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

December 11, 2020

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 it's 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,

$$y = \Omega(\sum w_j x_j) \quad j = 0, 1, .., n.$$

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,

$$y_i = \Omega(\sum w_{j,i} x_j)$$

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 it's 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 it's 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.

1.2 Learning in Dense Networks

Typically, dense networks employ backpropagation for the purposes of learning. The network as described in the previous section is what is known as a feed forward network, information is passed and

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

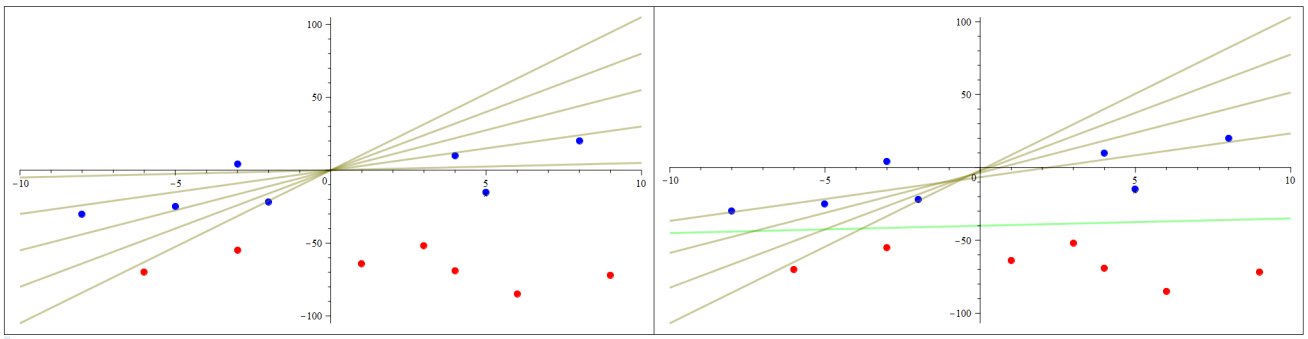

Figure 1.1: The graph on the left shows linear representation without a bias term, the model can not find the function as it is restricted to passing through the origin. On the right graph, a bias term is included that allows the linear function to be found.

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 to reduce the error and 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,

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

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,

$$w^k_{0,i} = b^k_i$$

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

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,

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

and by the chain rule,

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

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

$$y = \Omega(a_i^k)$$

$$\Rightarrow \frac{\partial C}{\partial a_i^k} = (y - \hat{y})\Omega'(a_i^k)$$

and,

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

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

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

Step 3: Back propagation to the hidden layer/s Extending this to further layers, by the chain rule, we have,

$$\frac{\partial C}{\partial a_i^k} = \sum_{r=1}^{r^k+1} \frac{\partial C}{\partial a_r^{k+1}}\frac{\partial a_r^{k+1}}{\partial a_i^k}$$

Where r denotes the number of neurons in the proceeding layer. The bias term is not dependant on previous layers so it is not included in this summation. It follows that,

$$\frac{\partial a_r^{k+1}}{\partial a_i^k} = w_{j,i}^{k+1}\Omega'(a_j^k)$$

$$\Rightarrow \frac{\partial C}{\partial a_i^k} = \Omega'(a_j^k)\sum_{r=1}^{r^k+1} w_{j,r}^{k+1}(y - \hat{y})\Omega'(a_j^{k+1})$$

and

$$\frac{\partial C}{\partial w_{j,i}^k} = \frac{\partial C}{\partial a_i^k}\frac{\partial a_i^k}{\partial w_{j,i}^k} = \Omega'(a_j^k)x_j^{k-1}\sum_{r=1}^{r^k+1} w_{j,r}^{k+1}(y - \hat{y})\Omega'(a_r^{k+1})$$

It is from here that the term backpropagation earns it's 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,

$$\Delta w_{i,j}^{k} = -\alpha[(y-\hat{y})\Omega'(a_1^k)x_j^{k-1} + \Omega'(a_j^k)x_j^{k-1}\sum_{r=1}^{r^k+1} w_{j,r}^{k+1}(y-\hat{y})\Omega'(a_r^{k+1})]$$

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 prime advantage of a CNN over other neural networks is the use of convolution, a linear operation used for feature extraction. 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 "kernal" (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,

$$\Omega(\sum_{i=0}^{n}(w_i x_i) + b) \ . \qquad i = 0,1,2,... \tag{2.1}$$

$$b = bias$$

$$w = weight$$

$$\Omega = activation\ function$$

[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.

