Deep Learning for Medical Image Classification

March 12, 2020
Pavlos Demetriou - 201116459

Contents

Abstract

This review introduces some basic and more advanced concepts of Deep Learning, its structure and how it works. A huge sector that has been significantly advanced from deep learning is medicine and more specifically applications on medical imaging. We are going to present ideas and variations in order to create more efficient systems as well as how to apply them in medical imaging.

1 Introduction

1.1 What is Artificial Intelligence

Artificial Intelligence (AI) is a new filed in our world used for our own benefits, from security to healthcare. But what is Artificial Intelligence? AI is a computer system that imitaes the way our brain and neurons work, in order to learn patterns and various human-like tasks. In a more detailed definition, according to *Patterson Dan W.* book named *Introduction to Artificial Intelligence and expert systems* [2, p. 2] ”AI is a branch of computer science concerned with the study and creation of computer systems that exhibit some form of intelligence systems that learn new concepts and tasks. Systems that can understand a natural language or perceive and comprehend a visual scene and perform tasks that require human types of intelligence”. Artificial Intelligence is a very broad set of subjects with a lot of subsets.

1.2 Machine Learning, Deep Learning and its applications in Medicine

We will focus on *Machine Learning* and more specifically in *Deep Learning*. *Machine Learning* is a set of statistical algorithms that is able to learn computer systems patterns from data. *Deep Learning* is one machine learning model that uses neurons to build a network called *neural network* , which means it uses multiple layers of neurons in order to achieve accuracy finding the patterns. A key difference is that in this case a labeled datasets is not necessary in order to make a classification. Artificial intelligence and especially deep learning are very popular in medical purposes, as applications appears in every aspect of the field. In the past few years, according to recent studies, the transition into the computer-based systems seems to be very promising and effective. One application of deep learing is in mammography analysis. Breast cancer is the most common cause of death in women. Usually mammograms are analysed by human radiologists with help from Computer-aided detection (CAD) programs. According to the article *Detecting and classifying lesions in mammograms with Deep Learning* by *Dezs Ribli, Anna Horvth, Zsuzsa Unger, Pter Pollner and Istvn Csabai*, since 2012, deep *convolutional neural networks* (which we are going to discuss in detail) have improved the performance in mammogram analysing overcoming the human performance in medical image classification although more research is needed on the topic [15]. Machine learning can be applied to classify and detect complicated patterns in medical images with a higher effective rate than humans, such as in MRI radiotherapy and many more. For example, Electrinoc Health Records (EHR) is an electronic version of a patient’s medical history including important data such as medications, reports, progress done, medical images etc. Radiologists are responsible to examine patients’ medical images in order to detect any signs of threat. Unfortunatly, the human factor is limiting the process speed, since humans need money and a lot of years to be trained up to the point where they can decide correctly upon a patient’s case. For this reason, some health-care systems prefer to send their radiology reports to lower-cost countries such as India via tele-radiology. We can already see problems arising due to the human factor as this delay and possible false alanysis provides wrong diagnosis which can cause great harm in the patients' health [12, p.1]. This is where machine learning and deep learning algorithms are used to prevent those errors and speed up the process of medical imaging. In order to understand how these topics are related and how are they used in medical imaging, some mathematical knowledge is required together with some programming techniques which are going to be discussed below.

2 Supervised and Unsupervised Learning

Machine learning is divided in three categories, *supervised learning*, *unsupervised learning* and *reiforcement learning*. In supervised learning the model uses a dataset that is already labeled which means that every sample is prior assigned to a label. The model has to produce prediction labels on new data using the patterns that the model discovered during the training phase with two main applications. *Classification*, when our data needs to be categorised and *regression* when we need to predict a numerical value based on our input data. The key difference of unsupervised learning is that it uses a non-labeled dataset in order to discover patterns and a structure of the data. We can apply unsupervised laerning to *data clustering* where we can divide our dataset into groups that have similar features in order to create a visual representation of the grouped data. Another application is *dimensionality reduction* where the dimension is the number of features necessary to describe a single data. This process neglects some of features and keeps the features that are most reliable and compresses them into principal values with a small loss of the data, in order to accelerate the process of the algorithms. Last category is *reinforcment learning* where the algorithm is using trial and error in order to reach its goal. The *agent* receives information in form of numerical data which is called *state* and the agent receives a *reward* or a *penalty* depending of the action that the agent took. The result can be used as a feedback to imporove and update the parameters of the agent. This method is often used in navigation, robotics and gaming [3].

3 Supervised Learning

3.1 Perceptron

Before we go into multilayer neural networks, we have to learn about the *perceptron* (Fig. 1). A *perceptron* is a single-layer neural network. It is used as a binary classifier and it interprets as a single neuron. It performs dot product on a weight vector **w** (3.1) with random and unbalanced weights and a vector **x** (3.1) (input vector), adding a bias $\mathbf{w_0}$. It sums all the dot products and the result is fed into the neuron using an *activation function*, commonly a *step function* for a perceptron (3.2). The output value **y** is interpreted as probability.

$$\mathbf{w} = \begin{bmatrix} w_1 & w_2 & w_3 & ... & w_n \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 & ... & x_n \end{bmatrix} \tag{3.1}$$

$$f(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases} \qquad y = f(\sum_i^n w_i \cdot x_i + w_0) \tag{3.2}$$

For example, we can create a perceptron model that learns the **AND** gate. The model for AND gate is shown in Figure 2. The goal of this model is to learn that when the inputs are 1 and 1 then the output is also 1 and 0 otherwise. We set the random weights to $w_1, w_2 = 1, 1$ and the bias to $w_0 = -1$ for the ease of the example such that they do not need to be tuned. the process of tuning the parametes will be discussed later on.

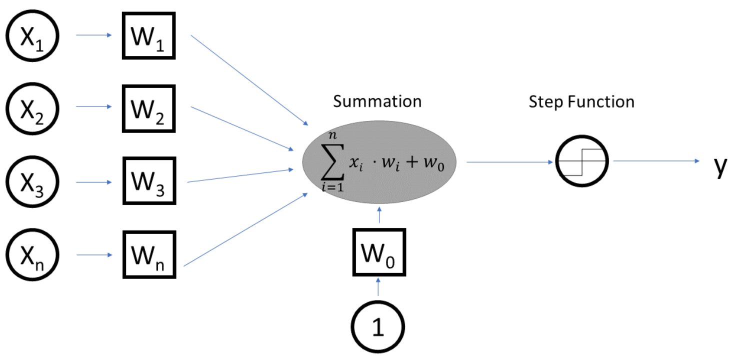
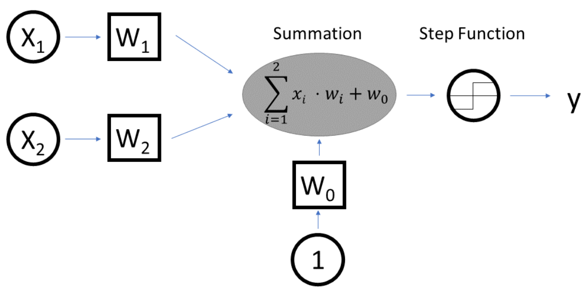


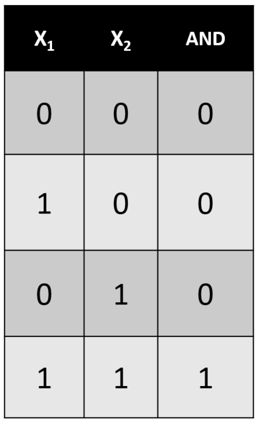
Figure 1: Perceptron Diagram

To test the model we input our data testing the 4 cases.

1. $x_1, x_2 = 0, 0 \implies x_1 \cdot w_1 + x_2 \cdot w_2 + w_0 = 0 \cdot 1 + 0 \cdot 1 + -1 = -1 \implies f(-1) = 0$
2. $x_1, x_2 = 1, 0 \implies x_1 \cdot w_1 + x_2 \cdot w_2 + w_0 = 1 \cdot 1 + 0 \cdot 1 + -1 = 0 \implies f(0) = 0$
3. $x_1, x_2 = 0, 1 \implies x_1 \cdot w_1 + x_2 \cdot w_2 + w_0 = 0 \cdot 1 + 1 \cdot 1 + -1 = 0 \implies f(0) = 0$
4. $x_1, x_2 = 1, 1 \implies x_1 \cdot w_1 + x_2 \cdot w_2 + w_0 = 1 \cdot 1 + 1 \cdot 1 + -1 = 1 \implies f(1) = 1$



(a) AND Gate Model



X_1	X_2	AND
0	0	0
1	0	0
0	1	0
1	1	1

(b) AND Table

Figure 2: AND Gate Perceptron

Which is indeed the table of AND gate. A disadvandage of using the perceptron model is that it can only be used is very specific situations where the classification can be achieved linearly (sepratating the two classes with a line). [5, p.3]. This problem can be solved by introducing a *multilayer neural network*.

