Using Machine Learning to Diagnose Pneumonia from Covid-19 in Patients from Chest X-Rays

James Lamb

Student Ref:10558719

BSc Mathematics with Theoretical Physics

School of Engineering, Computing and Mathematics

University of Plymouth

March 25, 2021

Contents

1 Fully Connected Neural Networks

1.1 Neurons and Networks

Historically, the nodes of a neural network were modelled after the neurons in the brain, hence the nomenclature. The fundamental concept being that each neuron is essentially a continuous function that maps its output $y \in [0,1]$. We consider some activation function Ω which defines whether or not this neuron "fires", and some weight function, w, to model the synaptic plasticity of biologically connected neurons. This neuron can then be described,

$$y = \Omega(\sum_{j=0}^{n} w_j x_j) \tag{1.1}$$

Note that the j subscript denotes each individual input value or "signal" connected to the neuron. So, extending this to a layer of multiple neurons; intuitively we have,

$$y_i = \Omega(\sum_{j=0}^{n} w_{j,i} x_j)$$

This describes a fully connected layer of neurons if $\{x_j\}$ contains every input value. Further, if we consider the set of outputs of this layer as the set of inputs for another subsequent layer of neurons, prescribed by its own individual weights and possibly its own activation functions[1] then we have a fully connected neural network, sometimes referred to as a dense neural network (in reference to the large number of connections between each neuron). It is important that we include a bias term within these functions also. Without it our model is restricted to learning a function that specifically passes through the origin which limits its capability to model even simple functions (Gebel, 2020). Typically, this is introduced as a "bias neuron", an extra neuron in each layer that does not take any input value but stores some constant multiplied by a weight as an input for further layers. This weight is learned just as every other weight value is learned by the model. The effect of this allows the estimated function learned by the model to shift appropriately, this is shown graphically in Figure 1.1.

1.2 Non-Linearity

First let us consider a simple example of a neural network foregoing the use of an activation function, shown in Figure 1.2.

[1] In the case of the utilisation of a parameterised channel-wise activation function within the model.

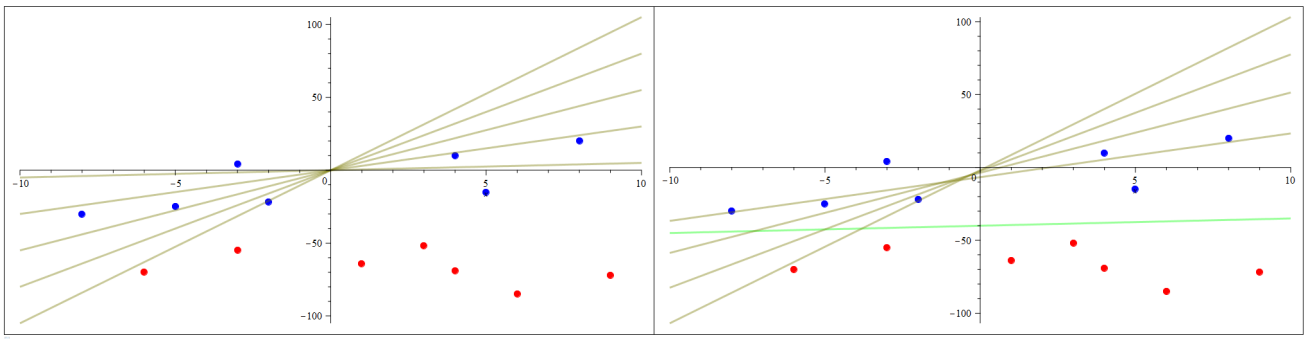


Figure 1.1: Consider an arbitrary binary classification problem, red or blue. The graph on the left shows linear representation without a bias term, the model can not find the linear function between the blue and red data points as it is restricted to passing through the origin. On the right graph, a bias term is included that allows the linear function to "shift" between the data points.

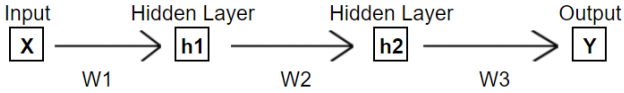


Figure 1.2: A simple neural network example.

Using the definition given for a neuron (1.1), and define the output of a hidden layer i to be h_i. We add bias terms for layer i, b_i, and omit the activation function, then we can describe this network as,

$$\begin{aligned} y &= h_2 \times w_3 + b_3 \\ &= (h_1 \times w_2 + b_2) \times w_3 + b_3 \\ &= h_1 \times w_2 \times w_3 + b_2 \times w_3 + b_3 \\ &= (x \times w_1 + b_1) \times w_2 \times w_3 + b_2 \times w_3 + b_3 \\ &= x \times w_1 \times w_2 \times w_3 + b_1 \times w_2 \times w_3 + b_2 \times w_3 + b_3 \\ &= x \times w_1 \times w_2 \times w_3 + b_1 \times w_2 \times w_3 + b_2 \times w_3 + b_3 \end{aligned} \tag{1.2}$$

Now we can combine each of the bias terms into a matrix, B and considering w_i to be some linear transformation and knowing that a combination of linear transformations is itself a linear transformation we can write them in a matrix, W, and rewrite equation 1.2 as,

$$y = x \times W + B. \tag{1.3}$$

Hence, any number of hidden layers with linear parameters will still result in a linear regression problem. This alone prevents the model from learning more complex functions. Further, a multi-neuron model is essentially no different to a single neuron model, hence adding more layers will not increase the efficacy.

