-AllParts-sq-tv(single) *Avail*	0.747
9	Jul 14, 2018	type_resnet (single model) *CCLab*	0.746
10	Jun 18, 2018	VGG19 (single model) *ZHAW*	0.744

Figure 20: MURA Leaderboard Top 10 - as of 30th August 2018

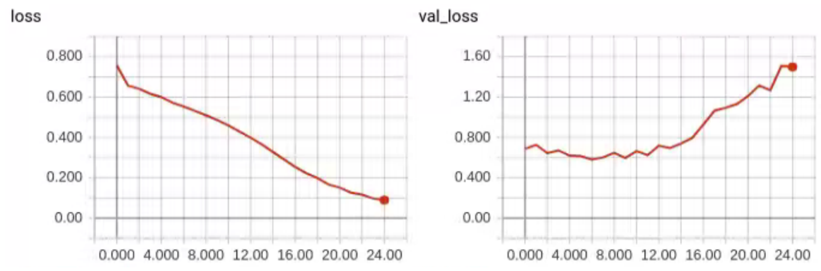
6.2 Baseline Performance

6.2.1 Vanilla Model

Below we show the (image-wise) performance over 25 epochs of training the small version of the baseline model (as described in Section 5.3.1), using an image size of 48×48. We begin with such a small image size for computational convenience (faster training, chronologically), and to examine how the performance changes as we move to larger images sizes. We call this our "vanilla model", and use it as a reference point for all subsequent experiments and models.



(a) Accuracy - Training and Validation Sets



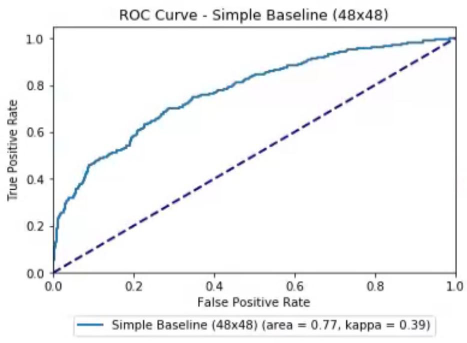
(b) Loss - Training and Validation Sets

Figure 21: Small Baseline Model - Image Size 48×48

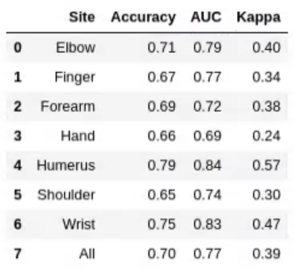
Where the x-axis on these charts is the epoch, and the y-axis is either accuracy rate or cross-entropy loss[9]. This appears to show overfitting - with the training accuracy approaching 1, while the validation accuracy has plateaued around 0.68 and the validation loss increasing at the later epochs.

Below we present the study-wise performance on this model against the validation data.

[9]Unfortunately TensorBoard, from where these graphs are taken, does not allow axes to be annotated, or a legend to be displayed with the chart. Hence these are missing.



(a) ROC Curve



	Site	Accuracy	AUC	Kappa
0	Elbow	0.71	0.79	0.40
1	Finger	0.67	0.77	0.34
2	Forearm	0.69	0.72	0.38
3	Hand	0.66	0.69	0.24
4	Humerus	0.79	0.84	0.57
5	Shoulder	0.65	0.74	0.30
6	Wrist	0.75	0.83	0.47
7	All	0.70	0.77	0.39

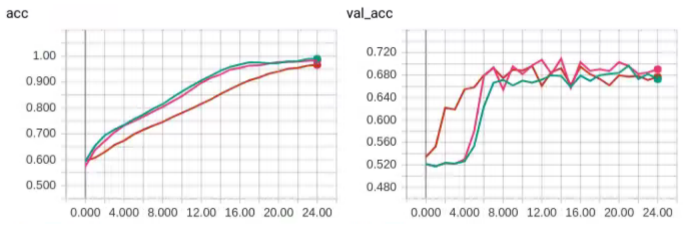
(b) Performance by Site

Figure 22: Studywise Performance - Vanilla Model

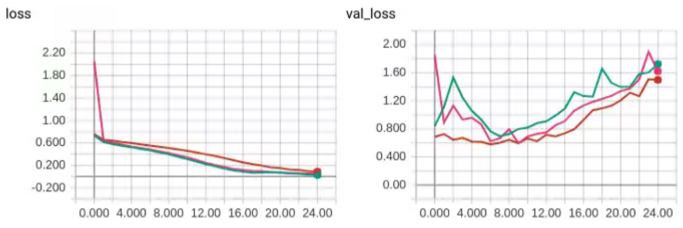
While the overall kappa of 0.39 might not compare favourably to the MURA benchmark of 0.778, this is a simple model trained on very small data so we do not expect cutting-edge performance.

6.2.2 Regularisation

Since the vanilla model appeared to be overfitting, we tried to use some regularisation techniques to prevent this.



(a) Accuracy - Training and Validation Sets

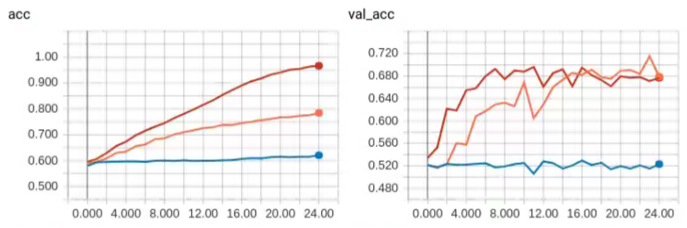


(b) Loss - Training and Validation Sets

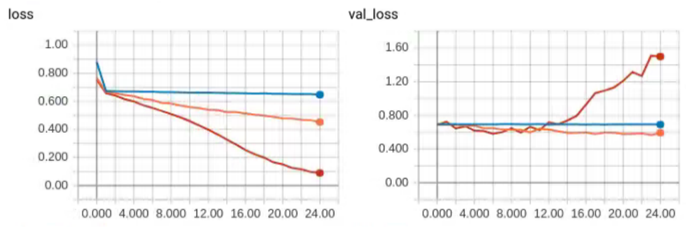
Figure 23: Vanilla Model with Batch Normalisation

The green and pink lines show the vanilla model with batch normalisation applied to all convolutional layers, while the brown line is the vanilla model from above. The green line uses the Adam optimiser while pink uses RMSProp. Unfortunately, while their training performance improved more quickly than the vanilla (albeit plateauing at a roughly similar level), the validation performance does not show any significant regularisation effect. Indeed, their validation loss is higher than the vanilla model's at almost all epochs.

So we decided to try applying dropout to all layers (except the input) instead:



(a) Accuracy - Training and Validation Sets



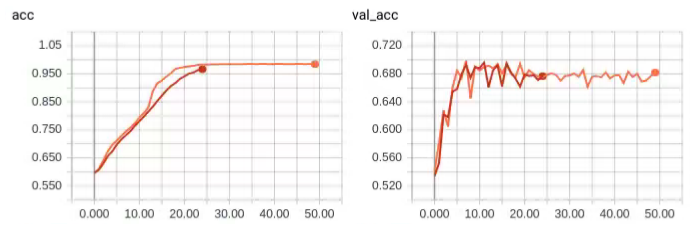
(b) Loss - Training and Validation Sets

Figure 24: Vanilla Model with Dropout

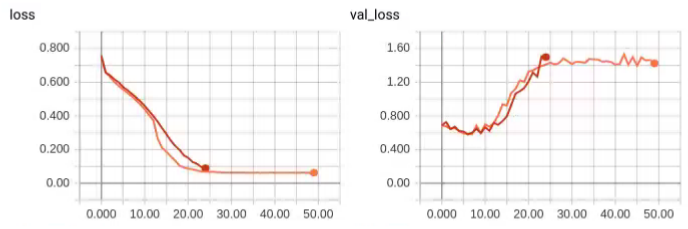
The orange line shows dropout with probability 30% of removing a neuron, while the blue line shows 70%. Clearly the blue line removes too many neurons, since it demonstrates training failure - the resulting network is likely far too small. The orange line shows some regularising effect - it prevents the validation loss from increasing. Although it plateaus at a similar validation accuracy rate as the vanilla model.

6.2.3 Learning Rate Decay

Next we decided to try the technique of learning rate decay - where the learning rate was reduced by a factor of 5 if the validation loss has not improved for more than 5 epochs. This could help prevent overfitting, since it slows down learning and prevents the weights from fitting too specifically to the training data. Note that we ran this experiment for 50 epochs rather than 25, to allow more time for the decay to take effect.



(a) Accuracy - Training and Validation Sets



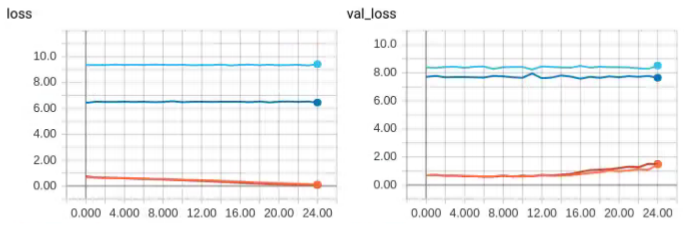
(b) Loss - Training and Validation Sets

Figure 25: Vanilla Model with Learning Rate Decay

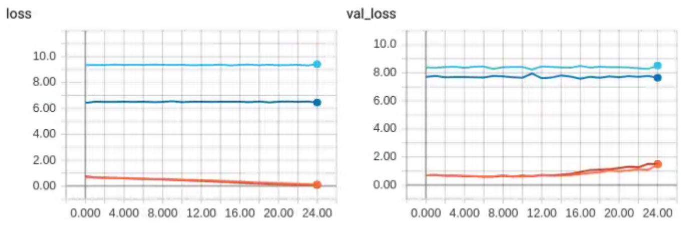
The orange line shows the results with learning rate decay. Unfortunately this did not have the desired regularising effect - the model seems to settle at the same local minimum as the vanilla model. The extra, slower-learning, epochs showing only that the loss and accuracy plateaued (both training and validation) shortly after the 25 epochs of the vanilla model. Interestingly, training seems to accelerate around the 12^{th} epoch, presumably when the learning rate was decreased. This may be because the vanilla model was oscillating around the minimum, due to the relatively high learning rate. Whereas decreasing the learning rate allowed the local minimum to be approached more directly and hence improved the training performance more quickly. But since both models appear to be approaching the same local minimum, this doesn't have much effect on the validation performance - indeed, the validation loss gets worse for the learning rate decay model learning when this acceleration effect kicks in.