Now consider the weight matrix, W, which consists of each weight element, w_i. In the case of a regular ANN, shown in (2), every input pixel is connected to each neuron with a different weight. However within a convolution layer (3), many of the neurons are disconnected and therefore have zero value, 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.

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

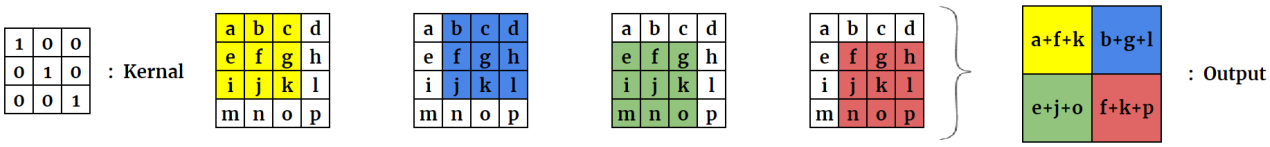

Figure 2.1: Kernal of order 3 acting on 4×4 image

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 in figure 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. 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.

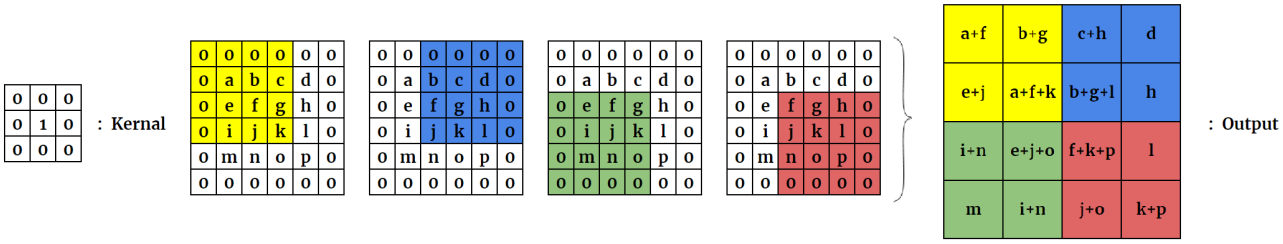

Figure 2.2: Kernal of order 3 acting on 4×4 image with zero-padding

2.1.4 Pooling

Pooling, much like valid padded convolution, is a process utilised to reduce the sampling rate of it's 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 it's 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, however.

2.1.5 Non-Linearity

First let us consider a simple example of a neural network foregoing the use of an activation function, shown in fig 4. Using the definition given for a neuron (1) and omitting the activation function, we can describe this network as,

$$y = h_2 \times w_3 + b_3$$

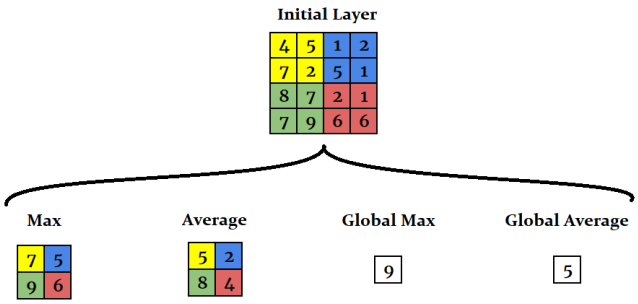

Figure 2.3: Depiction of different pooling outputs in regards to a sample initial layer

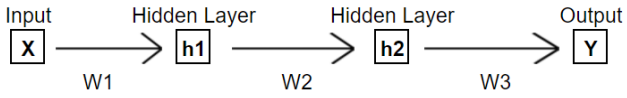

Figure 2.4: A simple neural network example.

$$= (h_1 \times w_2 + b_2) \times w_3 + b_3$$

$$= h_1 \times w_2 \times w_3 + b_2 \times w_3 + b_3$$

$$= (x \times w_1 + b_1) \times w_2 \times w_3 + b_2 \times w_3 + b_3$$

$$= x \times w_1 \times w_2 \times w_3 + b_1 \times w_2 \times w_3 + b_2 \times w_3 + b_3$$

$$= x \times w_1 \times w_2 \times w_3 + b_1 \times w_2 \times w_3 + b_2 \times w_3 + b_3$$

Now we can combine each of the bias terms 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,

$$y = x \times W + B.$$

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 and further, a multi-neuron model can be replaced by one with a single neuron thus adding more layers does not increase the efficacy of the model.