3.2 Artificial Neural Networks

If we add more layers in the perceptron model then we create a multilayer network which is called a *Multilayer Perceptron (MLP)* or an *Artificial Neural Networks (ANN)*. ANN is a machine learning model that consists of neurons that are arranged in layers. All units (neurons) in a network commute with each other (fully connected) exept neurons in the same layer. The basic structure of a neural network consists of an input layer, the hidden layer/s and the output layer. One advandage having a multilayer network is that it can classify data that cannot be classified with a single line. In 1987 Lippmann in his paper *An introduction to computing with neural nets* found that an MLP with two hidden layers can create classifications on any desired shape but he didn't mention how many neurons to include in each layer. [14]

The *universal approximation theorem* states that:

A feedforward network with a linear output layer and at least one hidden layer with any squashing activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable [1] *function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units.*

One should ask how many neruons in each layer should be. It is clear that the input layer has neurons equal to the number of input variables that are fed into the network. As for the hidden layer/s things get a little more complicated. First of all, hidden layers are required iff (if and only if) the data must be separeted non-linearly [16, p.99]. The more layers are added to the network the harder it is to be trained and optimized. There is not a cetrein way to find the number of layers or neurons for the hidden layers so the structure of the network must be found by experimentation. The above theorem tells us that the network will be able to represent this function but this does not mean that the training algorithm will be able to train the function and find the correct weights. This could happen due to *overfitting* that we are going to discuss later on [6].
At last, the output layer should have neurons equal to the number of classes that we want where we will get our prediction output. The labled input data are fed into the network through the input layer and they change accordingly when passing through the hidden layers. Finally, the output layer will give predictions about the label of the sample data. The network is fed with a big amount of data called *training data* in order to train the model using appropriate algorithms. After the training process the network should have identifed patterns and make precise predictions on unseen data.
There are a lot of different kinds of ANN but we will consider the basic form, *feedforward neural networks*.

[1] Any continuous function on a closedd and bounded subset of $\mathbb{R}^n$ is Borel measurable.

3.2.1 Feedforward Neural Netwoks

A *feedforward neural network* is a map $y = f(x; w)$ that trains the parameters w in order to get the best approximation from the input **x**. They are called feedforward because the data only flows in one direction and there are no feedback connections which means that the outputs are not fed again into the model. Such networks are called *recurrent neural networks*. The main feature of this networks are the composition of functions since the deep structure (more than 1 hdden layer) [6].
The input data are fed into the network throught the input layer and they change accordingly passing through the hidden layers. Each neuron in the hidden layer can be seen as a perceptron model where the input data is the output values from the previous layer. A key difference in a deep model is that the activation function used for the neurons is not a step-function but usually a *sigmoid function*, $y = \sigma(\lambda) = \frac{1}{1+e^{-\lambda}}$ (see Figure 3). Since our model consists of more than one hidden layer we have to introduce new formulas. Let l be the l-th layer of the network and i be the i-th neuron of the layer. Now the equations used for a perceptron model can be transformed into:

$$\lambda_i^{(1)} = w_0^{(1)} + \sum_j w_{ij}^{(1)} \cdot x_j \tag{3.3}$$

for the first layer. The inputs x_j are multiplied by the correspoding weights w_{ij}, summed up adding the bias w_0^1 of the first layer and then fed into the activation function. In general for layer *l*, the output of a node is

$$y_i^{(l)} = \sigma(\lambda_i^{(l)}) = \frac{1}{1 + e^{-\lambda_i^{(l)}}} = \frac{1}{1 + e^{-(w_0^l + \sum_j w_{ij}^{(l)} \cdot y_j^{(l-1)})}} \tag{3.4}$$

Since the network consists of multiple hidden layers, equation (3.4) can be stacked adding more layers expressing the output expected value through the following composition function (the biases are neglected for the simplicity of the equation).

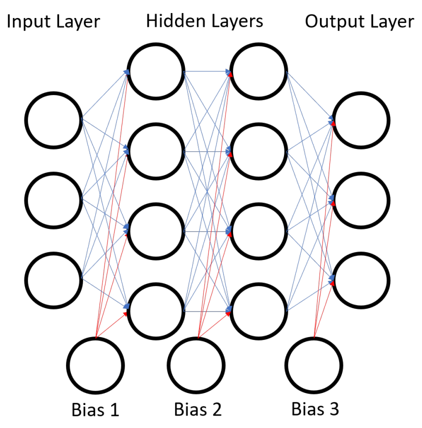
$$y_k = \sigma^{(l)}(\sum_p w_{kp}^{(l)} \sigma_p^{(l-1)}(\sum_m w_{lm} \sigma_m^{(l-2)}(...\sigma^{(1)}(\sum_j w_{ij}^{(1)} x_j)))) \tag{3.5}$$

[4, Section 1.2]

The input data are fed into the network giving an expected value of y_k. Before the model is able to make accurate preditions a *training set* must pass through the model in order to adjust the random weights and train the model to detect patters using certain training methods. We will discuss the method of training the model later on.

3.3 Convolutional Neural Networks

An image can be considered as a matrix of pixel values. When analysing an image, the input size is very large and that causes problems to the model. For example, consider an image with size 100x100 pixels, then we would have 10000 input values for the input layer. If we use a feed forward neural network then all neuron are connected with the neurons of the previous layer leading to each neuron having 10000 parameters. This is very inefficient as it is very time-consuming and computationaly expensive having an extremly big amount of parameters. As said in the article *An introduction to Convlolutional Neural Networks* by *Keiron O'Shea* and *Ryan Nash*, another problem that the feed forward neural network will face is again overfitting [8].



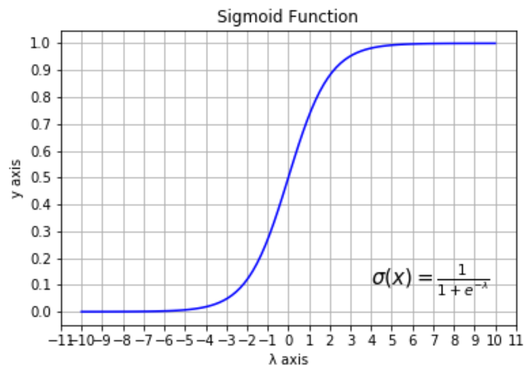


Figure 3: A neural network with 2 hidden layers on the left and the Sigmoid function on the right

Instead, we use *Convolutional Neural Networks* (CNNs) where neurons are not all connected with each other making easier the computational process. CNNs are a type of ANNs with the difference that each neuron has less connections as inputs. CNNs are structured in three type of layers, *convolutional layers*, *pooling layers* and *fully-connected layers*. The image is arranged in a grid structure creating a matrix with values (pixel values).

3.3.1 Convolutional Layers

A convolutional layer or *filter*, is usually a 3x3 matrix with random weights. This filter is place in the upper left corner of the image and it performs dot-product with the upper left 3x3 matrix of the image giving a scalar output which describes how well does that filter is represented in that position. The scalar value is stored in a tensor. This scalar product is done for every possible position on the image, and this is called *convolution* creating a map that represents where that filter occurs on the image. When all the filters are applied on the image it creates a set of tensors. Then follows an activation function usually a **Rectified Linear unit** (ReLu) which is defined as ReLU(x) = max(0,x) which is used to introduced nonlinearity. This activation function basically translates all the negative values to zero. Feeding the tensors in the activation functions produces new tensors, the *feature maps* (see Figure 5 as an example).

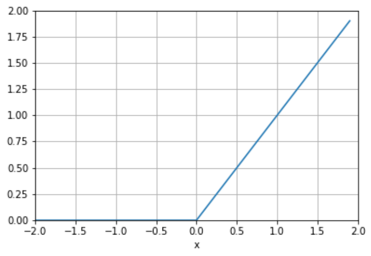


Figure 4: Rectified Linear unit (ReLu)

3.3.2 Visualisation of Filters using Python

For this part we are going to use the dataset *Chest X-Rays for Pneumonia* (the model used for this classification can be seen in Section 8 as well as the summary of the model in Fgure 14). First, we have to check each layer name and sum up only the convolutional layers. Each convolutional layer contains the filters and the bias values. We store the filters and the biases via *kayer.get_weights()* and we print all the layers that we stored together with the shape of the filters.

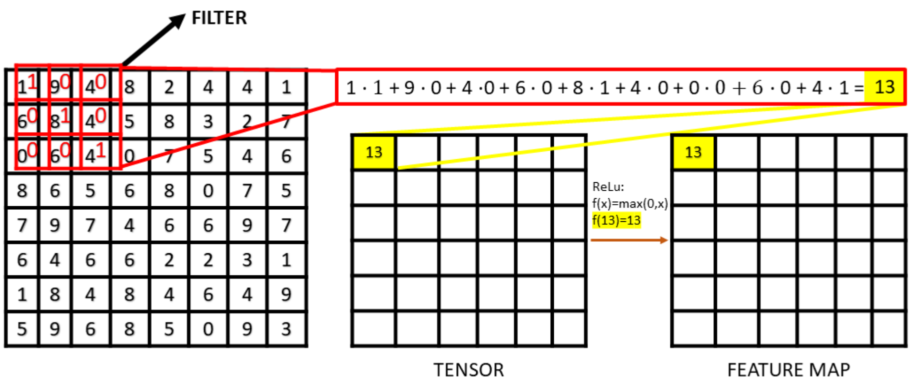


Figure 5: Convolution Process

```
for layer in model.layers:
  if 'conv' not in layer.name:
    continue
  filters, biases = layer.get_weights()
  print(layer.name, filters.shape)
```

We can see that inside the model we have three convolutional layers. The first one consists of 32 3 x 3 filters with depth 3 (three channels one for each colour red, green and blue that correspond to the three channels of the input image).

```
conv2d_1 (3, 3, 3, 32)
conv2d_2 (3, 3, 32, 32)
conv2d_3 (3, 3, 32, 64)
```

We normalize the values of the filters between the range (0,1) to be easier to visualize.

```
f_min, f_max = filters.min(), filters.max()
filters = (filters - f_min) / (f_max - f_min)
```

We can visualize the first 5 filters out of the 64 filters of the first layer using the following code presenting the three channels as columns.

```
n_filters = 5
ix = 1
for i in range(n_filters):
  f = filters[:, :, :, i]
  for j in range(3):
    ax = plt.subplot(n_filters, 3, ix)
    ax.set_xticks([])
    ax.set_yticks([])
    plt.imshow(f[:, :, j],cmap ='gray')
    ix = ix + 1
plt.show()
```

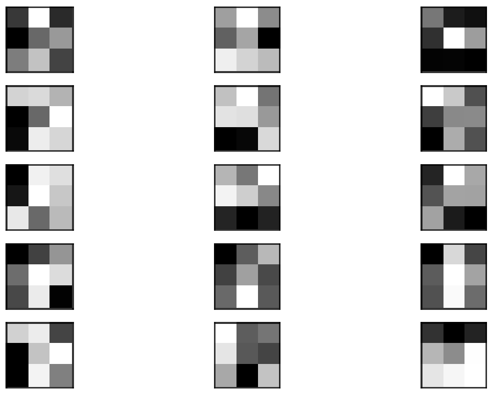


Figure 6: First 5 filters of 1st Conv. Layer with their three channels

Dark colour represents small value of the weight and light colour represents a large value.