6.2.4 Weight Initialisation

Since the two models above appeared to be approaching the same local minumum, we decided to try alternative weight initialisation schemes, in the hope that this would lead to an alternative, better, local minimum.



(a) Accuracy - Training and Validation Sets



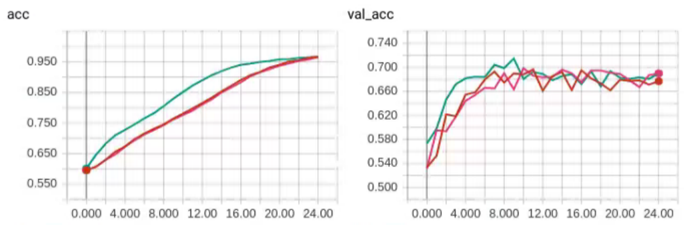
(b) Loss - Training and Validation Sets

Figure 26: Vanilla Model with Alternative Weight Initialisation

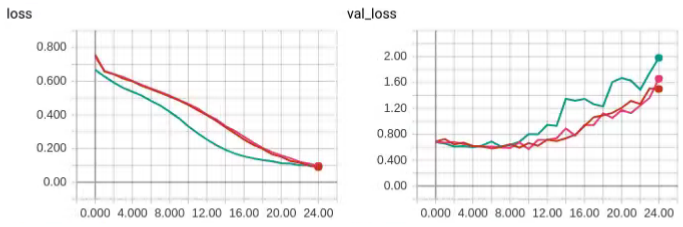
Note the vanilla model used Glorot Uniform initialisation. The orange line shows Glorot Normal, while the light blue and dark blue are random normal (mean 0, standard deviation 0.05) and He Normal respectively. While Glorot Normal does give a slightly lower validation loss in the early epochs, by the later epochs it appears to approach the same local minimum as the vanilla model. Random and He Normal result in training failure - perhaps because the scale of their weights was too large.

6.2.5 Vanilla Model Reruns

At this point we decided to rerun the vanilla model in its original form, with a couple of tweaks to see if we could get it to shift to a different local minima. The first rerun changed the random seed used when generating the data - so that different data appeared in each batch than the original vanilla model. The second rerun reduced the batch size from 512 to 128.



(a) Accuracy - Training and Validation Sets



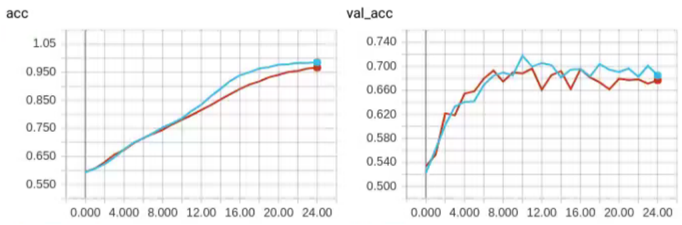
(b) Loss - Training and Validation Sets

Figure 27: Vanilla Model Reruns

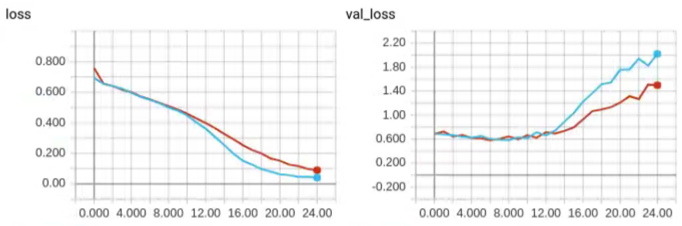
The brown line, as always, shows the original vanilla model. The pink line is the first rerun and the green line is the second. Using a different data seed appears to have a negligable effect - the charts are plotted by epoch, which consitutes a full pass through the data, so the order of appearance of each data item seems not to effect the epoch-level results much. Reducing the batch size seems to result in faster training, presumably due to having more weight updates per epoch. However it does not seem to lead to a different local minimum. It does have a considerably larger validation loss, but the original vanilla model would likely approach a similar level given enough training time.

6.2.6 Alternative Architectures

We now tried altering the vanilla model architecture to see if this helped imporve performance. First we tried the medium version of the simple architectures - using 128 filters in the initial convolational layers, comapared to 32 in the vanilla model.



(a) Accuracy - Training and Validation Sets

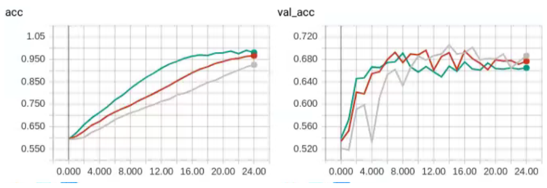


(b) Loss - Training and Validation Sets

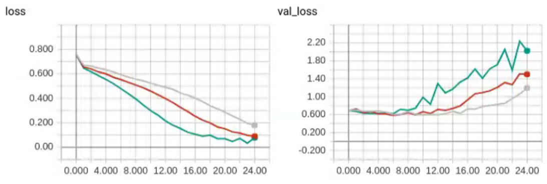
Figure 28: Vanilla Model (32 vs 128 initial filters)

Having more filters per layers, the medium size model has greater scope to overfit - and it appears to do so. There is a curious acceleration in performance from around the 10th epoch. It's not clear what causes this, since learning rate decay was not used here.

We also tried using the shallower and deeper versions of the vanilla architecture, again with 32 initial filters.



(a) Accuracy - Training and Validation Sets



(b) Loss - Training and Validation Sets

Figure 29: Vanilla Model - Shallower and Deeper Versions

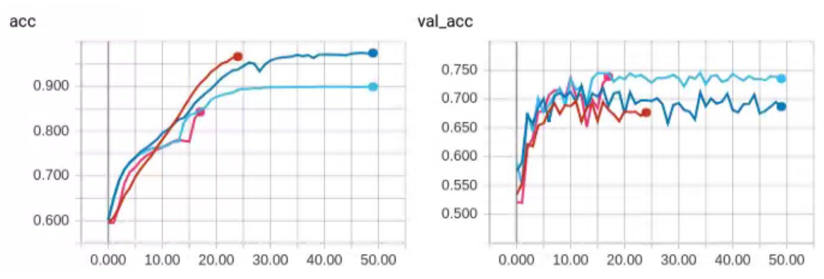
The green line shows the shallower model, and the silver line the deeper one. The deeper model appears to be training more slowly - it's training accuracy and loss have not yet plateaued after 25 epochs. However we decided not to train it for longer, since the validation loss shows that it is already overfitting in any case. The shallower model trains much quicker, but this just results in greater overfitting.

6.3 Pre-Built Models

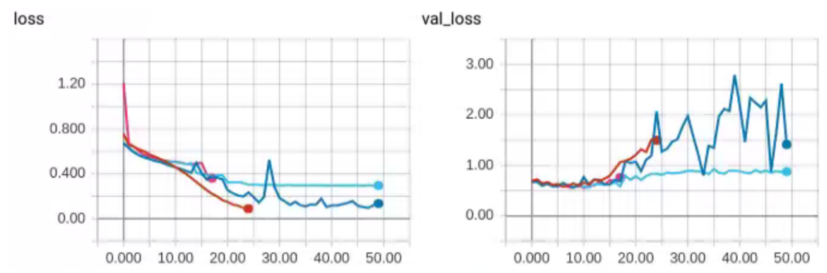
After trying a number of experiments with the simple baseline architecture, as described above, we decided to compare the vanilla model's performance against pre-built architectures. As mentioned, these were trained from scratch, end-to-end, rather than use the pre-trained ImageNet weights.

6.3.1 VGG16

We trained this architecture using RMSProp with a learning rate of 0.0002, since the default learning rate 0.001 was too high, resulting in training failure. We also trained it on a variety of image sizes.



(a) Accuracy - Training and Validation Sets



(b) Loss - Training and Validation Sets

Figure 30: VGG16

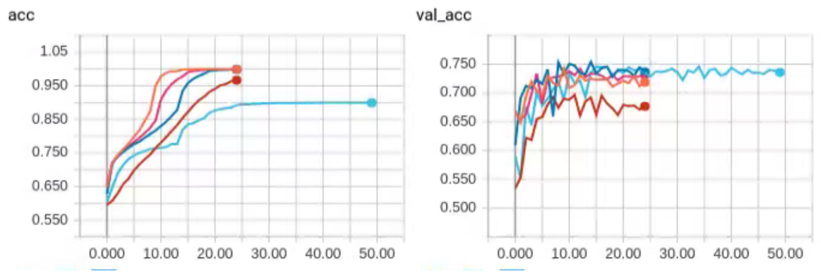
The brown line again shows the original vanilla model, trained as mentioned on 48×48 images. The dark blue line shows VGG16 also trained on 48×48 images, while light blue is on 100×100. The pink line is trained on 300×300 images. The latter two also used learning rate decay as above - reducing by a factor of 5 when the validation loss does not improve for more than 5 epochs.