To combat this, an activation function is used to adjust or limit the output and in doing so achieves non-linearity in the model parameters. There are a number of commonly used activation functions, we shall discuss a few to highlight the properties that are important when choosing a function.

1.2.1 Types of Activation Function

Heaviside Step Function

$$f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}, \qquad \frac{d}{dx} f(x) = \begin{cases} 0 & \text{if } x \neq 0 \\ undefined & \text{if } x = 0 \end{cases}$$

The simplest of activation functions; the binary step function activates a neuron in the model if positive and suppresses the neuron otherwise. This function is clearly suited to a simple binary classification model but is unsuitable for a multi-value classification system.

Tanh Function

$$f(x) = \frac{e^x - e^x}{e^x + e^x} \qquad \frac{d}{dx} f(x) = 1 - f(x)^2$$

The hyperbolic tangent function is essentially a scaled sigmoid function; rather than returning an output value $0 \leq y \leq 1$ it returns a value $-1 \leq y \leq 1$, meaning it has a stronger gradient and thus will train more aggressively. Furthermore, being a zero-centered function makes it a highly suitable choice for model inputs with neutral, strongly positive and negative values.

This has been a popular choice for non-linearity in machine learning due to the fact that it is non-linear in nature and has a smooth gradient. Also, because this gradient is steep, small changes in the input will result in a significant change in the output and hence brings the activation value to either side of the curve, making distinct predictions for a classification model.

It is also monotonic, so it will give better performance during the back propagation step of the model training.

There is a significant drawback to the Tanh activation function, however. On the extreme ends of the curve, the gradient becomes very small[2] which means as values tend further from 0, the change during back propagation (the gradient) tends toward 0. This can result in the network refusing to learn or becoming extremely slow in doing so.

ReLu

$$f(x) = \max(0, x) \qquad \frac{d}{dx} f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ undefined & \text{if } x = 0 \end{cases}$$

The ReLu (Rectified Linear Unit) function has gained popularity in recent years due to its computational advantage over other activation functions. This is due to the fact that it does not activate every neuron simultaneously. Namely, if the linear transformation of the output is less than 0 then it is not activated

[2] Also referred to as a vanishing gradient.

(Pedamonti, 2018). Further, those that are activated do not then involve more complex computational methods such as division.

Another advantage to the function is that it is easy to optimise using gradient-descent due to the fact that it is otherwise linear.

It is important to note, however, the drawbacks of this function. With all values for $x < 0$ mapping to 0, the negative portion of the function then has a derivative of 0. As mentioned previously in discussing Tanh, this can easily result in the vanishing gradient problem; dead neurons that refuse to learn further. Although the effect is somewhat diminished in regards to positive values, it is certainly more pronounced for the negative portion.

Modified versions of the Relu function have been defined to address this issue:

- PReLu (Parameterised Relu) -

$$f(x) = \begin{cases} \alpha_i x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \qquad \frac{d}{dx} f(x) = \begin{cases} \alpha & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

  - Formally, if $\alpha = 0.01$ then this function is the Leaky Relu function. The aim of this function is to eliminate the vanishing gradient for $x < 0$. The range, then, of the function is no longer bounded to a minimum of 0, but retains the majority of its computational power. This may also introduce a new parameter in the model for training, α. Typically this function is used when the Relu function fails to train a model effectively.

    In the case that the parameter a is defined as a learnable parameter, the function is formally a PReLu function. There are two case distinctions to make when this is the case; Whether the parameter is distinct for each channel or whether the parameter is common to all channels. This is referred to as channel-wise or channel-shared PReLu, respectively. It is important to note that the increase in the number of parameters in regards to the channel-wise case also increases the computational cost of the model, but as the number of parameters introduced (equal to the number of channels) is vastly less than that of the remainder of the model this cost can be disregarded as negligible.

- SReLu (S-Shaped ReLu) -

$$f(x) = \begin{cases} t_i^r + a_i^r(x_i - t_i^r) & \text{if } x_i \geq t_i^r \\ x_i & \text{if } t_i^r > x_i > t_i^l \\ t_i^l + a_i^l(x_i - t_i^l) & \text{if } x_i \leq t_i^l \end{cases} \qquad \frac{d}{dx} f(x) = \begin{cases} a_i^r & \text{if } x_i \geq t_i^r \\ 1 & \text{if } t_i^r > x_i > t_i^l \\ a_i^l & \text{if } x_i \leq t_i^l \end{cases}$$

  - Consisting of three piecewise linear functions, the SReLu function introduces four trainable parameters in the model (Jin et al., 2015). t_i^l, t_i^r represent the threshold between the central and left or right functions and a_i^l, a_i^r the slope of the left and right function lines respectively. Similar to the PReLu function, these extra parameters ($4N$, where N is the number of

channels, if channel-wise variant is used) increase the computational cost but as previously argued, the number remains negligible in comparison to the model as a whole.

The biggest advantage of using a modified ReLu function is in avoiding the dying neuron problem, further, parameterisation of the activation function allows a model to train more efficiently and reach greater accuracy at a minimal cost of computation time. It is worth noting though, in models that do not suffer from neuron necrosis, use of a modified Relu function is inefficient in comparison.

1.3 Learning in Dense Networks

Dense networks use the method of backpropagation in order to minimise error in its prediction. The network as described in the previous section is what is known as a feed forward network, information is passed and processed forward through the layers of the network to give the predicted output value, y. Backpropagation is an algorithm that essentially compares this given output to the expected value, $\hat{y}$, and adjusts the weights of the neurons proportionally to the respective derivative of the error. This is to minimise the overall cost function and hence produce more accurate results. It can be described in four steps (Cilimkovic, 2015):

Step 1: Feed-forward computation:

This is as described previously, only we now formally refer to layers with weight and activation functions as hidden layers. The original input is passed to the hidden layers whose function is y_i, the resultant values are fed forward to any further hidden layers and eventually to the output layer neuron, to give the prediction value.