3.3.3 Visualisation of Convolution using Python

The next step is to visualize the feature map created actere the application of the filter on the image. This will help us understand the features that the filter extract and detect. First lets plot the first image of the dataset.

```
img = x[0]
plt.imshow(img,cmap=plt.cm.binary)
```

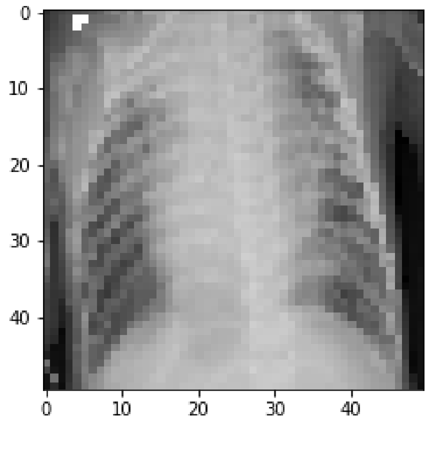


Figure 7: Chest X-Ray

We expand the dimensions so that it represents a singe example and we scale the pixel values. We create a model that outputs a feature map from the first convolutional layer and we predict the output.

```
img = expand_dims(img, axis=0)
img = preprocess_input(img)
convmodel = Model(inputs=model.inputs, outputs=model.layers[0].output)
feature_maps = convmodel.predict(img)
```

We know that we are going to get 32 feature maps that have been convoluted with the filters and we can visualize the outcome using the following code.

```
plt.figure(figsize=(15,15))
ix = 1
for _ in range(8):
   for _ in range(4):
      ax = plt.subplot(8, 4, ix)
      ax.set_xticks([])
      ax.set_yticks([])
      plt.imshow(feature_maps[0, :, :, ix-1], cmap='gray')
      ix += 1
plt.show()
```

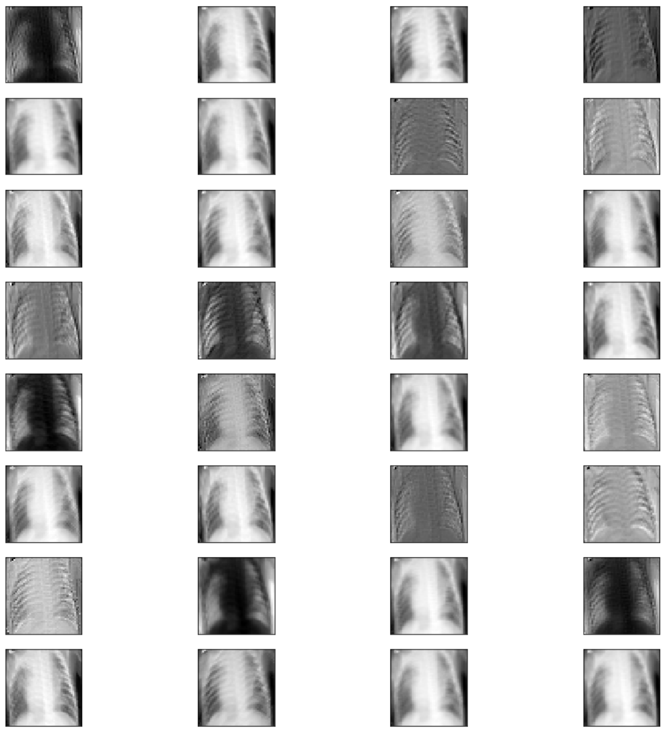
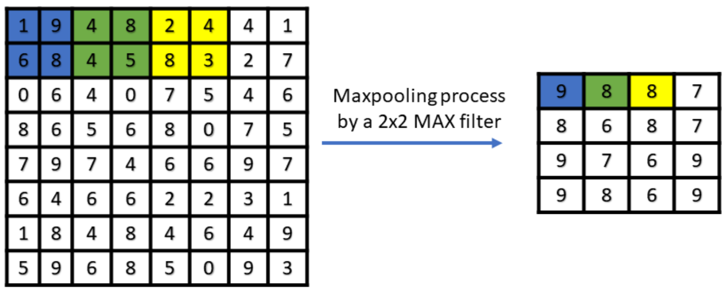


Figure 8: Chest X-Ray after the first convolution with the 32 filters

3.3.4 Pooling Layers

Each feature map is then passed through the pooling process where it is pooled by a pooling layer. This operation takes a matrix block (usually 2x2 or 3x3) of the upper left corner of the feature map and outputs a single value using the function MAX or the average number of the matrix performing downsampling. It outputs the maximum or the average value of the matrix in a new tensor. The pooling layer moves 2 or 3 steps to the next block accodingly and applies the function for the rest of the image. This is done for all the feature maps reducing the dimension of the data without any significant loss reducing the computations that will follow. For the example shown before the pooling proess is show in Figure 7.
Convolutional layers with the activation functions and pooling layers can be stack multiple times before concluding to a fully connected layer to create a stronger and more accurate network [8, pp, 3-5]. This is common in a CNN architecture since just one cycle of these opeations are not enough to create a stable and accurate model. The image gets even more filted as it passes through the the convolution layers and the dimension of hte feature maps also reduce as they pass through the pooling process. Also pooling process is a useful technique since it helps the outcome to be *invariant* under small translations of the input. Which means that if the input is slightly distorted then the pooled values won't have a significant change. Under this consideration, we do not care if a feature is in a certain position but instead we just need to know that the feature is in the image [6].



$$\begin{bmatrix} 1 & 9 & 4 & 8 & 2 & 4 & 4 & 1 \\ 6 & 8 & 4 & 5 & 8 & 3 & 2 & 7 \\ 0 & 6 & 4 & 0 & 7 & 5 & 4 & 6 \\ 8 & 6 & 5 & 6 & 8 & 0 & 7 & 5 \\ 7 & 9 & 7 & 4 & 6 & 6 & 9 & 7 \\ 6 & 4 & 6 & 6 & 2 & 2 & 3 & 1 \\ 1 & 8 & 4 & 8 & 4 & 6 & 4 & 9 \\ 5 & 9 & 6 & 8 & 5 & 0 & 9 & 3 \end{bmatrix} \longrightarrow \begin{bmatrix} 9 & 8 & 8 & 7 \\ 8 & 6 & 8 & 7 \\ 9 & 7 & 6 & 9 \\ 9 & 8 & 6 & 9 \end{bmatrix}$$

Figure 9: Pooling Layers

3.3.5 Fully-Connected Layers

The final step consists of the *fully connected layers* that will make the predictions for classification. The last outputs from the pooling layers are some useful features that are extracted from the convolution and pooling process. They are flattened and presented as a single vector. From this point follows a familiar multilayer neural network to make the classification of the image that was input by finding the probabilities . ReLu are used between these layers to imporve the performance of the network by learning non-linear compinations of the features. At the output layer an activation function called **softmax** is applied to scale the output values between 0 and 1 in order to be considered as probabilities, giving a sum of one. The class with the biggest probability outcome is the class that the neural network predicted.

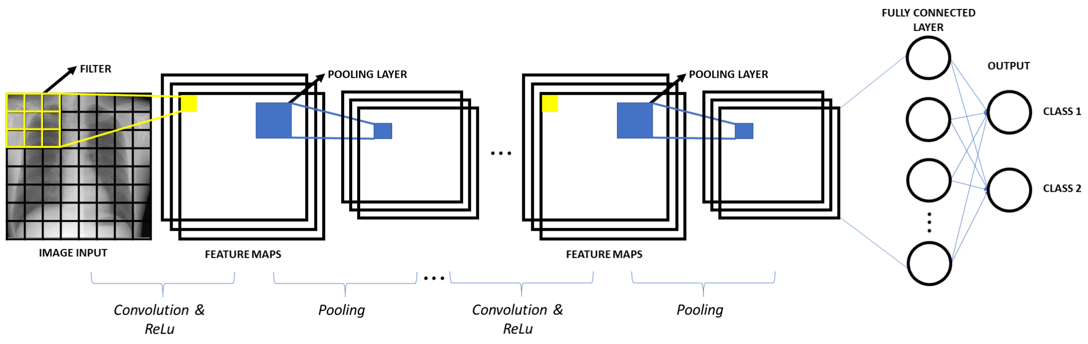


Figure 10: Convolutional Neural Network Architecture

The training method is still the same, we can use gradient descent and modify the backpropagation algorithm since the neurons are not fully connected. Neurons that are not connected with the next layer have weights of zeros and therefore we do not need to compute the gradient and change its values [7, p.3].