VGG16 trained on 48×48 images shows very large and erratic jumps in validation loss at the later epochs - likely because by this point the learning rate is far too high (since it was not decayed). So it may be oscillating around a local minimum, but its weight updates are too large to get closer towards it. The overfitting is also likely exacerbated by the very small image size, which may not be appropriate for such a large model.

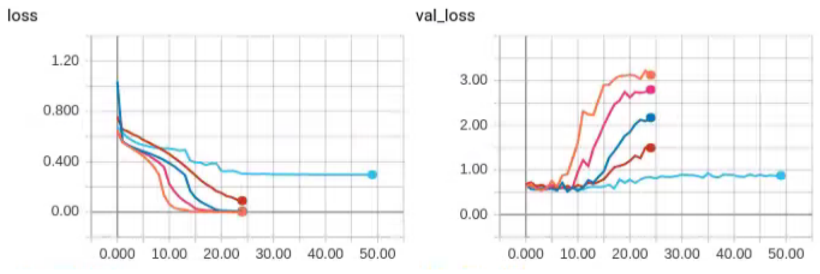
Meanwhile the model trained on the 100×100 (light blue line) likely avoids this issue, and it has learning rate decay to help it progress smoothly towards its local minimum. It has a noticable acceleration in training accuracy after about the 15^{th} epoch, likely when the learning rate was reduced. This model also significantly out-performs the vanilla model in terms of validation loss and accuracy, which might be expected since it is a more complicated architecture.

6.3.2 Inception-ResNet-v2

We also tried the Inception-ResNet-v2 architecture, again trained from scratch using RMSProp with learning rate 0.0002 and with learning weight decay, and again on variety of image sizes.



(a) Accuracy - Training and Validation Sets



(b) Loss - Training and Validation Sets

Figure 31: Inception-ResNet-v2

The brown line is the vanilla model, and the light blue line is the same VGG16 model trained on 100 × 100 images as above. The orange line is Inception-ResNet-v2 trained on 139 × 139 images, the smallest dimensions allowed. The pink line is the same but with L2 weight regularisation applied to the output of the convolutional block and the subsequent fully-connected and softmax layers. The dark blue line is Inception-Resnet-v2 trained on 300 × 300 images, with no weight regularisation. Note we tried to train on 300 × 300 images with regularisation but this led to several training failures and we did not have the time to try to fix this.

Strikingly, these Inception models are able to achieve 100% accuracy on the training data, although of course this does not translate as well to the validation accuracy, where they have comparable performance to the VGG16 model. And in terms of validation loss, they appear to perform worse than even the vanilla model. However the regularising effect of the L2 model is evident.

6.4 MURA Competition Submission

When deciding which of the above collection of models and experiments to submit to the MURA competition, we analysed the study-wise performance on the validation data. This is more comparable to the competition's performance metric - study-wise performance on their test data - than the image-wise training curves shown above. The resulting ROC curves, and AUC and kappa statistics are shown for a selection of models below:

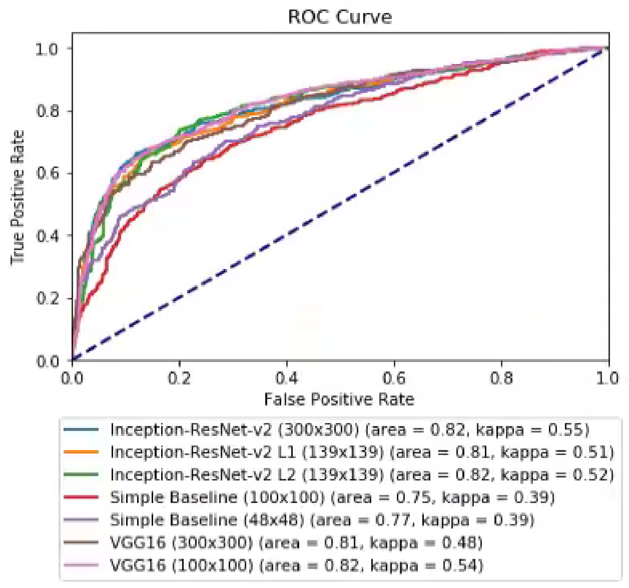
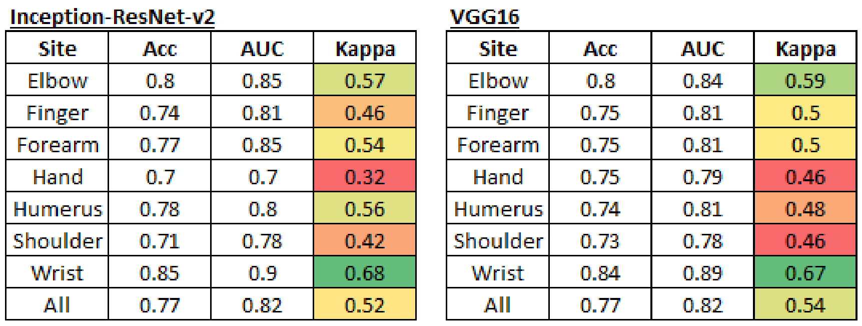


Figure 32: ROC Curves - Model Selection

The best performing of these models, per Cohen's kappa, were the Inception-ResNet-v2 architectures, and VGG16 trained on 100×100. In the end we decided to submit the Inception-ResNet-v2 trained on 139×139 images with L2 regularisation. We preferred it over the version trained on 300×300 since this submission would be quicker to run computationally. We preferred it to the version with L1 regularisation, since it had the slight edge of overall performance We preferred it over the VGG model, even though VGG appears to out-perform it, since we felt there was a risk of ”overfitting” to the validation set - i.e. VGG performing better on the validation set won't necessarily translate to better performance on the test set. And Inception-ResNet is a more sophisticated, modern architecure than the relatively straightforward VGG design, and so we have hope that it should perform better on the unseen data. In any case, the difference in performance between these two models is relatively small so it's something of a toss-up between the two.

Breaking down the study-wise results of these two models shows they are comparable by site - each strong where the other is strong, and weak where the other is weak.



Inception-ResNet-v2

Site	Acc	AUC	Kappa
Elbow	0.8	0.85	0.57
Finger	0.74	0.81	0.46
Forearm	0.77	0.85	0.54
Hand	0.7	0.7	0.32
Humerus	0.78	0.8	0.56
Shoulder	0.71	0.78	0.42
Wrist	0.85	0.9	0.68
All	0.77	0.82	0.52

VGG16

Site	Acc	AUC	Kappa
Elbow	0.8	0.84	0.59
Finger	0.75	0.81	0.5
Forearm	0.75	0.81	0.5
Hand	0.75	0.79	0.46
Humerus	0.74	0.81	0.48
Shoulder	0.73	0.78	0.46
Wrist	0.84	0.89	0.67
All	0.77	0.82	0.54

Figure 33: Results by Site - VGG16 & Inception-ResNet-v2

The largest gap in performance was for Hand, where VGG comfortably beats Inception. However VGG is not the winner across the board - being beaten for Forearm, Humerus and Wrist.

7 Professional and Ethical Issues

There are a number of ethical issues raised when considering the use of Deep Learning within medicine. This is different to other forms of medical technology in that the purpose would be to make *decisions* regarding a patient, in lieu of or alongside a human doctor. Of course, if these decisions are ultimately based on data, as learned by the model during training and perhaps while in use afterwards, then it might be argued that these decisions should be superior to those made by human doctors. Human doctors will be subject (among other things) to their own internal biases, external pressures (e.g. overwork, or fashions in medical diagnosis), their state of mind (e.g. tired at the end of a long shift) etc. Nevertheless there is likely to be a psychological barrier for patients to fully accept a diagnosis reported by computer over that given by a human whom they trust. As such, the use of these machine learning tools is likely to be a behind-the-scenes aspect of medicine, used by human doctors to help inform their decision but unlikely to replace them.

A more likely scenario is that Deep Learning could be used to help triage and prioritise a doctor's work. For instance, a radiology system working with analogous data to this project could leave a nurse to deal with patients with a very low probability abnormality, while sending high probability and more ambiguous cases to the senior radiologist. This could help ease the burden on the radiologist's time as he would not have to be involved with the obvious clear-cut cases. If the deep learning system tried to make some estimate of the nature and severity of the abnormality, it could help the radiologist focus on the most severe cases who need treatment more urgently.

Further, by acting as a "second opinion" of sorts, such a system could help reduce medical errors. For instance, spotting some form of abnormality that the radiologist might have missed - even if the system isn't enitrely certain that it is indeed a true abnormality. Merely highlighting the relevant part of the image could motivate the radiologist to investigate further, and so help reduce his false negative rate. Although this has a drawback, if the system highlights a lot of false positives of this nature, then it could end up wasting a lot of the radiologist's time over nothing. That said, the degree of conservatism with which the system operates could ultimately be tweaked and tuned, and changed in response to conditions or the nature of the task, whereas for a human doctor it is entirely subjective and will depend on their personality.

Clearly any such system would need to be very thoroughly tested and validated against a range of many and varied datasets before being used in the real world. Any bias in the training data will be reflected in its subseuqent performance. For instance, if the MURA data from Stanford Hospital came from an unusually young age demographic (due to the nearby University), a system trained on that alone may be more likely to detect abnormalities affect young people (e.g. fractures, dislocations), over conditions affecting the elderly (e.g. joint degeneration). So if the system was then used in practice primarily with the elderly, it may be prone to mistakes.