Step 2: Back propagation to the output layer:

Firstly the error, the result of the model's cost function, is calculated, typically given as the mean squared error,

$$E = \frac{1}{2N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{1.4}$$

Intuitively it is clear that in minimising the cost function we inherently increase the accuracy of the model. In order to do this we need to understand the measure of change in the cost function in relation to a specific weight function, bias or activation (Hansen, 2020). First, for simplification, we incorporate the bias neurons into the weights, such that,

$$\begin{gathered} w^k_{0,i} = b^k_i \\ \longrightarrow a^k_i = b^k_i + \sum_{j=1}^{n_k-1} w^k_{j,i} x^{k-1}_j = \sum_{j=0}^{n_k-1} w^k_{j,i} x^{k-1}_j \end{gathered} \tag{1.5}$$

Where, n_k denotes the number of neurons in layer k. Then, considering that the derivative of a sum of functions is equal to that of the sum of the derivatives of each function, we can consider the derivation for a single input-output pair and derive a general form for all pairs. So,

$$C = \frac{1}{2}(y_i - \hat{y}_i)^2 \tag{1.6}$$

and by the chain rule,

$$\frac{\partial C}{\partial w_{j,i}^k} = \frac{\partial C}{\partial a_i^k}\frac{\partial a_i^k}{\partial w_{j,i}^k} \tag{1.7}$$

Here, a simply represents the sum of the products plus bias before applying the activation function. Now, as,

$$\begin{aligned} y &= \Omega(a_i^k) \\ \Rightarrow \frac{\partial C}{\partial a_i^k} &= (y - \hat{y})\Omega'(a_i^k) \end{aligned} \tag{1.8}$$

and,

$$\frac{\partial a_i^k}{\partial w_{j,i}^k} = x_j^{k-1} \tag{1.9}$$

So we have the derivative with respect to the weights given for the final layer,

$$\frac{\partial C}{\partial w_{j,i}^k} = (y - \hat{y})\Omega'(a_1^k)x_j^{k-1} \tag{1.10}$$

Step 3: Back propagation to the hidden layer/s:

Extending this to further layers, by the chain rule, we have,

$$\frac{\partial C}{\partial a_i^k} = \sum_{n=1}^{n_k+1} \frac{\partial C}{\partial a_n^{k+1}}\frac{\partial a_n^{k+1}}{\partial a_i^k} \tag{1.11}$$

The bias term is not dependant on previous layers so it is not included in this summation. It follows that,

$$\begin{aligned} \frac{\partial a_n^{k+1}}{\partial a_i^k} &= w_{j,i}^{k+1}\Omega'(a_j^k) \\ \Rightarrow \frac{\partial C}{\partial a_i^k} &= \Omega'(a_j^k)\sum_{n=1}^{n_k+1} w_{j,n}^{k+1}(y - \hat{y})\Omega'(a_j^{k+1}) \end{aligned} \tag{1.12}$$

and

$$\frac{\partial C}{\partial w_{j,i}^k} = \frac{\partial C}{\partial a_i^k}\frac{\partial a_i^k}{\partial w_{j,i}^k} = \Omega'(a_j^k)x_j^{k-1}\sum_{n=1}^{n_k+1} w_{j,n}^{k+1}(y - \hat{y})\Omega'(a_n^{k+1}) \tag{1.13}$$

It is from here that the term backpropagation earns its nomenclature, the error calculated at a layer, k, depends on the error calculated at the next layer, $k+1$. Whereas the output layer's error is calculated only from the predicted output and the expected output, all previous layer's are essentially then a weighted product sum of this error scaled by the derivative of the activation function iterated back through the layer's to the input layer.

Step 4: Weight updates:

Combining these results and applying a scaling factor equivalent to the learning rate of the model,

$$\Delta w_{i,j}^{k} = -\alpha[(y-\hat{y})\Omega'(a_1^k)x_j^{k-1} + \Omega'(a_j^k)x_j^{k-1}\sum_{n=1}^{n_k+1} w_{j,n}^{k+1}(y-\hat{y})\Omega'(a_n^{k+1})]. \tag{1.14}$$

Here, α denotes the learning rate of the model. This is a hyper-parameter that restricts the magnitude of the change applied to every weight. This is generally a very small number but it's optimum value depends on the data and the model (This is discussed further in section 3.2).

It is conventional that the individual summation terms and the final layer's gradient are collected as a vector which is then applied to each weight (including bias) of the model. It is also conventional that the weights of the model therefore also be stored as a matrix (See section 2.1.1), this not only simplifies the application of the updates but specifically optimises the update for GPU computation.

This algorithm continues until the error function becomes sufficiently small. It is also a general convention that the algorithm is performed on small batches of the data at a time in order to improve the performance and computational time of the model (Paeedeh and Ghiasi-Shirazi, 2020).

2 Overview of Convolutional Neural Network

2.1 Elements of a CNN

2.1.1 Convolutions

The main difference of a Convolutional Neural Network (CNN) over other neural networks is the use of convolution, a linear operation used for feature extraction. CNNs have been found to be very useful in image classification problems. Formally, the convolutional formula (Albawi, Mohammed, and Al-Zawi, 2017) can be given as,

$$G[i,j] = f * k[i,j] = \sum_m \sum_n k[m,n] \times f[i-m, j-n] \ .$$

$$f = Image$$

$$k = Kernal$$

Consider an input image of 16×16 pixels with RGB channels. In a typical neural network, to connect this input to a single neuron would require $16 \times 16 \times 3 = 768$ weighted connections. To reduce this number, we define a "kernel" (perhaps several) as an $n \times m$ matrix [1] to isolate local regions of the input image.