4 Backpropagation Method

During the training phase the weights of the model are tuned until the model can predict accuratly outcomes on unseen data. The network is fed with the training data while in each iteration the weights and the outcomes are recorded. In each iteration the outcome is compared with the actual labels using an *objective function* (error function). *Backpropagation method* is used to compute the gradient of the error function and afterwards *gradient descent* comes to minimise this loss function and update the values of the weights. Backpropagation was first introduced in 1970, but the first publication was done by *David Rumelhart Geoffrey Hinton and Ronald Williams* in 1986 at their article *Learning representations by back-propagating errors*. by An important point is that the error function must be continuous and differentialble because we have to differentiate the function and we don't want the function to blow up. Some of the activation functions that could be used are:

- *Linear Activation Function* $y = cx$ (where c is some constant)
- *Sigmoid Activation Function* $\sigma(x) = \frac{1}{1+e^{-x}}$
- *Sigmoid Symmetric Activation Function (hyperbolic tangent)* $y = \tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- *Elliot Activation Function* $\sigma(x) = \frac{1}{1+|x|}$

An experiment was done by *P.Sibi, S.Allwyn and P.Siddarth* students at SASTRA University to test the precision of 12 types of activation functions during the training phase using a mushroom data. The network had to detect whether a mushroom was edible or poisonous based on 22 observable features shown on Figure 9.

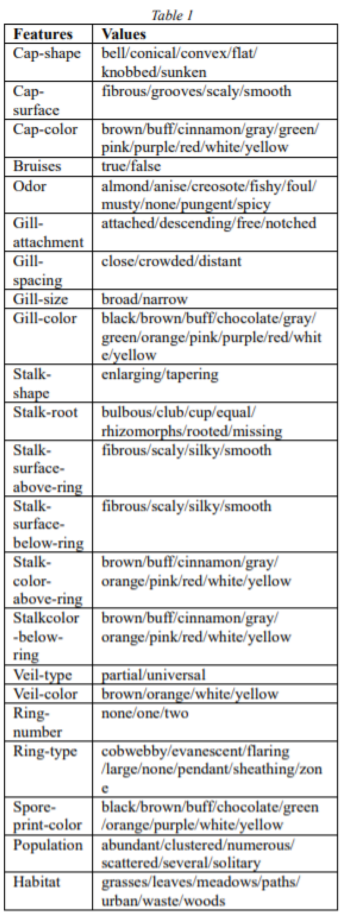


Table 1

Features	Values
Cap-shape	bell/conical/convex/flat/ knobbed/sunken
Cap-surface	fibrous/grooves/scaly/smooth
Cap-color	brown/buff/cinnamon/gray/green/ pink/purple/red/white/yellow
Bruises	true/false
Odor	almond/anise/creosote/fishy/foul/ musty/none/pungent/spicy
Gill-attachment	attached/descending/free/notched
Gill-spacing	close/crowded/distant
Gill-size	broad/narrow
Gill-color	black/brown/buff/chocolate/gray/ green/orange/pink/purple/red/whit e/yellow
Stalk-shape	enlarging/tapering
Stalk-root	bulbous/club/cup/equal/ rhizomorphs/rooted/missing
Stalk-surface-above-ring	fibrous/scaly/silky/smooth
Stalk-surface-below-ring	fibrous/scaly/silky/smooth
Stalk-color-above-ring	brown/buff/cinnamon/gray/ orange/pink/red/white/yellow
Stalkcolor -below-ring	brown/buff/cinnamon/gray/ orange/pink/red/white/yellow
Veil-type	partial/universal
Veil-color	brown/orange/white/yellow
Ring-number	none/one/two
Ring-type	cobwebby/evanescent/flaring /large/none/pendant/sheathing/zon e
Spore-print-color	black/brown/buff/chocolate/green /orange/purple/white/yellow
Population	abundant/clustered/numerous/ scattered/several/solitary
Habitat	grasses/leaves/meadows/paths/ urban/waste/woods

(a) $Features$

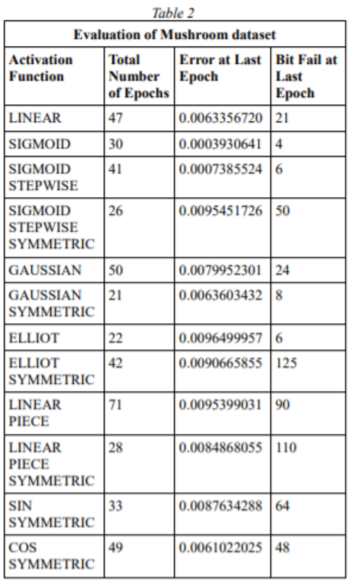


Table 2

Evaluation of Mushroom dataset			
Activation Function	**Total Number of Epochs**	**Error at Last Epoch**	**Bit Fail at Last Epoch**
LINEAR	47	0.0063356720	21
SIGMOID	30	0.0003930641	4
SIGMOID STEPWISE	41	0.0007385524	6
SIGMOID STEPWISE SYMMETRIC	26	0.0095451726	50
GAUSSIAN	50	0.0079952301	24
GAUSSIAN SYMMETRIC	21	0.0063603432	8
ELLIOT	22	0.0096499957	6
ELLIOT SYMMETRIC	42	0.0090665855	125
LINEAR PIECE	71	0.0095399031	90
LINEAR PIECE SYMMETRIC	28	0.0084868055	110
SIN SYMMETRIC	33	0.0087634288	64
COS SYMMETRIC	49	0.0061022025	48

(b) $Results$

Figure 11: Activation Function Experiment [13, pp. 1267]

The results show that there is not a significant difference between the functions used in terms of error. Moreover it is shown up to which point each activation function creates small errors in the last iteration and the sigmoid function after 30 iterations has an error at the last iteration of E=0.0003930641. In general, every activation function has small error and if a network is trained properly with a particular activation function then it doesn't matter which one we choose. The activation function chosen for the neurons is important but there are other factors that has to be considered carefully during the training phase that play an important role in the accuracy of the training that we will discuss later on [13, pp. 1266-1268]. Since the error function consist of the activation function, we have to use a continuous and differentiable activation function, so we can choose the sigmoid function.

We want to minimise the difference between the predicted value and the true value as much as possible by minimising the cost function. We define the *cost function*

$$E = \frac{1}{2}\sum_{j=1}^{J}(p_j - y_j)^2 \tag{4.1}$$

with p_j to be the predicted j^{th} output and y_j the true output of the j^{th} input (the factor $\frac{1}{2}$ will vanish when we differentiate). We sum all the errors from all J output nodes as the *Total Error*. Backpropagation calculates the gradient with respect to every weight

$\nabla E = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, ..., \frac{\partial E}{\partial w_n})$ using the *chain rule*. The numerical value of this calculation tells us how much each weight must be adjusted in order to minimise E. We define the weights $w_{kj}^{(l)}$ for layer *l* going from node *j* to node *k* in the next layer.

First, lets denote x_j to be the inputs with the weighted sum for the 1^{st} hidden layer

$$\lambda_k^{(1)} = w_0^{(1)} + \sum_j w_{kj}^{(1)} \cdot x_j \tag{4.2}$$

and the activation value for the 1^{st} layer.

$$y_k^{(1)} = \sigma(\lambda_k^{(1)}) = \frac{1}{1 + e^{-\lambda_k^{(1)}}} = \frac{1}{1 + e^{-(w_0^{(1)} + \sum_j w_{kj}^{(1)} \cdot x_j)}} \tag{4.3}$$

If we have *l* number of layers then the weighted sum of the inputs of the k^{th} node of the l^{th} layer will be

$$\lambda_k^{(l)} = w_0^{(l)} + \sum_j w_{kj}^{(l)} \cdot y_j^{(l-1)} \tag{4.4}$$

In general, the output of the k^{th} node of the l^{th} layer after the activation fucntion is

$$y_k^{(l)} = \sigma(\lambda_k^{(l)}) = \frac{1}{1 + e^{-\lambda_k^{(l)}}} = \frac{1}{1 + e^{-(w_0^{(l)} + \sum_j w_{kj}^{(l)} \cdot y_j^{(l-1)})}} \tag{4.5}$$

[19]
We use the sigmoid function with derivative $\sigma'(x) = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1 \cdot e^{-x}}{(1+e^{-x})(1+e^{-x})} = \sigma(x) \cdot \frac{e^{-x}}{(1+e^{-x})} = \sigma(x) \cdot \frac{1+e^{-x}-1}{(1+e^{-x})} = \sigma(x) \cdot (\frac{1+e^{-x}}{(1+e^{-x})} - \frac{1}{(1+e^{-x})}) = \sigma(x) \cdot (1 - \sigma(x))$ [17]

By chain rule we have:

$$\frac{\partial E}{\partial w_{kj}^{(l)}} = \frac{\partial E}{\partial y_k^{(l)}} \frac{\partial y_k^{(l)}}{\partial \lambda_k^{(l)}} \frac{\partial \lambda_k^{(l)}}{\partial w_{kj}^{(l)}} \tag{4.6}$$

If we break down the chain rule we get:

$$\frac{\partial \lambda_k^{(l)}}{\partial w_{kj}^{(l)}} = y_j^{(l-1)} \tag{4.7}$$

$$\frac{\partial y_k^{(l)}}{\partial \lambda_k^{(l)}} = y_k^{(l)} \cdot (1 - y_k^{(l)}) \tag{4.8}$$

The first part of the chain rule $\frac{\partial E}{\partial y_k^{(l)}}$ can be easily calculated when *l* is the final layer (output layer).

$$\frac{\partial E}{\partial y_k} = (y_j - p_j) \tag{4.9}$$

If *l* is not the final layer then we have to look for the error in the hidden layers

$$\frac{\partial E}{\partial y_k^{(l)}} = \sum_i \frac{\partial E}{\partial y_i^{(l)}} \frac{\partial y_i^{(l)}}{\partial \lambda_i^{(l)}} \frac{\partial \lambda_i^{(l)}}{\partial y_k^{(l)}} \tag{4.10}$$

Now if we combine equations (5.7), (5.8), (5.10) we get that the gradient of the cost function with respect to weight $w_{kj}^{(l)}$ is

$$\frac{\partial E}{\partial w_{kj}^{(l)}} = y_j^{(l-1)} y_k^{(l)} (1 - y_k^{(l)}) \sum_i \frac{\partial E}{\partial y_i^{(l)}} \frac{\partial y_i^{(l)}}{\partial \lambda_i^{(l)}} \frac{\partial \lambda_i^{(l)}}{\partial y_k^{(l)}} \tag{4.11}$$

[19]

4.1 Gradient Descent

Gradient Descent now comes and updates the weights in order to minimise the cost function. In each iteration the weights are updated in the opposite direction of the gradient of the objective function. The gradient always shows the direction to the nearest local minimum. One of the most important hyperparameters is the *learning rate* η which is the step size that the weights are going to be updated towards the direction of the local or global minimum that we aim to reach. This hyperparameter is often betwenen 0 and 1, although the exact value will be defined by experimentation. The learning factor must be chosen carefully in order to train the model correctly. If a big learning rate is chosen then weights change in a faster rate which may lead to divergence and inaccurate predictions of the model. Choosing a small learning rate requires more iterations since the rate at which the weights change is small and it can result in failure to complete the training phase. A good approach is to start with a relatively large learning rate and reduce its value with each step. Recent studies suggest that the use of a non-constant learning rate as the iterations increased is more efficient.[21] As quoted in the book *Deep Learning " the learning rate is perhaps the most important hyperparameter. If you have time to tune only one hyperparameter, tune the learning rate."* [6]

There are three different gradient descent techniques that we are going to discuss.