8 Self-Assessment

8.1 Project Next Steps

Our MURA competition submission achieved a Cohen's kappa result of 0.52 on the validation set. Compared to the MURA authors' model result of 0.705 on the test set and the top-scoring result in the competition of 0.775, there is clearly a lot of scope to take the work done here further. Unfortunately, we ran out of time to do this before the project submission deadline. Nevertheless I think the work done here is a good start. If I were to take the project forward, I would look to do the following:

- Data Augmentation: artificially increase the amount of training data by applying random adjustments to the original images - such as small rotations, random-cropping, taking the mirror image. This not only gives more data for "free" but also should have a regularising effect, since the extra source of randomness helps models avoid overfitting to noise specific to the training set.
- Other Architectures: try a broader range of pre-built models. For instance try to follow the MURA authors' use of DenseNet architectures. Investigate more recent image classification competition winners, such as squeeze-and-excitation networks.
- Model ensembles: instead of submitting a single, final model, train several models with different architectures then aggregate their results to form the final predictions. This helps average out overfitting by single models.
- Hyperparameter tuning: take a more principled approach to searching the hyperparameter space, e.g. use grid search or cross-validation.
- Site-specific models: we could train specific models on each individual site (hand, wrist etc) and then use all of these on the test data if possible, picking the specific model for the specific site. This would be useful to the extent that abnormalities across different sites do not share common features. One test of this would be to assess site-specific models against the validation data for the other sites.
- Class activation maps: these use the activations of the final representation produced by a model's convolutional base to produce a heatmap over the original image. This heatmap shows which region(s) of the image resulted in the largest activations, and hence are responsible for

the model's output. This provides a window into what is otherwise a black box as to why the model made the prediction it did.

8.2 Learning Outcomes

I learned a lot while working on this project. I had never previously done a major piece of work using Python, and had never really done coding work on a project of this scope before. This includes learning how to use various data science packages such as Pandas and SciKit-Learn. I tried to follow reasonable design principles when structuring my Python code. For instance putting different phases of the data-model-results pipeline into different scripts - e.g. a script to process and resize the raw image files, another script to train models on these processed images, and a third script to produce results based on trained models.

In addition, I had to learn how to set up my own working environment - eventually settling on using a virtual machine on Google Cloud - which again is not something I've had experience with in the past. This involved installing all the software I needed, such as Python, Tensorflow etc. To do this, I had to get up to speed with using Linux which is something else I have not used much in the past.

I also had the opportunity to learn some Tensorflow, since I used it for my data handling and, technically, my model building and training (although really this was Keras moreso than Tensorflow proper). Originally I wrote my code using Tensorflow's older queue-runners approach, but this has been deprecated in favour of the much simpler Dataset API - so I used that instead. I had originally wanted to explore Tensorflow further, since it allows much finer-grained control and analysis of your neural network's computation graph. But in the end the need to produce an end-product pushed me towards to simplicity of Keras.

My review of the state-of-the-art convolutional network models helped me appreciate more clearly the design principles behind them - for instance, using 1×1 convolutions to reduce dimension, or factorising larger convolutions into successive 3×3 layers.

References

[1] P. Rajpurkar, J. Irvin, A. Bagul, D. Ding, T. Duan, H. Mehta, B. Yang, K. Zhu, D. Laird, R. L. Ball, C. Langlotz, K. Shpanskaya, M. P. Lungren, and A. Y. Ng, "MURA: Large Dataset for Abnormality Detection in Musculoskeletal Radiographs," *ArXiv e-prints*, Dec. 2017.

[2] "Artificial neural network." `https://en.wikipedia.org/wiki/Artificial_neural_network`, 2018. [Online; accessed 5-August-2018].

[3] Y. Bengio and X. Glorot, "Understanding the difficulty of training deep feed forward neural networks," *International Conference on Artificial Intelligence and Statistics*, pp. 249–256, Jan. 2010.

[4] "Tensorflow." `https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/ops/init_ops.py`, 2018. Commit: 0771f37 [Online; accessed 9-August-2018].

[5] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[6] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *ArXiv e-prints*, Feb. 2015.

[7] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[8] "Receiver operating characteristic." `https://en.wikipedia.org/wiki/Receiver_operating_characteristic`, 2018. [Online; accessed 12-August-2018].

[9] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[10] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *CVPR09*, 2009.

[11] M. Jacobs, "Deep learning in five and a half minutes." http://www.videantis.com/deep-learning-in-five-and-a-half-minutes.html, 2018. [Online; accessed 13-August-2018].

[12] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *ArXiv e-prints*, Sept. 2014.

[13] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," *ArXiv e-prints*, Sept. 2014.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *ArXiv e-prints*, Dec. 2015.

[15] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," *ArXiv e-prints*, Feb. 2016.

[16] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," *ArXiv e-prints*, Dec. 2015.

[17] X. Wang, Y. Peng, L. Lu, Z. Lu, M. Bagheri, and R. Summers, "Chestx-ray14: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases," 09 2017.

[18] L. Oakden-Rayner, "Exploring the chestxray14 dataset: problems." https://lukeoakdenrayner.wordpress.com/2017/12/18/the-chestxray14-dataset-problems/, 2017. [Online; accessed 29-August-2018].

[19] "Bilinear interpolation." https://en.wikipedia.org/wiki/Bilinear_interpolation, 2018. [Online; accessed 20-August-2018].

[20] "Bone x-ray deep learning competition." https://stanfordmlgroup.github.io/competitions/mura/, 2018. [Online; accessed 30-August-2018].

[21] "Keras documentation." https://keras.io/layers/normalization/, 2018. [Online; accessed 17-August-2018].

[22] F. Chollet, "Building powerful image classification models using very little data." https://blog.keras.io/

`building-powerful-image-classification-models-using-very-little-data.html`, 2016. [Online; accessed 30-August-2018].

[23] A. Alemi, “Improving inception and image classification in tensorflow.” `https://ai.googleblog.com/2016/08/improving-inception-and-image.html`, 2016. [Online; accessed 30-August-2018].

A How To Use My Model

A.1 Stucture

This appendix describes the structure of my working environment, with notes on what the various scripts do.

The MURA-Code.zip included in my project submission contains all the scripts I wrote as part of this project. The organisation (files and folders etc) inside it mirrors that of my working environment. However all data has been removed due to size constraints of the submission. With all raw and processed datasets, Keras model files, Tensorboard logs etc the size comes to 21GB. Therefore I have only included the scripts in the electronic submission.

The purpose of each of the scripts is described below. Their numbering indicates the order in which they should be run.

- 00_create_images_summary.py - creates a csv file that captures info about all the images in the MURA-v1.1 dataset, e.g. patientID, study outcome etc. This csv is used by subsequent scripts to know where each image is located.
- 01_process_images_for_training.py - resizes the raw images into a chosen dimension
- 02_keras_train_MURA_submission.py & other keras_*.py - trains the chosen model on the image data. These are all mostly duplicates of one another, with minor differences (e.g using one model vs another, adding regularisation etc). Some are labelled with _GPU_1 or _GPU_2 for my own benefit, to keep track when training different models simultaneously on different GPU's.
- 03_Predictions.py - calculates image-wise and study-wise performance of a provided list of trained models on the validation data
- ./models/*.py - provide functions that create pre-built architectures that are available in Keras. Used by some of the training scripts (should be clear which by the names).
- ./utils/utils.py - utility functions used across many scripts, e.g. a function to create a Tensorflow Dataset, used by all the training scripts.

The other folders are:

- keras_saves - the training scripts saved the Keras .h5 model files here
- tensorboard_logs - the training scripts saved Tensorboard log files here, which let me examine their accuracy/loss over training in Tensorboard
- results - contains the images_summary.csv produced by 00_create_images_summary.py. Also has some Jupyter Notebooks to breakdown that summary, and to produce results from the output of 03_Predictions.py (e.g. plot the ROC Curve).

There are also the following spreadsheets:

- MURA Breakdowns.xlsx - spreadsheet used to produce the data breakdowns, based off the images_summary.csv table, seen in Section 4.3.
- Model_Runs.xlsx - a log of all the various runs I performed.

A.2 Dependencies

Note that some of these scripts require the output of other scripts to run correctly. Specifically:

- 00_create_images_summary.py - expects ./data/raw to contain a folder called MURA-v1.1 which contains the raw image files.
- 01_process_images_for_training.py - expects ./results to contain the file images_summary.csv, which is created by 00_create_images_summary.py. Expects the raw images in ./data/raw as above, as well.
- 02_keras_train_MURA_submission.py & the other keras_*.py scripts - expect ./data/processed/folder to contain a folder called MURA-v1.1 with resized versions of the raw image files (created by 01_process_images_for_training.py). Typically the final folder will indicate the dimensions of the new resized images, e.g. ./data/processed/resized-100-100. These scripts also expect images_summary.csv in ./results as well.
- 03_Predictions.py - expects keras .h5 model files in the ./keras_saves folder. These are created by the keras_*.py scripts. Also expects processed images and images_summary.csv as above.

- ./models - the .py scripts in here have sum_weights_* functions that expect Keras .h5 weight files for their associated models inside the ./models folder with them. These are the pretrained ImageNet weights. Keras downloads these into the /.keras/models/ folder if necessary, however I copied them into my ./models folder to keep the project more self-contained. Note that the sum_weights_* functions are called in some of the model training scripts to verify that the pre-built architectures' weights have updated as required (or not, if the case may be).

A.3 Packages

Below is a list of the Python packages I used in my scripts, and the versions I used:

- Tensorflow: 1.9.0
- NumPy: 1.15.0
- Pandas: 0.23.3
- Python Image Library (PIL): 1.1.7
- SciKit-Learn: 0.19.1

Note that Tensorflow must be at least version 1.4.0 for my scripts to run, as they use the Dataset API which was added in this release.