The kernal can be visualised as a kind of view-port sliding over the image. This means an assumption can be made in that for each local region we use a duplicate neuron, the weights applied are remain fixed and identical across each neighbouring neuron, thus reducing further the number of parameters. Further, by considering that the filters which are intended to be applied to the input image can be represented in matrix form then the kernal itself can be used as a filter. To highlight this, consider a 1-dimensional convolutional layer wherein every neuron can be described in the function,

$$\Omega(\sum_{i=0}^{n}(w_i x_i) + b) \ . \tag{2.1}$$

Now consider the weight matrix, W, which consists of each weight element, w_i. In the case of a regular ANN, shown in (2.2), every input pixel is connected to each neuron with a different weight. However within a convolution layer (2.3), many of the neurons are disconnected and therefore have zero value,

[1] typically $n = m$, unless specific features of the input image are known prior to processing for which $n \neq m$ would be computationally optimal.

and due to the multiple copies of neurons we have the same weight functions applied in differing positions.

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,n} \\ w_{1,0} & w_{1,1} & & \\ \vdots & & \ddots & \\ w_{n,0} & & & w_{n,n} \end{bmatrix} \tag{2.2}$$

$$\begin{bmatrix} w_0 & w_1 & 0 & 0 & \dots \\ 0 & w_0 & w_1 & 0 & \dots \\ 0 & 0 & w_0 & w_1 & \dots \\ 0 & 0 & 0 & w_0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{2.3}$$

Hence each output neuron only has an interaction with the input pixel within the convolutional layer's kernal, and those weight parameters are shared across each neighbouring layer, vastly reducing the number of parameters requiring optimisation.

The fact that the weights are shared means that the features of an image are then considered to be spatially invariant. This implies that convolution may not be appropriate for applications where feature positions in the input image are important.

2.1.2 Stride

Simply put, the stride defines the pixel distance the kernal must translate to process each local region of the input image. By setting this value, and considering the chosen kernal size, the overlap of each region is also defined along with the size of the output (Krizhevsky, Sutskever, and Hinton, 2017).

Consider a $N \times N$ image with a $K \times K$ kernal. Choosing the stride to be S, then the overlap is given by $K - S$ and the size of the output matrix is given by,

$$O = \frac{N-K}{S} + 1 \ .$$

Note that if stride and kernal are chosen such that $\frac{N-K}{S} \notin \mathbb{Z}$, then the remainder of the image pixels are not parsed by the kernal and do not factor into the output, hence the output size is more accurately given by,

$$O = \lfloor \frac{N-K}{S} \rfloor + 1 \ .$$

This can be further extended to three dimensions such that the process is repeated for each channel

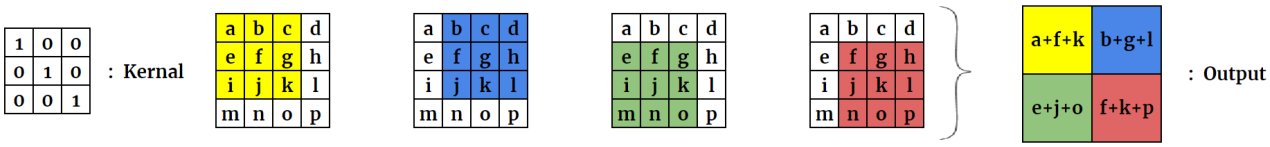


Figure 2.1: Kernal of order 3 acting on 4×4 image

separately and the resulting 2-dimensional outputs are collated into a single 3-dimensional tensor, ie.

$$O = [N,N,N_c] * [K,K,N_c] = [\lfloor \frac{N-K}{S} \rfloor + 1, \lfloor \frac{N-K}{S} \rfloor + 1, N_f] \ .$$

$$N_c = \text{Number of image channels}$$

$$N_f = \text{Number of filters}$$

A brief consideration of higher dimensions can be made. The fundamental amendment to the procedure here is in using a 3-dimensional tensor as a kernal. This acts on the input in a similar manner as before only it now slides in three dimensions, hence the output is also reduced in three dimensions.

2.1.3 Padding

When a convolution layer is formed as discussed, the kernal captures information and collates it into a tensor, as shown in figure 2.1. This typically results in a loss of information at the border of the local region.

In order to retain this information, a method called zero-padding[2] is utilised wherein an artificial border of zero value is added to the input matrix, as graphically depicted in Figure 2.2. Using this method prevents the output size reducing with the depth of the convolution layers and inherently then allows the network to have any number of convolutional layers.

It also permits further management of the size of the output, with the modification to the formula for output size given by,

$$O = \lfloor \frac{N-K+2P}{S} \rfloor + 1 \ .$$

Where P is the number of padding appended. Or the 3-D variant,

$$O = [N,N,N_c] * [K,K,N_c] = [\lfloor \frac{N-K+2P}{S} \rfloor + 1, \lfloor \frac{N-K+2P}{S} \rfloor + 1, N_f]$$

[2]Commonly referred to as "same" padding. Conversely, "valid" padding refers to convolution without any padding appended to the input image.