4.1.1 Batch Gradient Descent

Batch Gradient Descent calculates the gradient of the cost function for all the training data points and updates the values of the weights using the following algorithm with *n* to be the stage of iteration.

$$w_{kj}^{(l)}(n+1) = w_{kj}^{(l)}(n) - \eta \frac{\partial E}{\partial w_{kj}^{(l)}} \tag{4.12}$$

This method uses a fixed learning rate and we do not have to take care of the learning rate decay. A disadvantage is that it goes through the whole dataset making the training process slower especially with large datasets. [20]

4.1.2 Stochastic Gradient Descent

Stochastic Gradient Descent randomly picks one data from the dataset (x_i, y_i) at a time and calculates the gradient and the update of the weights using the algorithm mentioned above. Due to the randomness and to the frequent adjustments it causes the cost function to fluctuate. [20]

4.1.3 Mini-Batch Gradient Descent

Finally, *Mini-Batch Gradient Descent* combines the two methods and uses a small batcg $(x_{i:i+n}, y_{i:i+n})$ from the dataset in each iteration to update the weights. With this method, the variance of the weights is reduced which can cause a more stable convergence. [20]

4.2 Overfitting

Some problems may arise if the model is not trained properly, such as *overfitting*. In other words, the model learns too much from the training data. When training the model, we want to minimise the overall training error and create a separation line for the data in order to be classified. In addition, we also want to achieve minimal **generalisation error** or **test error**, or in other words to minimise the difference between the two types of errors. We can achieve that by using a lot of iterations to ensure the training error is minimised. This means that the classification line will pass through all points including the *noise* (irrelevant information and outliners) which is not the optimal case. **Generalisation** is when the model can give accurate predictions on unseen data. By using a lot of iterations the network will just memorise the training dataset without being able to make correct predictions on new data. A well-trained model will separate the noise from the useful data draw conclusions using only the useful information. Taking into consideration the noise will create a very big accuracy on the training set and a low accuracy on the testing data (the gap between the training error and the test error will be too big). This shows that the model is overfitted [6].

4.3 Underfitting

Another problem that can arise is the opposite of overfitting, *underfitting*. Underfitting can be a result of a high training error value. As a result the weights will be poorly calculated and the estimations will not be accurate. Few iterations can also be a factor causing this problem. If we stop the learning process early and the model does not learn the correct patterns that arise through the dataset.

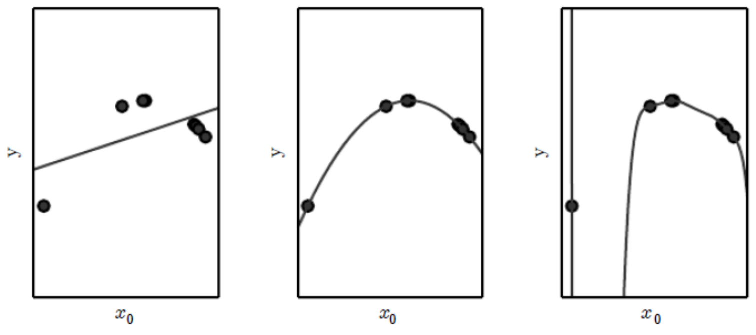


Figure 12: Underfitting, Good fit, Overfitting [6]

5 Dimensionality Reduction

According to *Fast Dimesionality Reduction and Simple PCA* by *Mathew Partridge* and *Rafael Calvo* high dimensional data are hard to process and understand its structure. Here comes *Dimensionality Reduction* which is a technique of reducing the dimensionality of large datasets without changing or loosing important features so we can plot them and visualize them more easily [9]. One technique achieving that is **Principal Component Analysis** (PCA) which we will explain in the following section.

5.1 PCA

PCA analyses the data, finds hidden patterns and reconstracts them based on their similarities.For the sake of the example we are going to assume that we have a dataset with only 3 dimensions (x ,y, z) of n samples, defining a matrix **A** holding our data.

$$A = \begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_n & y_n & z_n \end{pmatrix} \tag{5.1}$$

The first step is *standarization*, where we calculate the mean of each dimension and substruch it from the corresponding data dimension in order to create a dataset with mean value zero. This is a crusial part, preventing biased conclusions from variables with very large values that will dominate over smaller ones. So for a three dimensional datasen of n samples we get thre mean vector and the corresponding standarized matrix:

$$mean = \begin{bmatrix} \overline{x} \\ \overline{y} \\ \overline{z} \end{bmatrix} \tag{5.2}$$

$$\widetilde{A} = \begin{pmatrix} x_1 - \overline{x} & x_2 - \overline{x} & \dots & x_n - \overline{x} \\ y_1 - \overline{y} & y_2 - \overline{y} & \dots & y_n - \overline{y} \\ z_1 - \overline{z} & z_2 - \overline{z} & \dots & z_n - \overline{z} \end{pmatrix} = \begin{pmatrix} \widetilde{x_1} & \widetilde{x_2} & \dots & \widetilde{x_n} \\ \widetilde{y_1} & \widetilde{y_2} & \dots & \widetilde{y_n} \\ \widetilde{z_1} & \widetilde{z_2} & \dots & \widetilde{z_n} \end{pmatrix} \tag{5.3}$$

We know that variance calculates the spread of a one dimensional dataset.

$$Var(x) = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(x_i - \overline{x})}{n-1} \tag{5.4}$$

For our case, we are going to use the covariance, which can measue the variance between the dimensions and their means.

$$Cov(x,y) = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{n-1} \tag{5.5}$$

It is clear that the covariance between one dimension and itself will be equal to variance since $Cov(x,x) = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(x_i - \overline{x})}{n-1}$. If the sign of the covariance between two dimensions is positive that means that they both increase together. If the sign is negative then one decreases as the other increases. For our convenience we store all the covarianves in a matrix called *Coveriance Matrix* with dimensions 3 x 3.

$$C = \begin{pmatrix} Cov(x,x) & Cov(x,y) & Cov(x,z) \\ Cov(y,x) & Cov(y,y) & Cov(z,z) \\ Cov(z,x) & Cov(z,y) & Cov(z,z) \end{pmatrix} \tag{5.6}$$

The objective of PCA is to find the eigenvectors of the covarience matrix which corresponds to the *principal components* of the original data and their significance is shown by the corresponding eigenvalues. The biggest the eigenvalue the biggest the significance of the eigenvectors and the first principal component searches for the greatest variance in the dataset. the second principal component does the same thing but it has to be perpendicular (uncorrelated) with the first one. Eigenvectors are important because they are all *orthogonal* (perpenticular) between them. This means that you can express data with respect to those eigenvectors as your axis instead of in the *x*-*y* plane. To find the eigenvalues $\lambda_\mathbf{k}$ and eigenvectors $\mathbf{v_k}$ for $k = 1, 2, 3$ of the matrix we have to solve the *characteristic equation* $\mid \mathbf{C} - \lambda \mathbf{I} \mid = 0$.

$$\left| \begin{pmatrix} Cov(x,x) & Cov(x,y) & Cov(x,z) \\ Cov(y,x) & Cov(y,y) & Cov(z,z) \\ Cov(z,x) & Cov(z,y) & Cov(z,z) \end{pmatrix} - \lambda \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \right| = 0 \tag{5.7}$$

$$\implies \begin{vmatrix} Cov(x,x) - \lambda & Cov(x,y) & Cov(x,z) \\ Cov(y,x) & Cov(y,y) - \lambda & Cov(z,z) \\ Cov(z,x) & Cov(z,y) & Cov(z,z) - \lambda \end{vmatrix} = 0 \tag{5.8}$$

If we plot our data on the x-y plane together with the eigevectors we can see that the first eigenvector passes through the middle of the data, like an approximation line.The next step is to sort the eigenvalues in a descending order. We can neglect the eigenvalues with the lowest values in order to reduce the dimension of the data. In our case we want to keep the highest two eigenvalues and ingore the lowest one which expresses the lowest significance. We form a matrix called *feature vector* $FV = \begin{pmatrix} v_1 & v_2 \end{pmatrix}$ which consists of the eigenvectors that we decided to keep.