A.4 Other Notes

A complete copy of my working environment with all data included can be found on the teaching server (teaching.cs.rhul.ac.uk) here:
/rmt/nas/home/pgt/meac055/MURA/

The raw images data (MURA-v1.1.zip) can be downloaded from here:
https://stanfordmlgroup.github.io/competitions/mura/

Note that after extracting the raw MURA-v1.1.zip, I moved the 4 csv files (train_image_paths.csv etc) up a level, out of MURA-v1.1/ into the docs/ folder. My 00_create_images_summary.py script may throw an error if this is not done, as it expects the MURA-v1.1/ folder to contain only images.

B Background - Further Details

B.1 Linear Activation Functions

To see that composing several layers of neuron with linear activation functions reduces to taking a linear function of the original input neurons, let $f(x) = x$ be the linear activation function used throughout the network. Then if layer 0 is the input layer, the activations of layer 1 are:

$$a^{[1]} = f(\mathbf{W^{[1]}}^\top a^{[0]} + b^{[1]}) = \mathbf{W^{[1]}}^\top a^{[0]} + b^{[1]}$$

And the activations for layer 2 are:

$$\begin{aligned} a^{[2]} &= f(\mathbf{W^{[2]}}^\top a^{[1]} + b^{[2]}) && (1) \\ &= \mathbf{W^{[2]}}^\top a^{[1]} + b^{[2]} && (2) \\ &= \mathbf{W^{[2]}}^\top (\mathbf{W^{[1]}}^\top a^{[0]} + b^{[1]}) + b^{[2]} && (3) \\ &= (\mathbf{W^{[1]}}\mathbf{W^{[2]}})^\top a^{[0]} + \mathbf{W^{[2]}}^\top b^{[1]} + b^{[2]} && (4) \\ &= \mathbf{W'}^\top a^{[0]} + b' && (5) \end{aligned}$$

where:

- $\mathbf{W'} = \mathbf{W^{[1]}}\mathbf{W^{[2]}}$ is an $n_0 \times n_2$ matrix
- $b' = \mathbf{W^{[2]}}^\top b^{[1]} + b^{[2]}$ is an $n_2 \times 1$ vector
- Equation (4) uses the matrix transpose rule $(AB)^\top = B^\top A^\top$

Hence both $a^{[1]}$ and $a^{[2]}$ are linear functions of $a^{[0]}$. This argument can be repeated to any number of hidden layers, and so the output of the whole network (regardless of its size) is simply a linear function of the input. This is undesirable since such a network cannot learn more complicated, non-linear features of the data.

B.2 Calculating Gradients in Backward Pass

Using the cross-entropy loss function, for a classification task as described above, we can calculate the gradient of the loss with respect to $\hat{y_j}$ (considering y_j as a fixed constant):

$$\frac{\partial L}{\partial \hat{y_j}} = \frac{\partial}{\partial \hat{y_j}}(-\sum_{j=1}^{n} y_j log(\hat{y_j}))$$

$$= \frac{\partial}{\partial \hat{y_j}}(-y_j log(\hat{y_j}))$$
$$= -\frac{y_j}{\hat{y_j}}$$

Recall that $\hat{y_j}$ is the activation of the j^{th} neuron in the output layer. Which itself is a derived from the weights $\mathbf{W}$ and biases b in the output layer, and the activations from the previous layer (layer l, say) $a^{[l]}$, fed through the Softmax function:

$$z = \mathbf{W}^\top a^{[l]} + b$$
$$\hat{y_j} = \frac{e^{z_j}}{\sum_{k=1}^n e^{z_k}}$$

And so, by the quotient rule:

$$\frac{\partial \hat{y_j}}{\partial z_j} = \frac{e^{z_j}(\sum_{k\neq j} e^{z_k} + e^{z_j}) - (e^{z_j})^2}{(\sum_{k=1}^n e^{z_k})^2}$$
$$= \frac{e^{z_j}}{\sum_{k=1}^n e^{z_k}} \frac{\sum_{k\neq j} e^{z_k}}{\sum_{k=1}^n e^{z_k}}$$
$$= \hat{y_j}(1 - \hat{y_j})$$

Where the final equality is due to:

$$1 - \hat{y_j} = 1 - \frac{e^{z_j}}{\sum_{k=1}^n e^{z_k}}$$
$$= \frac{\sum_{k=1}^n e^{z_k} - e^{z_j}}{\sum_{k=1}^n e^{z_k}}$$
$$= \frac{\sum_{k\neq j} e^{z_k}}{\sum_{k=1}^n e^{z_k}}$$

But since z_j appears in the denominator for all other y_i, $i \neq j$, we also need to calculate:

$$\frac{\partial \hat{y_i}}{\partial z_j} = \frac{\partial}{\partial z_j}(\frac{e^{z_i}}{\sum_{k\neq j} e^{z_k} + e^{z_j}})$$
$$= -\frac{e^{z_i}e^{z_j}}{(\sum_{k=1}^n e^{z_k})^2}$$
$$= -\hat{y_i}\hat{y_j}$$

Therefore we can apply the chain rule of differentiation to calculate the gradients of the loss with respect to the weights and biases in the output layer as follows:

$$\begin{aligned}
\frac{\partial L}{\partial z_j} &= \sum_{i=1}^{n} \frac{\partial L}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial z_j} \\
&= (-\frac{y_j}{\hat{y_j}})(\hat{y_j}(1-\hat{y_j})) + \sum_{i \neq j} (-\frac{y_i}{\hat{y_i}})(-\hat{y_i}\hat{y_j}) \\
&= -y_j(1-\hat{y_j}) + \sum_{i \neq j} y_i \hat{y_j} \\
&= -y_j + \sum_{i=1}^{n} y_i \hat{y_j} \\
&= -y_j + \hat{y_j} \sum_{i=1}^{n} y_i \\
&= \hat{y_j} - y_j
\end{aligned}$$

Where we've used the fact that $\sum_{i=1}^{n} y_i = 1$. Now if layer l has n_l neurons:

$$z_j = \sum_{i=1}^{n_l} w_{i,j} a_i^{[l]} + b_j$$

So:

$$\begin{aligned}
\frac{\partial z_j}{\partial w_{i,j}} &= a_i^{[l]} \\
\frac{\partial z_j}{\partial b_j} &= 1
\end{aligned}$$

And we have:

$$\begin{aligned}
\frac{\partial L}{\partial w_{i,j}} &= \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial w_{i,j}} \\
&= (\hat{y_j} - y_j) a_i^{[l]}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial L}{\partial b_j} &= \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial b_j} \\
&= \hat{y_j} - y_j
\end{aligned}$$

where $w_{i,j}$ is an element from j^{th} column of the weight matrix $\mathbf{W}$, and b_j the bias of the j^{th} neuron in the output layer.

We can also obtain similar formulas for the gradients of the loss with respect to the activations in layer l. Since each neuron in this layer connects to all neurons in the output layer, a given activation $a_i^{[l]}$ will appear in the formula (as shown above) for all z_j, $j = 1, \ldots, n$. And each of the z_j appear additively in the loss function via $\hat{y_j}$. Therefore we have:

$$\frac{\partial z_j}{\partial a_i^{[l]}} = w_{i,j}$$

$$\begin{aligned}\frac{\partial L}{\partial a_i^{[l]}} &= \sum_{j=1}^{n} \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial a_i^{[l]}} \\ &= \sum_{j=1}^{n} (\hat{y_j} - y_j) w_{i,j}\end{aligned}$$

But these activations $a^{[l]}$ themselves depend on the weights and biases in layer l, and so we can extend the chain rule further to calculate the gradient of the loss with respect to these as well. And we can repeat this process for the activations, weights and biases, in layers $l-1, l-2, \ldots$ etc all the way back through the entire network.

B.3 Gradient Descent

This involves performing a forward pass, calculating the activations and the loss, then performing a backward pass, calculating the gradients of all the weights, for every item of data in the training set. The gradients of each weight are then averaged across the entire set of training data, before performing the weight update using these average values for each weight. That is, for a weight w let $\frac{\partial L}{\partial w}_i$ be its gradient after forward and backward passes with the i^{th} element of the training set as input to the network. Then if there are n items of data in the training set, perform the following weight update:

$$w = w - \frac{\eta}{n} \sum_{i=1}^{n} \frac{\partial L}{\partial w}_i$$

That is, we have to perform a forward pass and backward pass for every item in the training set, and keep track of all associated gradients, before making a single weight update. This can be both slow and consume a lot of memory (consider a large network with millions of weights, and a large training set with millions of items). Note this process will often be repeated, with training involving hundreds or thousands of weight updates.

B.4 Stochastic Gradient Descent

Stochastic gradient descent (SGD) aims to speed things up by dividing the training set up into batches of a certain size (say, 128). Each data item in the training set is assigned randomly (hence "stochastic") without replacement to each of the batches. Then the weight updates are performed after calculating and averaging the gradients for each item in a batch. That is, if the batches are of size m, for a given weight we make the following update:

$$w = w - \frac{\eta}{m}\sum_{i=1}^{m}\frac{\partial L}{\partial w}_i$$

where $\frac{\partial L}{\partial w}_i$ is the gradient of w for the i^{th} item in the batch. An *epoch* is then defined as a pass through the full set of batches, performing a weight update for each batch. Thus if the training data of size n is split up into batches of size m, an epoch would involve $\left\lceil \frac{n}{m} \right\rceil$[10] weight updates - compared to just one for ordinary Gradient Descent. Training using SGD often involves running for several epochs. The random assignment of the training data into batches is repeated fresh at the start of each epoch.