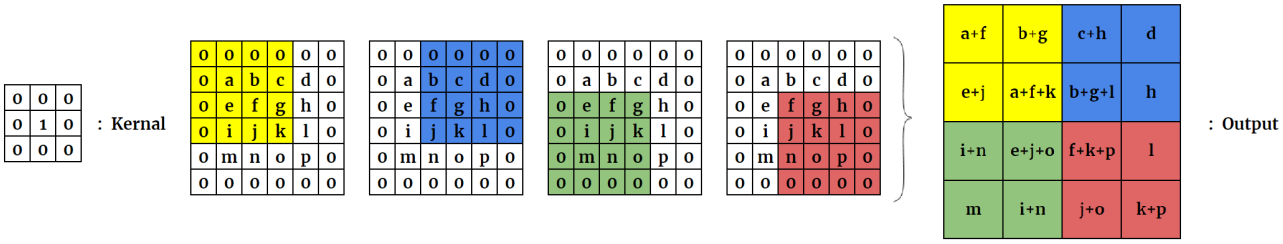


Figure 2.2: Kernal of order 4 acting on 4×4 image with zero-padding

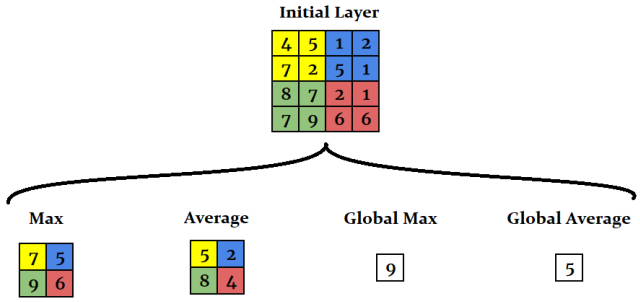


Figure 2.3: Depiction of different pooling outputs in regards to a sample initial layer

2.1.4 Pooling

Pooling, much like valid padded convolution, is a process utilised to reduce the sampling rate of its proceeding layers. Depending on whether max or average pooling is used this reduction is equivalent to either down-sampling or decimation, respectively. This acts to reduce noise in the input layers and secondly to speed up computation (Gholamalinezhad and Khosravi, 2020).

Pooling also has its own stride parameter which, much like the stride parameter for the kernal, defines how many pixels across the tensor moves each time an element of the output tensor is evaluated and again inherently defines the output size and overlap. In contrast however, it is not typical to have an overlap in the pooling layer, although it has been noted to provide more accurate results in the model. In either case, much like the kernal, a tensor evaluates a set of sub-regions of a layer and returns a new tensor consisting of values representing each of those sub-regions.

The most commonly used pooling technique is max pooling. The sub-region is evaluated such that the maximum value within the region is returned and the remaining values are discarded.

ie. Consider the set of sub-regions of the layer, $\mathfrak{s}$, then

$$p_{i,j} = \max x_{p,q} \; . \qquad\qquad x_{p,q} \in \mathfrak{s}_{\mathrm{i,j}}$$

The second pooling technique to consider is average pooling. In this case the average of all of the values within each sub-region becomes the relative element of the pooled output layer.

Both of these pooling techniques can also be extended to what is called global pooling. Instead of returning a tensor, a single number is returned, taken from either the maximum or average of all

the elements in the layer. This technique is more appropriate in applications such as natural language processing rather than purposes such as machine vision.

Each of these pooling techniques are shown graphically using an example layer in Figure 2.3.

3 Evaluation and Evolution of Model

Machine learning models generally do not work "out-of-the-box". There are a number of values to be modified in order to fine tune the performance of a model such that it is effective. Theses values are known as "hyperparameters" and refer to values which cannot be learned by the estimator of the model itself. Conversely, the term parameter refers to the values estimated within the model, although often "parameters" is used as an umbrella term for both hyperparameters and model parameters.

A large part of a model's effectiveness is then determined by the choice of these hyperparameters, it is then worth discussing the methods by which we come to these "best" values. Some are data and problem specific, much like the model architecture itself, and can be determined intuitively such as is the case of the choice of the activation function. E.g, a classification problem would imply the use of softmax/sigmoid function whereas a numerical problem would imply the use of a linear activation (Ronaghan, 2021).

Other times, however, the values may be less intuitive and there are a number of methods by which to determine them, these will be discussed further in the following sections. The important detail to note is that these methods generally rely on testing the model and evaluating the results, hence the model will typically evolve in accordance to its evaluation.

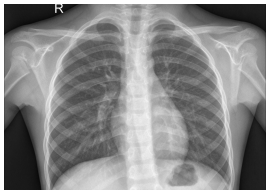
3.1 Data

The initial dataset, from which the model is trained, contained 5,856 grayscale chest x-ray images split into training, testing and evaluation sets (See Figure 3.1 for examples). The model does not partition the data[1]. The data was somewhat biased as it contained twice the number of pneumonia x-rays than that of the normal case. This bias inherently affects the learning of the model and so, to compensate for this, the model was designed to randomly filter pneumonia cases from each of the training, test and evaluation sets, such that the number of images for either case is equal and bias is eliminated.

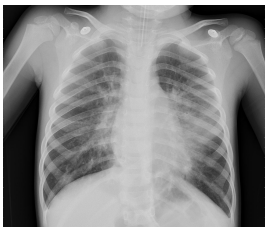
This kind of filtering inherently reduces the number of data available for the model to train on however, which in turn reduces the amount of information available also. In some cases, this reduction in information is manageable, but as we effectively lose half of the

In diagnosis of pneumonia, the main consideration made of x-rays is the presence of opacity in the lungs. This opacity can be the result of a number of different causes from pneumonia to pleural

[1]This is for the sake of efficiency in testing the model. In further implementations, shuffling the data, (using the random forest algorithm, for example) would be a typical approach.