The final step is to calculate the *transpose matrix of the feature vector* and the *transpose matrix of the standarized data* and multiply them.

$$FV = \begin{pmatrix} v_{11} & v_{21} \\ v_{12} & v_{22} \\ v_{13} & v_{23} \end{pmatrix} \implies FV^T = \begin{pmatrix} v_{11} & v_{12} & v_{13} \\ v_{21} & v_{22} & v_{23} \end{pmatrix} \tag{5.9}$$

$$A^T = \begin{pmatrix} x_1 & x_2 & ... & x_n \\ y_1 & y_2 & ... & y_n \\ z_1 & z_2 & ... & z_n \end{pmatrix} \tag{5.10}$$

$$FinalData = FV^T \times A^T = \begin{pmatrix} v_{11} & v_{12} & v_{13} \\ v_{21} & v_{22} & v_{23} \end{pmatrix} \times \begin{pmatrix} x_1 & x_2 & ... & x_n \\ y_1 & y_2 & ... & y_n \\ z_1 & z_2 & ... & z_n \end{pmatrix} \tag{5.11}$$

This transformation will give us our data in terms of the principal compnents that we chose. Our data are in terms of the eigenvectors that we kept. In summary, PCA transformed the data in terms of the two eigenvectors that describe the patterns that it have found as a result they are now more easy to be classified. [18]

This matrix method cannot be used for large n since finding the eigenvalues and eigenvectors can be very time-consuming. For this reason there have been developed alternative faster methods. Another method for finding all the eigenvalues and eigenvectors is **Singular Value Decomposition** (SVD) which is a more general method since it can find eigenvalues for non-square matrices but the faster technique so far for a square matrix is using the **Householder Transformation**. [9, pp.1-3]

5.1.1 PCA in Python

We now try to implement PCA to visualise how PCA works on our dataset *Chest X-Rays for Pneumonia* in Python. We import our data and split the dataset into two arrays, features and arrays.

5.1.2 Importing the Data

```
data = os.path.join("c:", os.sep, "Users", "Pavlos",
    "Desktop","Leeds","University of Leeds","Year 3","Deep Learning for
    Medical Image Classification","train")
cat = ["Normal","Pneumonia"]
cattest = []
img_size = 50
train_data = []
for category in cat:
   path = os.path.join(data,category) #path to chest or abd
   class_num = cat.index(category)
   for img in os.listdir(path):
      img_array = cv2.imread(os.path.join(path,img))
      new_array = cv2.resize(img_array,(img_size,img_size))
      train_data.append([new_array,class_num])

x = [] #train data
y = [] #train labels
for features, label in train_data:
   x.append(features)
   y.append(label)
x = np.array(x)
y = np.array(y)
x = x / 255

print(x.shape)
```

Our array has (2430,50,50,3) which means that the array stored 2430 images of size 50 x 50 with depth 3 since they are RGB. We have to make this array a 2-dimensional so that it can be imported in PCA command.

```
x = x[:,:,:,-3]
print(x.shape)
```

```
(2430,50,50)
```

We create a new array *x_reshape* to store the data with the new dimensions.

```
x_reshape = np.reshape(x,(2430,2500))
```

```
print(x_reshape.shape)
```

Here by reshaping each image instead of being stored in a 50 x 50 matrix is now stored in a vector with size $50x50 = 2500$.

```
(2430,2500)
```

5.1.3 PCA Process

We can now start the PCA process. To track the time it takes to complete the PCA we use *time_start = time.time()*. We want to track 4 principal components so we use *Singular Value Decomposition* (SVD) through $pca = PCA(n\_components = 4)$ to find the eigenvalues and the corresponding eigenvectors. We fit the model with our reshaped data and we can see the time taken for this operation.

```
time_start = time.time()
pca = PCA(n_components=4)
pca_result = pca.fit_transform(x_reshape)
print ('Time elapsed: {} seconds'.format(time.time()-time_start))
```

```
Time elapsed: 0.26021671295166016 seconds
```

```
pca_df = pd.DataFrame(columns = ['pca1','pca2','pca3','pca4'])

pca_df['pca1'] = pca_result[:,0]
pca_df['pca2'] = pca_result[:,1]
pca_df['pca3'] = pca_result[:,2]
pca_df['pca4'] = pca_result[:,3]

print ('Variance explained per principal component:
    {}'.format(pca.explained_variance_ratio_))
top_two_comp = pca_df[['pca1','pca2']] # taking first and second
    principal component
```

We store the principal components and we get the percentage of variance explained by the for principal components.

```
Variance explained per principal component: [0.22128215 0.10041321
    0.07816909 0.04648308]
```

5.1.4 Visualisation of PCA

We can now create a function that will visualise the data in two dimension.

```
def fashion_scatter(x, colors):
   # choose a color palette with seaborn.
   num_classes = len(np.unique(colors))
   palette = np.array(sns.color_palette("hls", num_classes))

   # create a scatter plot.
   f = plt.figure(figsize=(8, 8))
   ax = plt.subplot(aspect='equal')
```

```
    sc = ax.scatter(x[:,0], x[:,1], lw=0, s=40,
        c=palette[colors.astype(np.int)])
    plt.xlim(-25, 25)
    plt.ylim(-25, 25)
    ax.axis('off')
    ax.axis('tight')

    # add the labels for each digit corresponding to the label
    txts = []

    for i in range(num_classes):

       # Position of each label at median of data points.

       xtext, ytext = np.median(x[colors == i, :], axis=0)
       txt = ax.text(xtext, ytext, str(i), fontsize=24)
       txt.set_path_effects([
          PathEffects.Stroke(linewidth=5, foreground="w"),
          PathEffects.Normal()])
       txts.append(txt)

    return f, ax, sc, txts
```

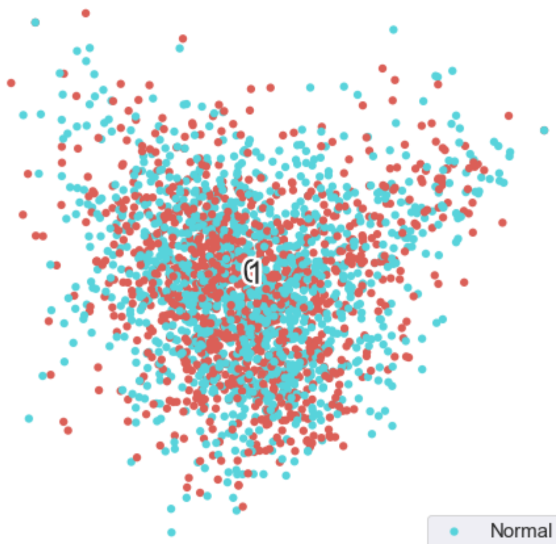


Figure 13: PCA on *Chest X-Rays* dataset

5.2 t-SNE

Another technique for visualizing high-dimensional data into a lower dimension is *t-SNE* which uses *Stochastic Neighbor Embedding*. Stochastic Neughbor Embedding is responsible to convert the high-dimensional Eucilidean distances between the points in the dataset into conditional probabilitites $p_{j|i}$. It is a non-linear dimensionality reduction algorithm that models the probability distribution of neighbors around each point under Gaussian distribution. These probabilities represent similarities, which translates to the probability that a point x_i would pick x_j as its neighbour. The probability $p_{j|i}$ of points x_i and $x_j$with variance σ_i^2 centered at x_i is given by

$$p_{j|i} = \frac{exp(-\|x_i - x_j\|^2/2\sigma_i^2)}{\sum_{i \neq k} exp(-\|x_i - x_k\|^2/2\sigma_i^2)} \tag{5.12}$$

[22]

Variance σ_i^2 is not constant for all datapoints because it has to be small for dense areas and large for areas that the datapoints are more spread. We introduce a new hyperparameter called *perplexity* which denotes the number of close neighbors for each datapoint.
It symmetrizes $(p_{ij} = p_{ji})$ these conditional probabilities in the high dimensional space into the joint probabilities $p_{ij} = \frac{p_{j|i}+p_{i|j}}{2n}$ (where n is the number of datapoints). Subsequently, a low-dimensional map is created with the joint probabilities q_{ij} under *Student t-distributor* (Cauchy distributor) using

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l}(1 + \|y_k - y_l\|^2)^{-1}} \tag{5.13}$$

[22]

The aim of this is to reflect the relation that probabilities p_{ij} have in the high dimensional space. The reason we do not express the low dimensional map under Gaussian distribution is because of the *Crowding Problem.* In Gaussian distribution closed by points will likely squeeze and to solve that Student t-distribution is used. It defines a *cost function* C given as

$$C = KL(P\|Q) = \sum_i \sum_j p_{ij} log \frac{p_{ij}}{q_{ij}} \tag{5.14}$$

[22] which is the *Kullback-Leibler divergence*, between p_{ij} and q_{ij} and P, Q are the conditional probability distributors. This function shows the difference between the high dimensional and low dimensional data. Now using *gradient descent* on the cost function it optimizes the distributor and tries to minimise the difference between these conditional probabilities such that the p_{ij} are similar to the q_{ij}. The value of the gradient shows the amount of attraction or repulsion between two points. In order to calculate the changes in the position of the points we use the following algorithm

$$Y^t = Y^{(t-1)} + \eta \frac{\delta C}{\delta Y} + \alpha(t)(Y^{(t-1)} - Y^{(t-2)}) \tag{5.15}$$

with $Y^t = \{y_1, y_2, ..., y_n\}$ to be the data representation, $\frac{\delta C}{\delta Y}$ the gradient of the cost function, η the learning rate and α the momentum at each iteration that reduces the number of iterations required.
This dimensionality reduction method is more accurate and it visualises the data better from PCA although it is more time consuming [11].

5.2.1 t-SNE in Python

We can now visualise t-SNE in Python using our dataset *Chest X-Rays for Pneumonia*. We import our data as before and split them into two arrays of images and labels. We se t the perplexity to 50 and we use *random state* in order to get the same output when we run the program multiple times.

```
from sklearn.manifold import TSNE
import time
time_start = time.time()
fashion_tsne = TSNE(n_components=2, perplexity = 50,
    random_state=RS).fit_transform(x_reshape)
print ('Time elapsed: {} seconds'.format(time.time()-time_start))
fashion_scatter(fashion_tsne, y)
```

```
Time elapsed: 55.711949825286865 seconds
```

We can immediately see that t-SNE takes more time to visualise our data but the classification is more accurate than PCA.