The advantage of SGD is that it is clearly quicker than GD to perform weight updates. While each weight update will be subject to more randomness than in GD, since each update corresponds to a single, random batch rather than the full training set. However, over the course of several hundred or thousand weight updates, involving many epochs or passes over the whole training set, this should average out to give comparable results to GD.

B.5 RMSProp

Let w_t be a weight in the network, and let $g_t = \frac{\partial L}{\partial w}_t$ be its gradient, at iteration t. Then over all iterations we keep an exponential moving average of the square of these gradients:

$$v_t = \alpha g_t^2 + (1-\alpha)v_{t-1}$$

where the base case is $v_0 = 0$, and $\alpha \in [0,1]$ is a chosen hyperparameter that determines the extent to which (squared) gradients from past iterations

[10] $\lceil x \rceil$ is the smallest integer greater than or equal to x. If m does not divide into n, the final batch will comprise ($n \mod m$) items to ensure an epoch covers the entire training set.

affect the value of v_{t+1}. Note that an iteration can refer either to a full pass over the whole training set, as in GD, or over a batch, as in SGD. Although in practice we will use it in conjunction with SGD, since it inherits the benefits mentioned above. So the "gradient" g_t above would in fact be the average of the gradients for the weight across a batch. The weight update for the given iteration is then:

$$\delta_t = \frac{\eta}{\sqrt{v_t} + \epsilon} g_t$$

$$w_{t+1} = w_t - \delta_t$$

where η is the (global) learning rate, and ϵ is some small number (e.g. $\epsilon = 10^{-6}$) added to avoid division by 0. The $\frac{\eta}{\sqrt{v_t} + \epsilon}$ term now means that the learning rate η is dampened by the factor $\sqrt{v_t} + \epsilon$, which varies for each training iteration and for each weight in the network. So for a weight that had larger gradients in previous iterations, so a larger v_t, the effective learning rate will be smaller. Hence its subsequent weight updates get smaller as training progresses - slowing them down compared to weights with smaller gradients and weight updates at earlier iterations. This is desired, since weights that have seen large changes over training are more likely to be closer to the loss function minimum on which they are converging, and so should progress more slowly towards it (to avoid overshooting it, or other erratic behaviour).

B.6 Adam

Let:

$$m_t = \beta g_t + (1 - \beta) m_{t-1}$$

Again the base case is $m_0 = 0$, and $\beta \in [0, 1]$ is another hyperparameter that determines to what extent past gradients are taken credit for. The weight update is then:

$$\delta_t = \frac{\eta}{\sqrt{v_t} + \epsilon} m_t$$

$$w_{t+1} = w_t - \delta_t$$

where v_t, η and ϵ are defined as in RMSProp. This has the same benefits as RMSProp of dampening the learning rate η during training, but the weight update uses the smoothed m_t rather than the raw g_t at each iteration.

B.7 Batch Normalisation

Suppose we have a batch of size m, and x_i is a neuron's activation for the i^{th} example in the batch - we replace it with the transformation $y_i = \gamma \hat{x_i} + \beta$. Where γ and β are the learned scale and shift parameters respectively, and $\hat{x_i}$ is given by:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2$$

$$\hat{x_i} = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

Where ϵ is a small number (e.g. 10^{-3}) added to avoid division by 0. While we referred to neuron activations above, the authors actually reccomend to apply the batch normalisation transformation *before* the activation function. So if u is are the actiations from the previous layer, and g is the activation function, so that new the layer's activations are ordinarily:

$$z = g(Wu + b)$$

We instead let them be:

$$z = g(BN(Wu))$$

Where BN is the batch normalistion transformation. Note that the bias term b is now redundant, since it serves the same role as the batch norm shift parameter β - so it is dropped.

The authors argue that batch normalisation enables larger learning rates, and hence faster training, since it prevents small parameter changes from amplifying through the whole network into large, suboptimal weight updates (due to large gradients). It also separates the scale parameters from the scale of the learning rate - on which parameters depend via the weight update formula - since batch norm learns the scale and position of each activation (and hence gradient). And so it should compensate for learning rate being too high.

The authors also argue that batch normalisation regularises the model, since activation values are averaged over a batch rather than depending only on a single data item. This makes the model less prone to overfitting.

They argue that dropout in a batch-normalised network can either be removed, or reduced in strength.

Note that in the Keras implementation of batch norm [21], a moving average of the mean and variance are kept (with an adjustable "momentum" parameter). The final values of these moving averages are then used at test time in lieu of μ and σ^2 in the normalisation above - since normalisation a test or validation batch would consitute "data snooping".

C Pre-Built Model Architechtures

Below we present diagrams showing the structure of the pre-built architectures considered in this project.

C.1 VGG16

VGG16 architecture (from [22]):

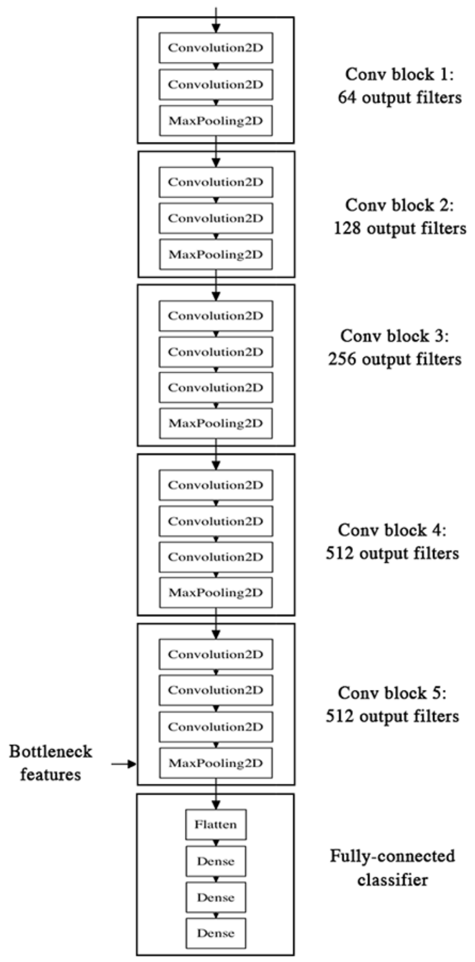


Figure 34: VGG16 Architecture

C.2 Inception-ResNet-v2

Inception-ResNet-v2 (from [23]) Example Inception block with skip connec-

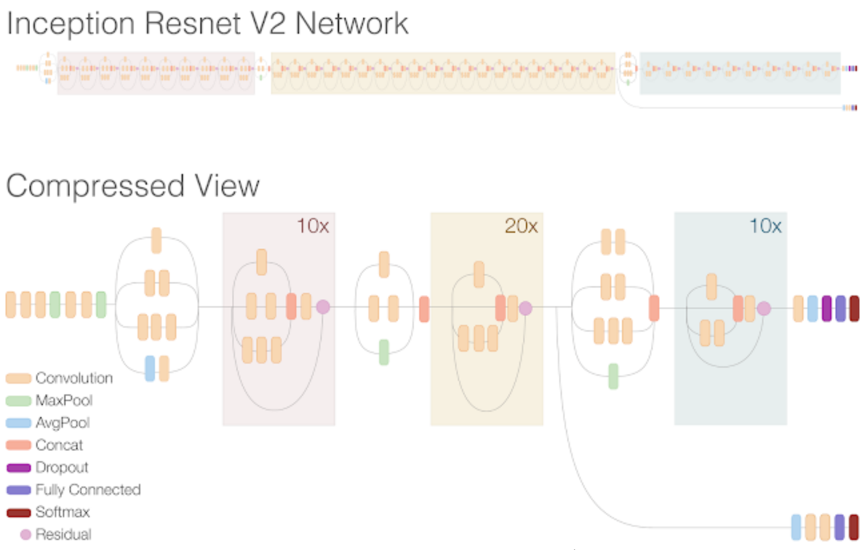
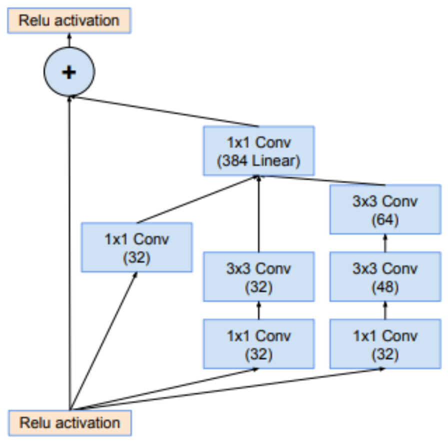


Figure 35: Inception-ResNet-v2 Architecture

tion:

Figure 36: Inception Block with Skip Connection



D Image Examples

The images from some example studies, of each body part and covering both normal and abnormal studies, are shown below. For some of the abnormal studies, the reason behind the result is clear - for instance, both Figure 40 of the Humerus and Figure 43 of the Wrist contain hardware (note the pin in the wrist). However the abnormality for some of the other studies is perhaps less clear to the untrained eye, for instance Figure 38 of the Finger.