(a) A normal lung x-ray, no opacity.



(b) Opacity in the lungs indicating pneumonia.

Figure 3.1: Two example images from the testing subset of the dataset.

effusion, so we look to create a model that can not only detect whether opacity is present in the x-ray images, but also differentiate if that opacity is in fact pneumonia.

3.2 Hyperparameters and Model Tuning

3.2.1 Learning Rate

Arguably the most important hyper-parameter to optimise in the model is the learning rate. As shown in section 1.2, the learning rate directly affects how large the error is when updating the weights. If the rate is too large, then the model may "overshoot" when correcting the weights and ultimately diverge away from optimal values; too small means the back propagation would slow down and could severely impact computational time. There is no sure-fire learning rate value for every model, it is dependent on both the model architecture and the function being learned. Instead, we start with a rough value and optimise the model through testing. A good typical starting value for the learning rate is 0.01 for standard multi-layer neural networks (Bengio, 2012). For optimising the learning rate, the model was tested around the initial learning rate value. By graphing the learning rate against the loss of the model, an optimal point where the loss is least was identified and then the model was tested again around this new value. This method is known as a grid search (Goodfellow, Bengio, and Courville, 2016). From the graphs of the results, shown in Figure 3.2, it is clear that an optimal learning rate for the model is 0.0007.

3.2.2 Initial Class Weights

Although technically class weights are considered general parameters in the model, one can treat the initial values of class weights as a hyperparameter. The biggest reason for setting these initial values is to offset the impact of an imbalanced dataset. For example, in our original model, there was a clear imbalance that created a bias toward pnuemonia. In this case we chose to address this issue by modifying the data directly such that we had "balanced data", ie. the number of data for each case fed into the model was equal. We could have instead found the ratio between the two classes

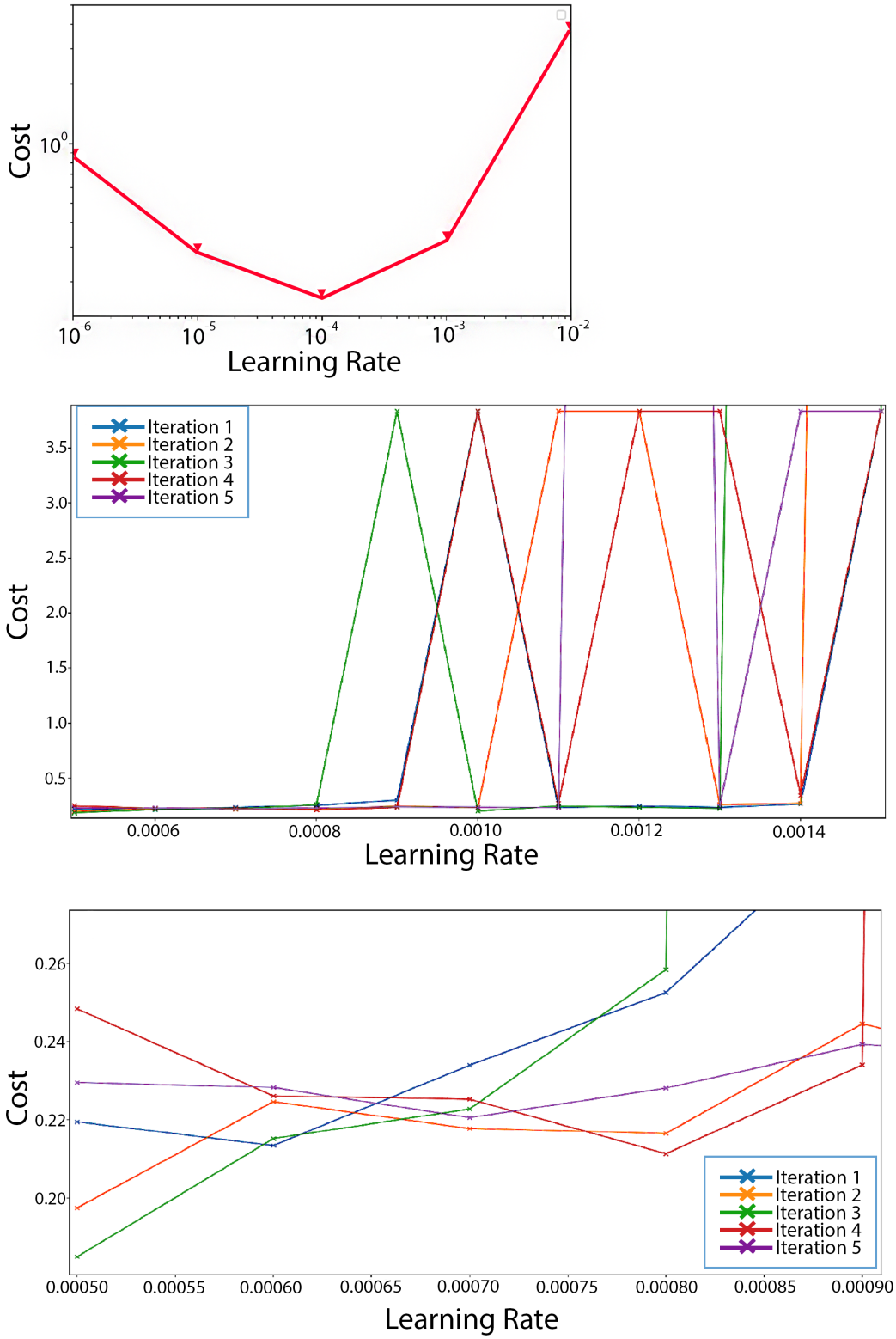


Figure 3.2: Cost vs Learning Rate of grid searches. The first graph suggest an optimal learning rate of 10^{-4} which is used as the centre point of the next set of experiments. The (most stable) minimum from the second graph then suggests the centre for the next set and so on. Here, grouping of points in third graph suggest a stable optimal value for the learning rate is 0.00070.

and modified the initial class weights to represent this so that, during back propagation, the effects of learning on the under-represented class would be more impactful to the model. In this case, we would have "representative data".