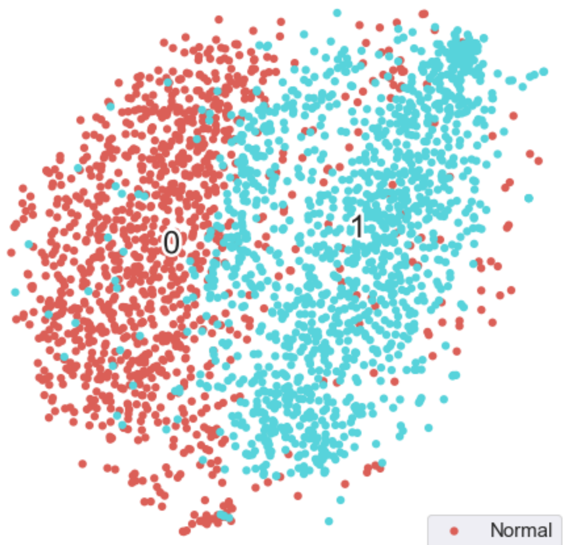


Figure 14: t-SNE on *Chest X-Rays* dataset

6 Clustering

7 Classification

8 Applications in Medical Science

We are going to focus on image classification, by creating a model that will classify *healthy lungs* and lungs infected with *pneumonia* . For this model we are going to use Python 3.6.

8.1 Importing the Data

First we have to import the packages.

```
import numpy as np
import matplotlib.pyplot as plt
import os
import cv2
import random
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten
from keras.layers import Dense
from keras.layers import Activation, Dropout
from keras.models import Sequential
```

We import our dataset using os.path.join(), and then map each image wirhr its correct label (Normal or Pneumonia) using os.path.join(). We resize the image to (50,50) using cv2 to make it more easy for the model to analysie the image and then we store the image and the label to a list which we call *train_data*.

```
data = os.path.join("c:", os.sep, "Users", "Pavlos",
    "Desktop","Leeds","University of Leeds","Year 3","Deep Learning for
    Medical Image Classification","train")
cat = ["Normal","Pneumonia"]
cattest = []
img_size = 50
train_data = []
for category in cat:
   path = os.path.join(data,category) #path to chest or abd
   class_num = cat.index(category)
   for img in os.listdir(path):
      img_array = cv2.imread(os.path.join(path,img))
      new_array = cv2.resize(img_array,(img_size,img_size))
      train_data.append([new_array,class_num])
```

A very important step in machine learning is to shuffle the training data before the training stage. This will prevent the model to detect any unwanted patterns in the dataset that might arise. To avoid this we use *random.shuffle(train_data)*.

```
random.shuffle(train_data)
```

We create two empty lists, and we append the images and the labels respectively. It is essential to convert the lists in arrays, since we have to deal with many variablels of the same type and it requires less memory. At last we want to normalise our data so they can be fed inside the activation function that we are going to use later.

```
x = [] #train data
y = [] #train labels
for features, label in train_data:
   x.append(features)
   y.append(label)
x = np.array(x)
y = np.array(y)
x = x / 255
```

We can see some examples from our training data by creating a figure.

```
plt.figure(figsize=(15,20))
for i in range(28):
   plt.subplot(10,7,i+1)
   plt.xticks([])
   plt.yticks([])
   plt.grid(False )
   plt.imshow(x[i], cmap=plt.cm.binary)
   plt.xlabel(cat[y[i]])
plt.show()
```

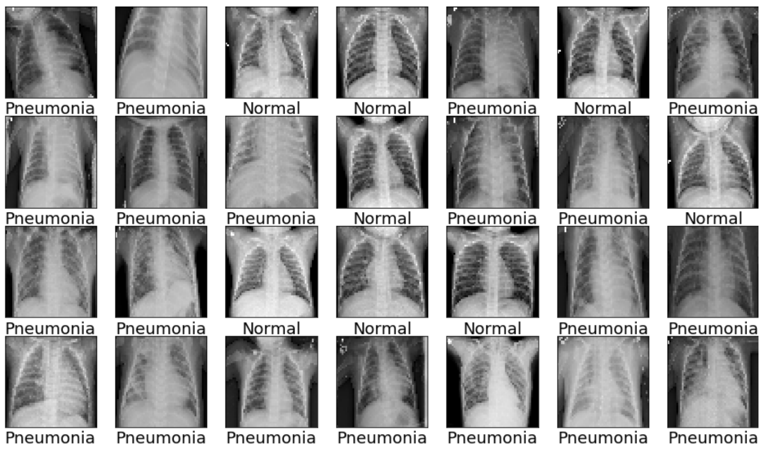


Figure 15: Training Sample

We now do the same to import the validation dataset and split the map the images with their labels.

```
for category in cat_v:
   path = os.path.join(v_data,category) #path to chest or abd
   class_num = cat_v.index(category)
   for img in os.listdir(path):
      img_array = cv2.imread(os.path.join(path,img))
      new_array = cv2.resize(img_array,(img_size,img_size))
      val_data.append([new_array,class_num])
random.shuffle(val_data)
x_val = []
```

```
y_val = []
for features, label in val_data:
   x_val.append(features)
   y_val.append(label)
x_val = np.array(x_val)
y_val = np.array(y_val)
x_val = x_val/255
```

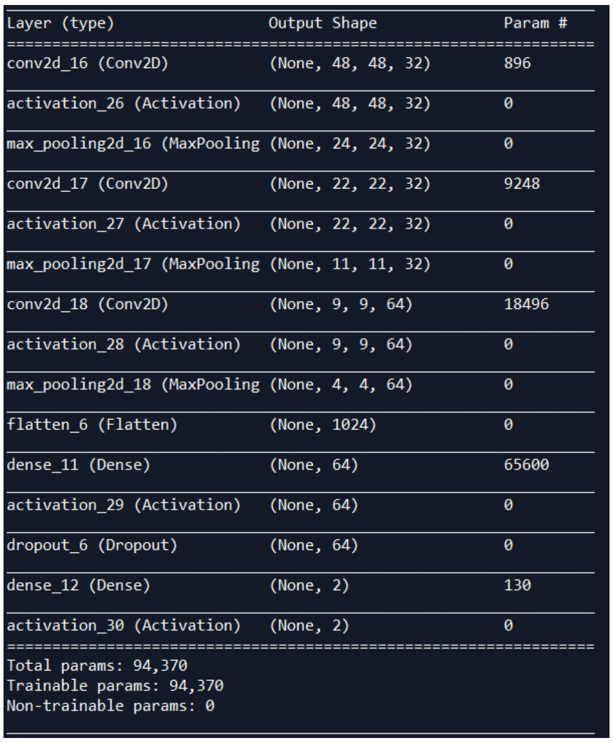
8.2 Creating the Convolutional Neural Network

Once we imported our training data and our validation data, we can create our convolutional neural netwok. For that we are going to use *Keras* library from *tensorflow*. We use the *Sequential* API (Application Programming Interface), to create our deep learning model. We create a convolution layer which consists of 32 filters of size 3 x 3 and we set the input size as the shape of the first train image. It follows an activation fucntion *ReLu* which introduces the non-linearity in the model and transforms all the negative values to zero. A final layer called *MaxPooling* is used to extract maxima from the image as discussed before reducing the dimensions of the images. this step will make the computations that follow easier. We repeat this proccess and add another stack but this time with 64 filters of size 3 x 3. The last layer is the model is flatten, where the last outputs of the pooling process are flattened into a 1D vector. From this point follows a multilayer feed forward neural network with a hidden layer which consists of 64 elements using *ReLu* as an activation function. *Dropout* technique is used at the end of the neural network whcih is a regularization technique which randomly ignores a ratio of the neurons by setting them to zero in each iteration during the training process as this prevents the model to observe the a pattern twice. This will prevent the model from overfitting. The last layer of the network consists of 2 elements, the two classes that we want to classify our data.

```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=x[0].shape))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten()) # this converts our 3D feature maps to 1D feature
    vectors
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(2))
model.add(Activation('sigmoid'))
```

An overview of the model's architecture can be seen using the following command which shows the dimensions of the data as they flow in the model.

```
print(model.summary())
```



```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_16 (Conv2D)           (None, 48, 48, 32)        896
_________________________________________________________________
activation_26 (Activation)   (None, 48, 48, 32)        0
_________________________________________________________________
max_pooling2d_16 (MaxPooling (None, 24, 24, 32)        0
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 22, 22, 32)        9248
_________________________________________________________________
activation_27 (Activation)   (None, 22, 22, 32)        0
_________________________________________________________________
max_pooling2d_17 (MaxPooling (None, 11, 11, 32)        0
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 9, 9, 64)          18496
_________________________________________________________________
activation_28 (Activation)   (None, 9, 9, 64)          0
_________________________________________________________________
max_pooling2d_18 (MaxPooling (None, 4, 4, 64)          0
_________________________________________________________________
flatten_6 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_11 (Dense)             (None, 64)                65600
_________________________________________________________________
activation_29 (Activation)   (None, 64)                0
_________________________________________________________________
dropout_6 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_12 (Dense)             (None, 2)                 130
_________________________________________________________________
activation_30 (Activation)   (None, 2)                 0
=================================================================
Total params: 94,370
Trainable params: 94,370
Non-trainable params: 0
_________________________________________________________________
```

Figure 16: Model Summary

We define the loss function to be *Sparce Categorical Crossentropy* to calculate the similarity between the prpedicted class and the true class of the sample..............................We set the *SGD* (Stochastic Gradient Descent) as the optimizer algorithm, and we define the metrics that we want to record with each epoch.

```
model.compile(loss='sparse_categorical_crossentropy', optimizer='SGD',
    metrics=['accuracy'])
```