D.1 Elbow

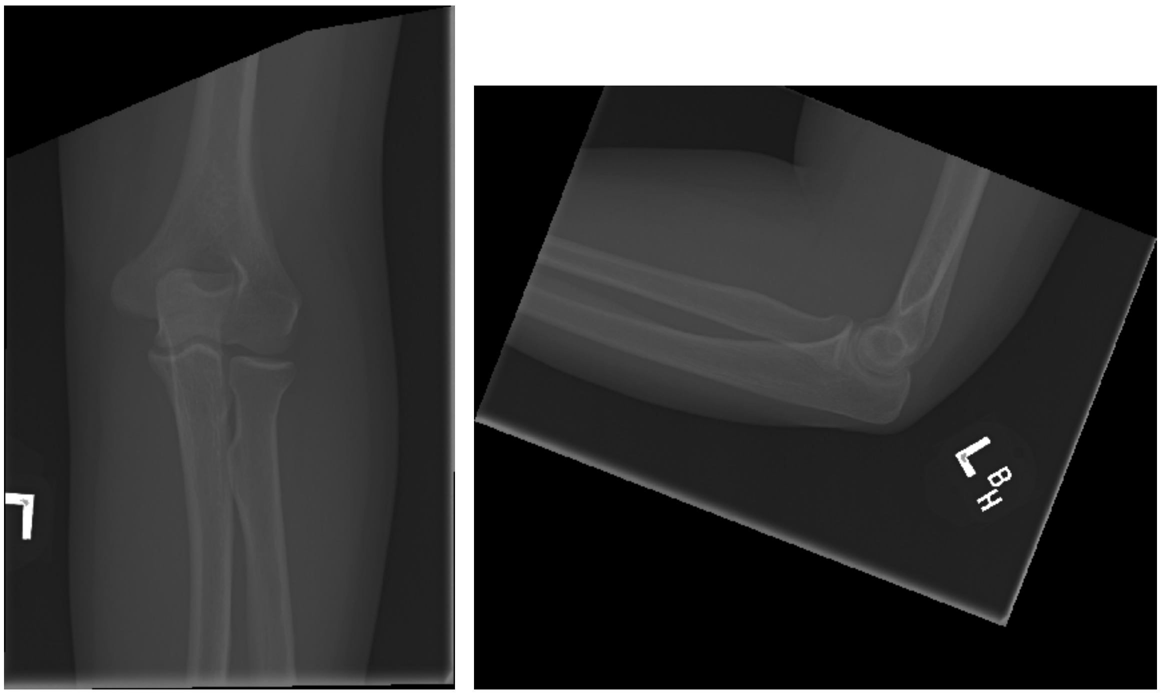


Figure 37: Patient 05926; Study 1; Elbow; Normal

D.2 Finger

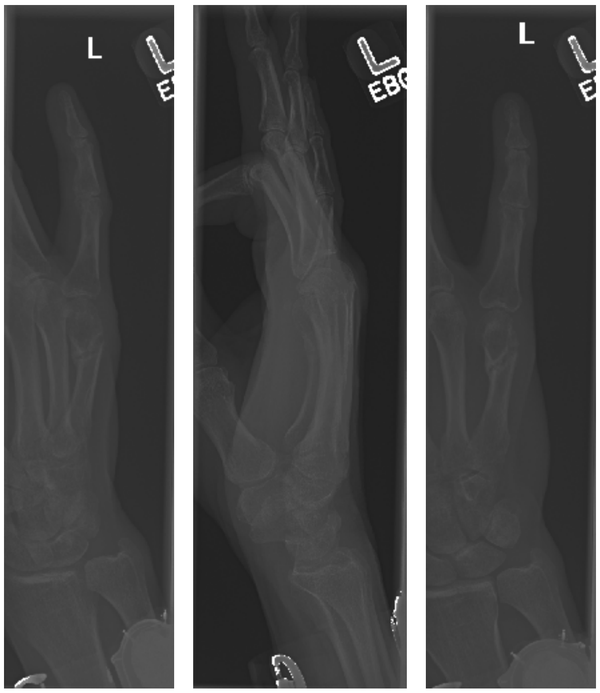


Figure 38: Patient 00234; Study 1; Finger; Abnormal

D.3 Hand

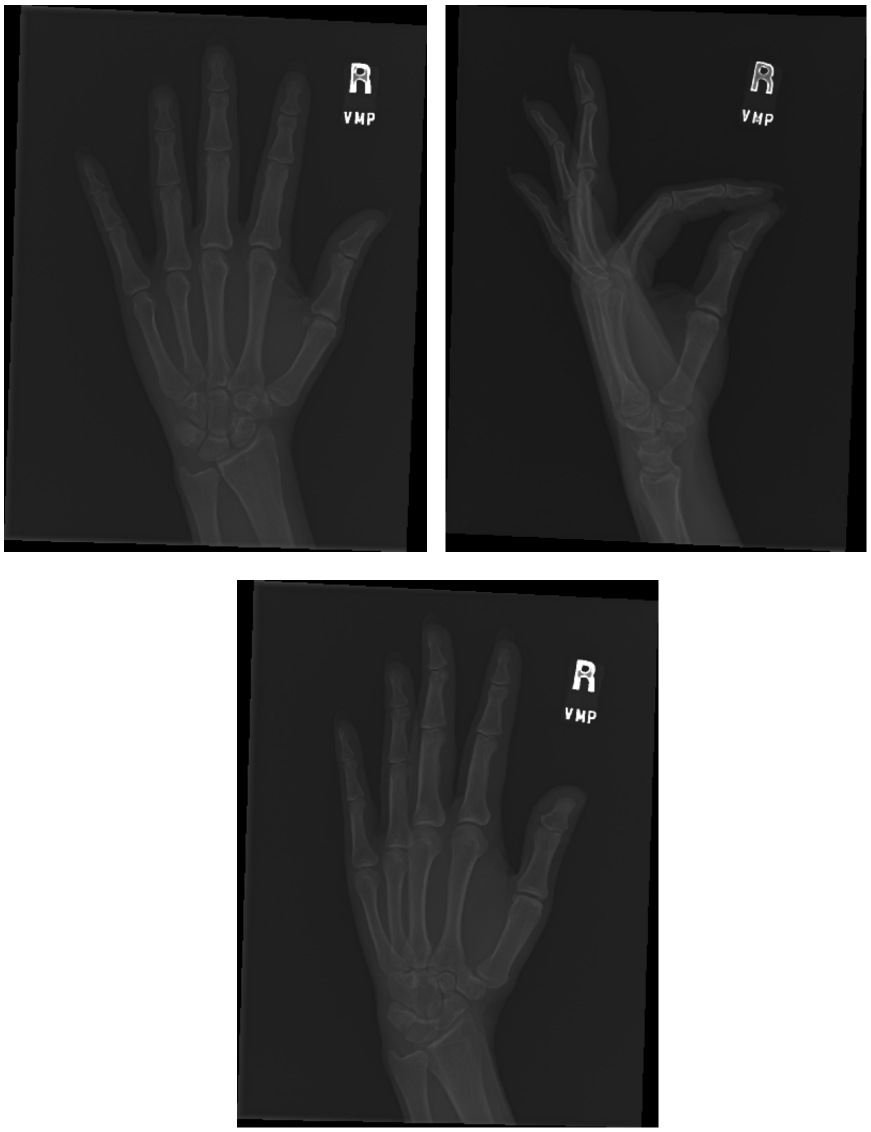


Figure 39: Patient 09836; Study 1; Hand; Abnormal

D.4 Humerus

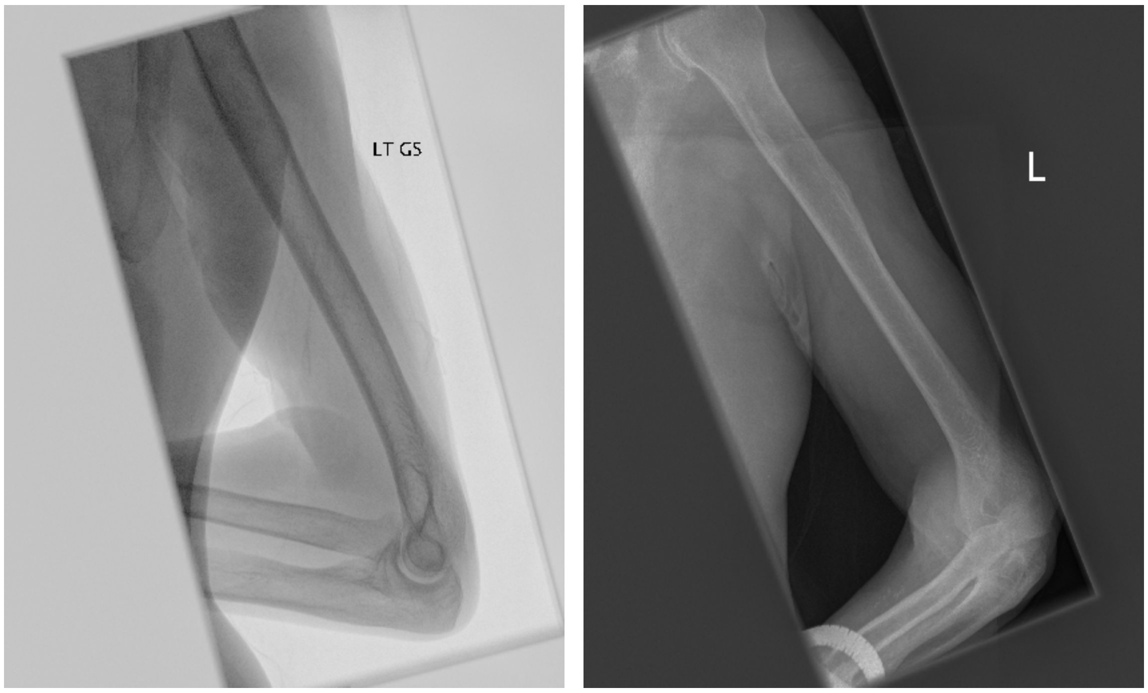