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, but we shall discuss but a few in order to highlight the properties that are important to note when choosing a function.

2.2 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 a positive values of the output 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[3] which means for 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 it's computational advantage over other activation functions. This is due to the fact that it does not activate every

[3] Also referred to as a vanishing gradient.

neuron simultaneously. Namely, if the linear transformation of the output is less than 0 then it is not activated (Pedamonti, 2018). Further, those that are activated do not then involve more complex computational methods such as division.

Another advantage in the function is that it is otherwise linear and is then easy to optimise using gradient-descent.

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 it's 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 to 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.

3 Results

3.1 Learning Rate Optimisation

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 backpropagation would slow down and could severely impact computational time. There is no sure-fire learning rate value for every model, it is dependant 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, it is clear that an optimal learning rate for the model is 7E-4.

3.2 Confusion Matrices

The confusion matrices show how the 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.

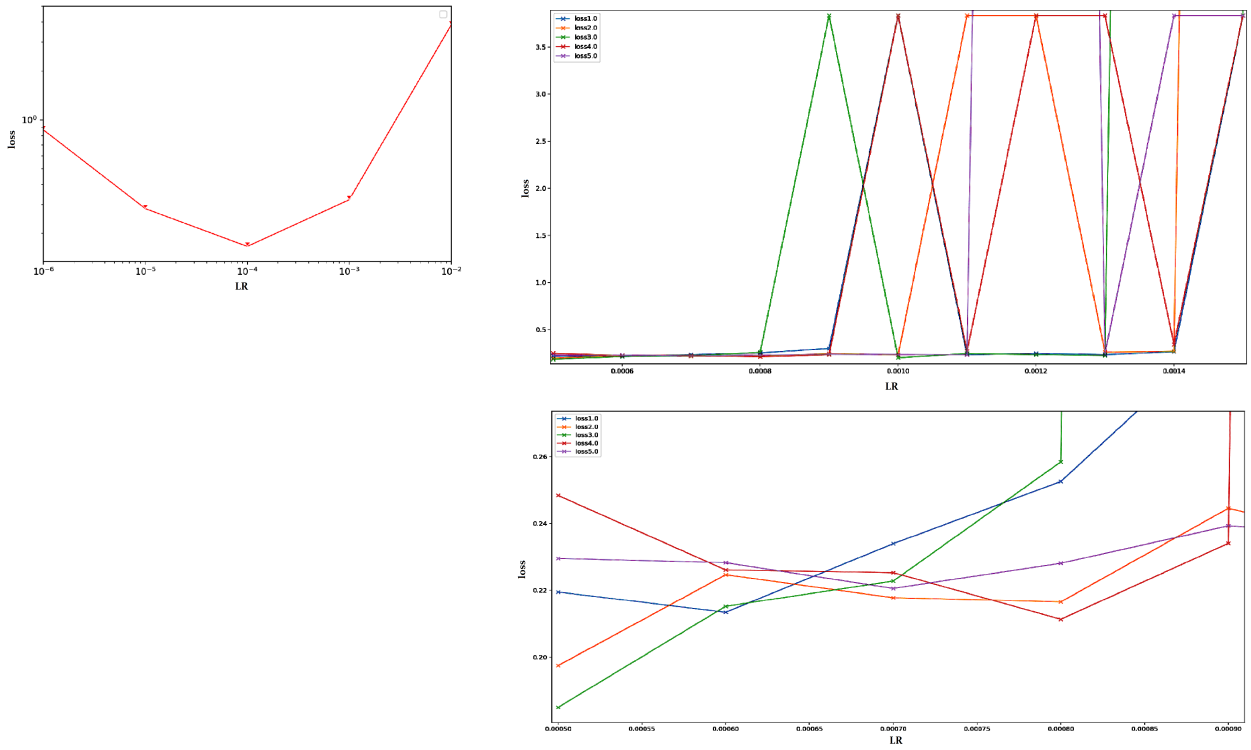

Figure 3.1: Loss vs Learning Rate of grid searches

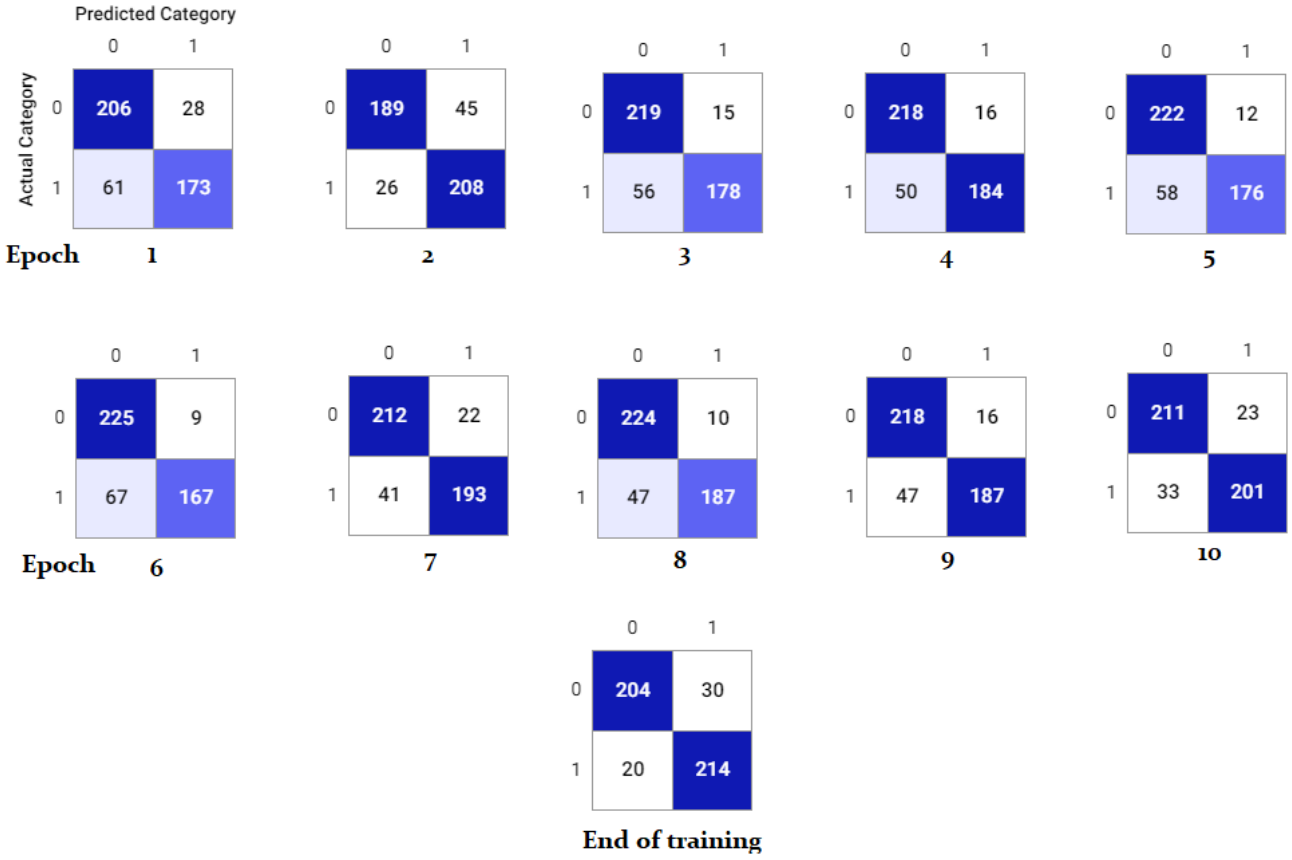

Figure 3.2: Confusion matrices of a 50 epoch run.

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].

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].

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.