It is also important to note that over consecutive epochs the effect of weighting tends to vanish and is only effective in the early stages of training (Byrd and Lipton, 2019). This is does not nullify the importance of weighting however. Even a balanced model may need an initial weighting in order to perform training more effectively. Consider our model, for example. In order to reduce the loss of information, we increased the number of data so that we had a balanced set without reducing information. Yet using a default non-weighted model we consistently received poor results in validation corresponding to little to no learning. So then we argue the case that what we are modelling is essentially an anomaly detection problem, it is more important that the model learns the normal case and classifies depending on it's detection of anomalies (opacity). So then we must modify the models weighting of the data in such a manner. By performing a grid search over an appropriate range of values we find an optimum weighting to be $\text{Class}_{\text{normal}} : 2.0$, $\text{Class}_{\text{opacity}} : 0.4$ and by initialising training in this manner, the model learns more effectively and can continue to train effectively throughout.

3.3 Evaluation

3.3.1 Accuracy, Precision and Recall: The F-Score.

Formally, when discussing accuracy of a model, we refer to the "correct" predictions of the models as being "relevant" data points. For the sake of simplification and ease of conceptualisation, we will focus on these metrics in terms of the model we have built, where positive predictions refer to relevant data points and analogously for negative predictions. Accuracy, as the name implies, measures how accurate the model is at predicting a true value/classification. It is given by it's expected mathematical expression,

$$\text{Accuracy} = \frac{\text{Accurate Predictions}}{\text{All Predictions}}$$

But this calculation only reveals so much. Consider the case of a model using inbalanced data, say 90% of the data is negative and 10% is positive. A model may learn to predict that all cases presented are negative, a poor assumption that would never predict a positive and yet the model would still report having 90% accuracy. In the case of healthcare, this is a potentially disastrous case.

So instead we delve further and determine more specific statistics. The precision and recall are two similar, but fundamentally different statistics of the model. Precision being a measure of the proportion of predicted positives that were in fact positive. Recall (Also known as sensitivity) being a measure of the proportion of accurate positive predictions over all positive data. They can be expressed,

$$\text{Precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}, \quad \text{Recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

These two metrics, precision and recall, each highlight a specific nuance of a model's accuracy. So in combination we can define a more precise estimation of accuracy. Specifically we define the F-Score of a model to be the harmonic mean of these metrics,

$$F_1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

This is known as the F1-score, wherein a perfect model would give a result of 1. It is used to evaluate binary classification systems. (Wood, 2021) Often, however, the data which we work with will influence how much we depends on a particular metric to be more important. An inbalanced dataset biased towards negative results, as previously discussed, would give a high accuracy and a high recall intrinsically, but a true reflection of the model's effectiveness would be found in evaluating its precision. Further, in the field of healthcare, a false negative (say, in diagnosing Covid-19) is far more dangerous than a false positive, so it would be more important to obtain a higher recall in evaluation. Therefore it is preferable to use the generalised F-Score,

$$F_\beta = (1 + \beta^2) \times \frac{Precision \times Recall}{\beta^2 \times Precision + Recall}$$

Here we have a parameter β by which to weight the precision or recall more highly. $\beta > 1$ gives a score favouring recall, and $\beta < 1$ favours precision.

3.3.2 Confusion Matrices

Confusion matrices show how a model evaluates data in comparison to how the data is actually categorised, tallying the comparison within a specific box denoting correct or incorrect assignment. (0,0) denotes a correctly predicted false; (0,1), a false positive; (1,0) a false negative and (1,1), a correctly predicted positive. This may be extended to a higher dimensional matrix, where positive and negative extends into classifications $1, 2, 3, ..., n$. For the sake of our discussion, however, we shall focus on the 2×2 case. By labelling the matrix,

		Prediction	
		Negative	**Positive**
Actual value	**Negative**	True Negative	False Positive
	Positive	False Negative	True Positive

We see that it is essentially a graphical representation of the metrics discussed in the previous section and is useful in representing a models performance. Figure 3.3 shows a sample of a set taken from experiment "e2c9144aa", a 50 epoch training run using balanced data and a binary loss function.