Figure 40: Patient 11225; Study 1; Humerus; Normal

D.5 Forearm

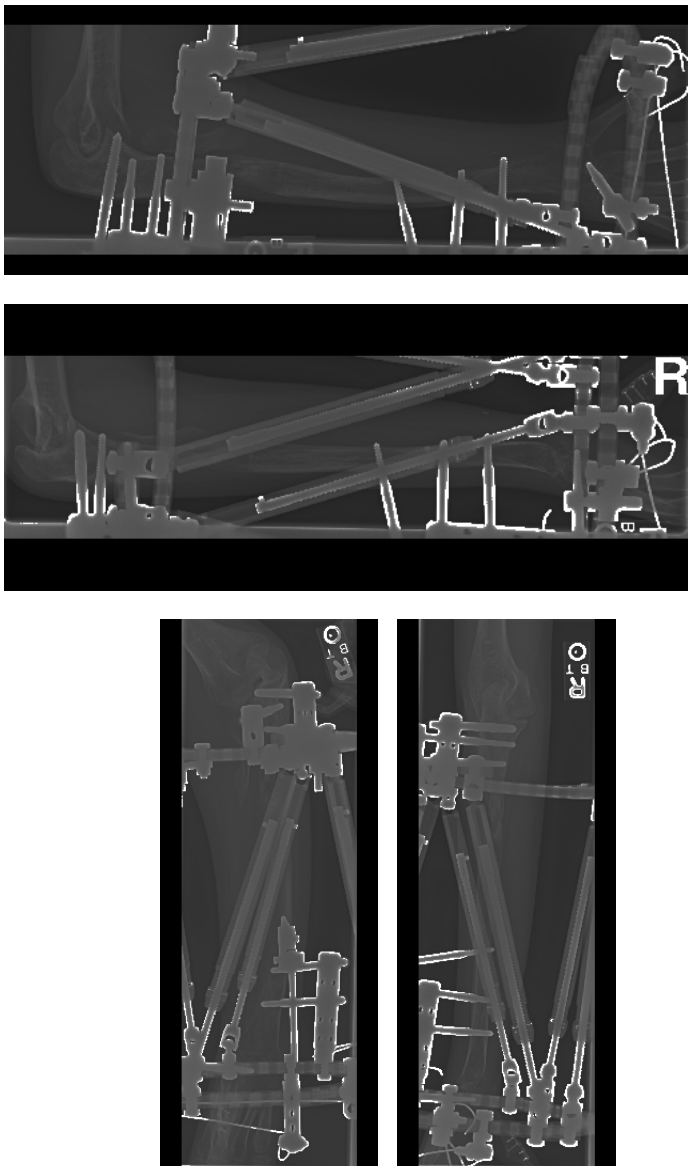


Figure 41: Patient 11417; Study 1; Forearm; Abnormal

D.6 Shoulder

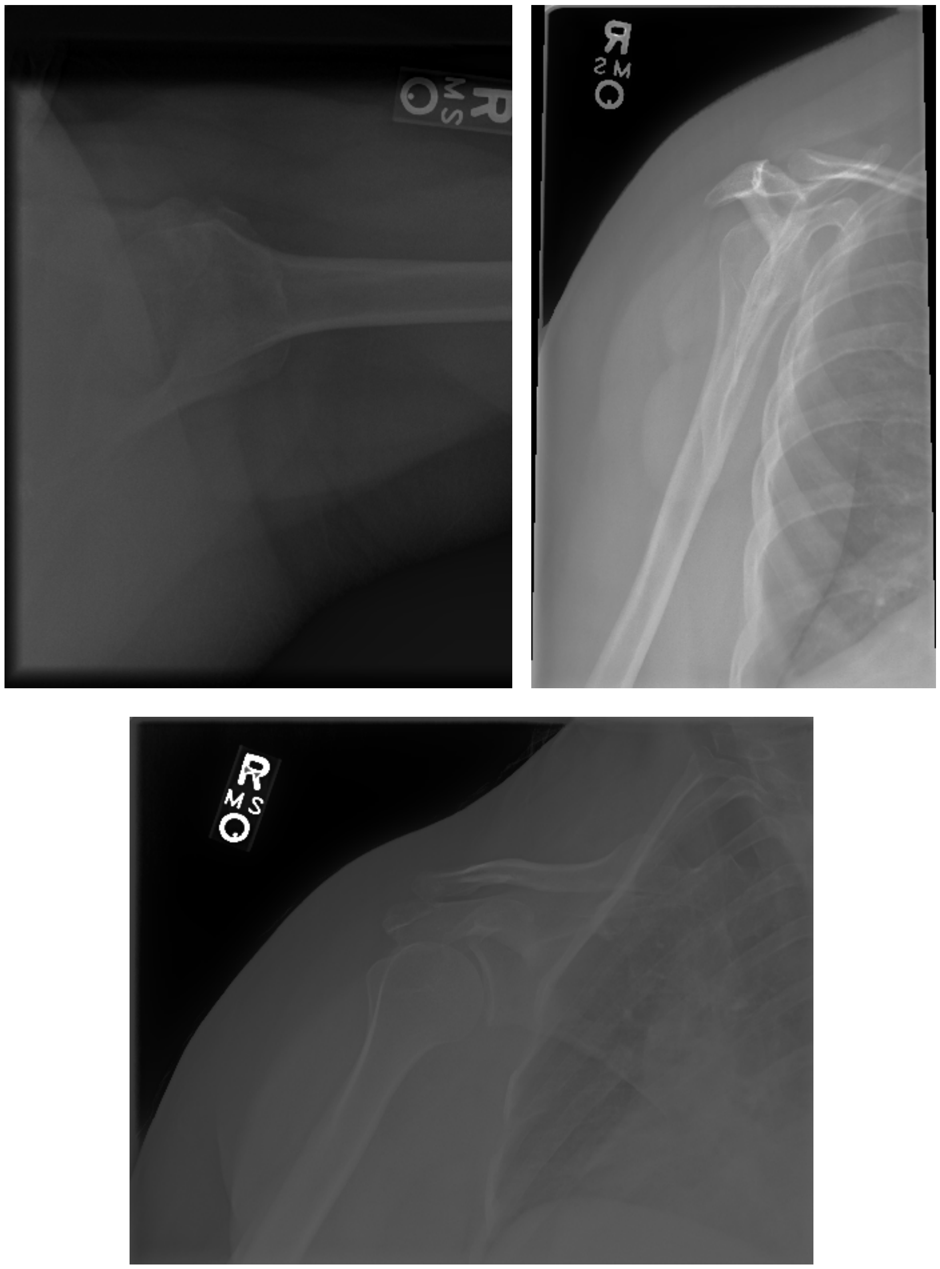


Figure 42: Patient 00001; Study 1; Shoulder; Abnormal

D.7 Wrist

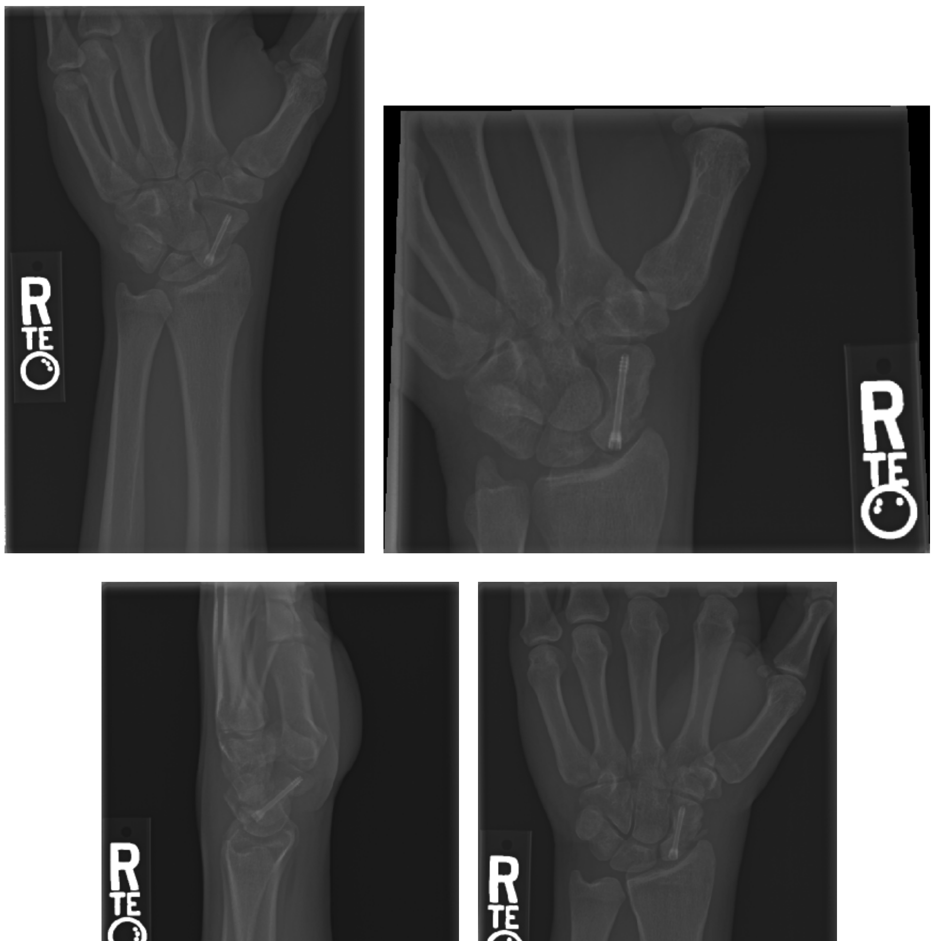


Figure 43: Patient 01521; Study 1; Wrist; Abnormal