We feed the training sets *x* , *y* into the model using a *batch size* of 32.Batch size is...........We set the number of times that the model will cycle through the whole training data set and we define the validation sets *x_val*, *y_val* . Note that the validation sets will not be used to train the model.

```
history=model.fit(x, y,batch_size=32,epochs=50,validation_data=(x_val,
    y_val))
```

We keep track of the loss and the accuracy of the model, both on training and on validation data.

```
train_loss, train_acc = model.evaluate(x, y, verbose=2)
print('\nTrain accuracy:', train_acc, '\nTrain loss:', train_loss)
val_loss, val_acc = model.evaluate(x_val, y_val, verbose=2)
print('\nValidation accuracy:', val_acc, '\nValidation loss:',val_loss)
```

```
Train accuracy: 0.9670782089233398
Train loss: 0.07896106427098498
```

```
Validation accuracy: 0.9599359035491943
Validation loss: 0.09584566874381824
```

8.3 Accuracy and Loss on Training and Validation Sets

We can plot a graph to visualise the accuracy and the loss on both training and vaidation datasets. We can see that the accuracy slowly converges to 1 and the loss is decreasing converging approximately on 0.1.

```
fig, (ax1, ax2) = plt.subplots(1, 2,figsize=(15,7))
ax1.set_xlabel('Epoch',size=15)
ax1.set_ylabel('Loss',size=15)
ax1.set_title('Model Loss',fontweight="bold", size=15)
ax1.set_ylim([0, 1])
ax1.set_xlim([0, epochs])
ax1.plot(history.history['loss'],color='red')
ax1.plot(history.history['val_loss'],color='green')
ax1.legend(['training loss', 'validation loss'], loc='upper left')
ax2.set_xlabel('Epoch',size=15)
ax2.set_ylabel('Accuracy',size=15)
ax2.set_title('Model Accuracy',fontweight="bold", size=15)
ax2.set_ylim([0, 1])
ax2.set_xlim([0, epochs])
ax2.plot(history.history['accuracy'],color='red')
ax2.plot(history.history['val_accuracy'],color='green')
ax2.legend(['training accuracy', 'validation accuracy'], loc='upper
    left')
fig.savefig('lossacc.png')
plt.show()
```

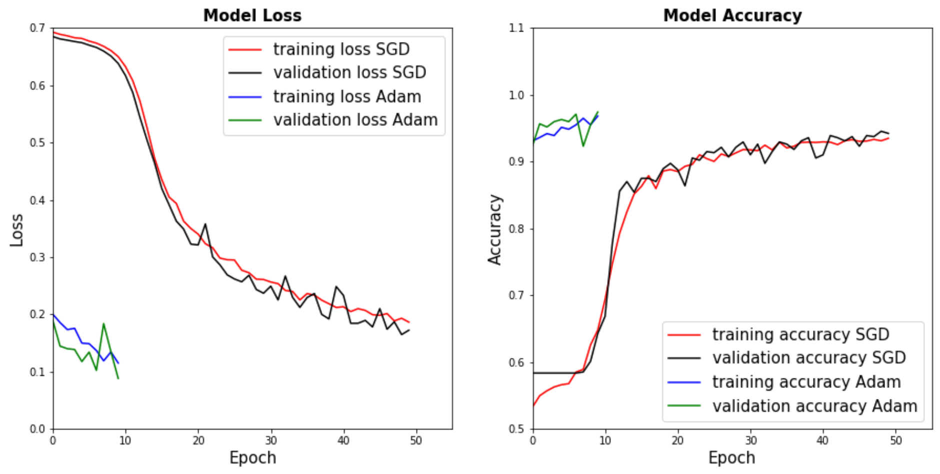


Figure 17: Accuracy and Loss on Training and Validation Datasets

We can plot a figure to see the accuracy of the predictions on the vaidation dataset using the following code.

```
predictions = model.predict(x_val)

def plot_image(i, predictions_array, true_label, img):
  predictions_array, true_label, img = predictions_array, true_label[i],
      img[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  plt.imshow(img)

  predicted_label = np.argmax(predictions_array)
  if predicted_label == true_label:
    color = 'blue'
  else:
    color = 'red'

  plt.xlabel("{} {:2.0f}% ({})".format(cat_v[predicted_label],
                                100*np.max(predictions_array),
                                cat_v[true_label]),
                                color=color)

def plot_value_array(i, predictions_array, true_label):
  predictions_array, true_label = predictions_array, true_label[i]
  plt.grid(False)
  plt.xticks(range(2))
  plt.yticks([])
  thisplot = plt.bar(range(2), predictions_array, color="#777777")
  plt.ylim([0, 1])
  predicted_label = np.argmax(predictions_array)

  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('blue')

num_rows = 3
num_cols = 3
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_image(i, predictions[i], y_val, x_val)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, predictions[i], y_val)
plt.tight_layout()
plt.savefig('valpred.png')
plt.show()
```

We are now ready to make predictions using the testing data.

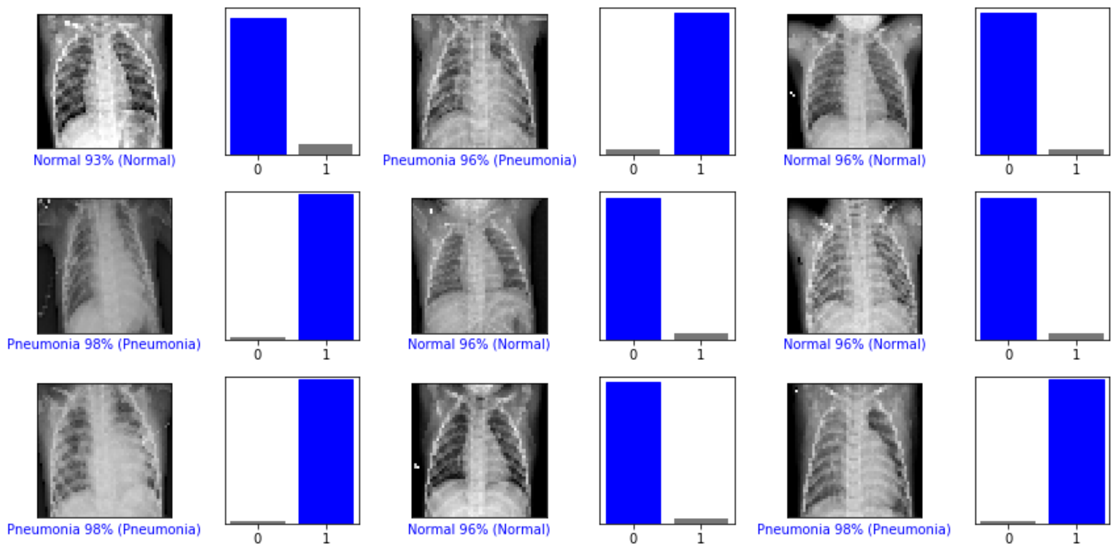


Figure 18: Accuracy of predictions on Validation Dataset

9 Discussion

References

[1] Kermany, Daniel; Zhang, Kang; Goldbaum, Michael (2018). *Dataset: Labeled Optical Coherence Tomography (OCT) and Chest X-Ray Images for Classification DOI: http://dx.doi.org/10.17632/rscbjbr9sj.2#file-41d542e7-7f91-47f6-9ff2-dd8e5a5a7861*

[2] Patterson, Dan W. (1990). *Introduction to artificial intelligence and expert systems*

[3] Krzyk, Kamil (2018). *Coding Deep Learning For Beginners*

[4] Zhou S. Kevin, Greenspan Hayit, Shen Dinggang eds. (2017). *Deep Learning for Medical Image Analysis*

[5] Maier, Andreas, Syben Christopher, Lasser Tobias, Riess Christian (2018). *A gentle introduction to deep learning in medical image processing*

[6] Goodfellow Ian, Bengio Yoshua, Courville Aaron, Bach Francis (2017). *Deep Learning*

[7] Quoc V. Le (2015). *A Tutorial on Deep Learning Part 2: Autoencoders, Convolutional Neural Networks and Recurrent Neural Networks*

[8] Keiron OShea, Ryan Nash (2015). *An Introduction to Convolutional Neural Networks*

[9] Partridge Matthew, Calvo Rafael (1997). *Fast Dimensionality Reduction and Simple PCA*

[10] Lundervold Selvikvag Alexander, Lundervold Arvid (2018). *An overview of deep learning in medical imaging focusing on MRI*

[11] Shamir Ron (2017). *Algorithms for Big Data Analysis in Biology and Medicine, Lecture 5: t-SNE and Consensus Clustering*

[12] Justin Ker, Lipo Wang, Jai Rao and Tchoyoson Lim (2017). *Deep Learning Applications in Medical Image Analysis*

[13] P.Sibi, S.Allwyn and P.Siddarth (2013). *Analysis of different activation functions using back propagation neural networks pp.1-2*

[14] R. Lippmann (1987). *An introduction to computing with neural nets*

[15] Dezs Ribli, Anna Horvth, Zsuzsa Unger, Pter Pollner, Istvn Csabai (2018). *Detecting and classifying lesions in mammograms with Deep Learning*

[16] Ahmed Fawzy Gad (2018). *Practical Computer Vision Applications Using Deep Learning with CNNs*

[17] Arunava (2018). *Derivative of the Sigmoid function*

[18] Lindasy I Smith (2002). *A tutorial on Principal Components Analysis*

[19] J.G. Makin (2006). *Backpropagation*

[20] Sebastian Ruder (2016). *An overview of gradient descent optimization algorithms*

[21] Rahul Yedida and Snehanshu Saha (2019). *A novel adaptive learning rate scheduler for deep neural networks*

[22] Laurens van der Maaten, Geoffrey Hinton (2008). *Visualizing Data using t-SNE*