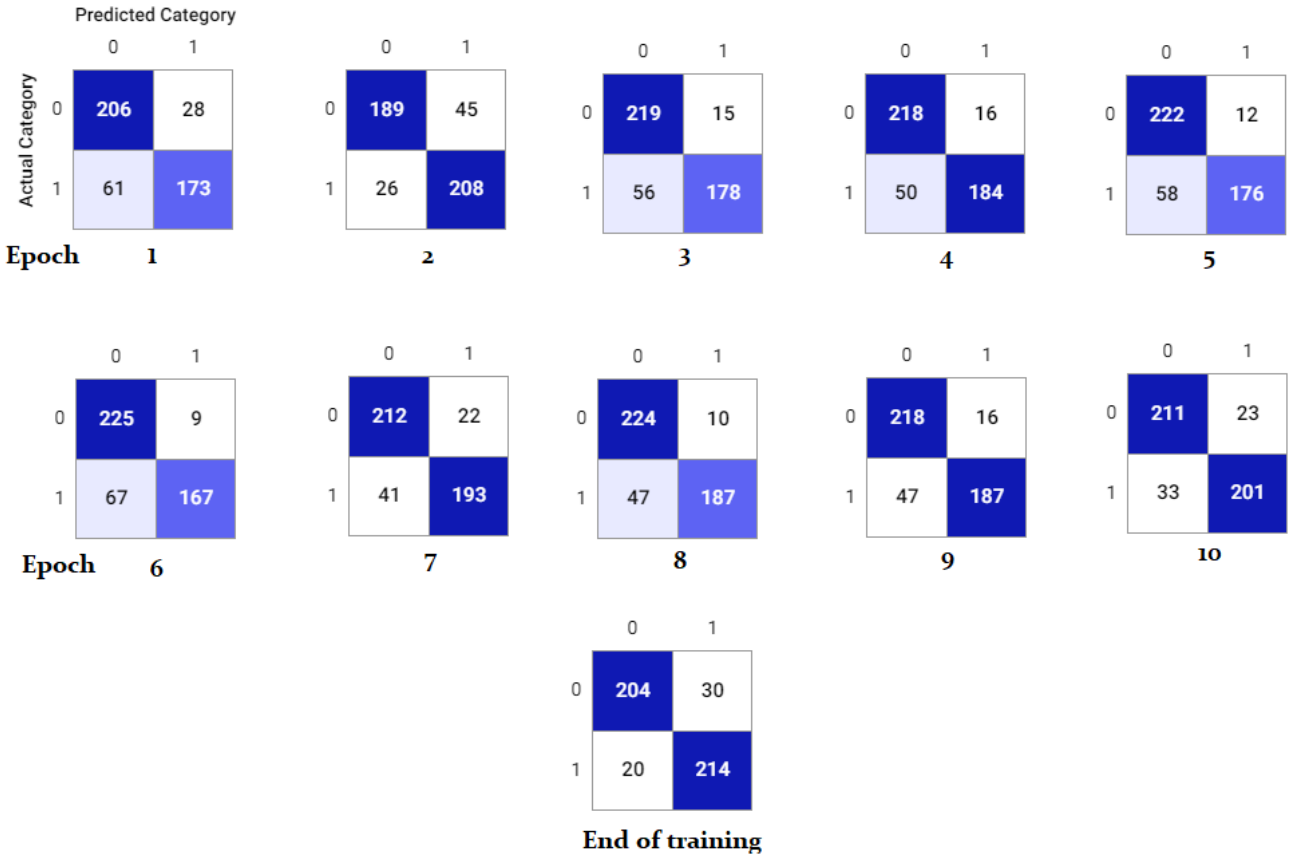


Figure 3.3: Confusion matrices of a 50 epoch run. Matrices taken from epoch 1 - 10 show a gradual increase in the total number of correctly predicted values. An imbalance between false positives and negatives show a possible bias toward positive predictions in the model during training despite the final confusion matrix showing a fairly balanced prediction rate.

References

Albawi, S., Mohammed, T. A., and Al-Zawi, S. (2017). "Understanding of a convolutional neural network". In: *2017 International Conference on Engineering and Technology (ICET)*. IEEE, pp. 1–6.

Bengio, Y. (2012). *Practical recommendations for gradient-based training of deep architectures*. arXiv: 1206.5533 [cs.LG].

Byrd, J. and Lipton, Z. C. (2019). *What is the Effect of Importance Weighting in Deep Learning?* arXiv: 1812.03372 [cs.LG].

Cilimkovic, M. (2015). Neural networks and back propagation algorithm. *Institute of Technology Blanchardstown, Blanchardstown Road North Dublin*. 15.

Gebel, Ł. (Nov. 2020). *Why We Need Bias in Neural Networks*. Available from: https://towardsdatascience.com/why-we-need-bias-in-neural-networks-db8f7e07cb98.

Gholamalinezhad, H. and Khosravi, H. (2020). *Pooling Methods in Deep Neural Networks, a Review*. arXiv: 2009.07485 [cs.CV].

Hansen, C. (Mar. 2020). *Neural Networks: Feedforward and Backpropagation Explained*. Available from: https://mlfromscratch.com/neural-networks-explained/#/.

Jin, X., Xu, C., Feng, J., Wei, Y., Xiong, J., and Yan, S. (2015). *Deep Learning with S-shaped Rectified Linear Activation Units*. arXiv: 1512.07030 [cs.CV].

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*. 60.6, pp. 84–90.

Paeedeh, N. and Ghiasi-Shirazi, K. (2020). *Improving the Backpropagation Algorithm with Consequentialism Weight Updates over Mini-Batches*. arXiv: 2003.05164 [cs.LG].

Pedamonti, D. (2018). *Comparison of non-linear activation functions for deep neural networks on MNIST classification task*. arXiv: 1804.02763 [cs.LG].

Ronaghan, S. (2021). *Deep Learning: Which Loss and Activation Functions should I use?* Available from: https://towardsdatascience.com/deep-learning-which-loss-and-activation-functions-should-i-use-ac02f1c56aa8.

Wood, T. (2021). *F-Score*. Available from: https://deepai.org/machine-learning-glossary-and-terms/f-score.

Bibliography

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. Adaptive Computation and Machine Learning series. MIT Press. Chap. 11, pp. 416–422. ISBN: 9780262035613. Available from: https://books.google.co.uk/books?id=Np9SDQAAQBAJ.