# CS4287 Neural Computing Assignment

***Convolutional Neural Networks***

***Link to Notebook***
***https://colab.research.google.com/drive/1f_DenNVZYAP33ZcCxh_faxXOSXJoGvdO?usp=sharing***

By:

*Robert Mooney 19278772*
*Oliver Nagy 19269749*

# Table of Contents

## 1. The Dataset

We chose to use MRI scans dataset for Alzheimer detection [1] for our project which consists of 4 classes including

1. Mild Demented

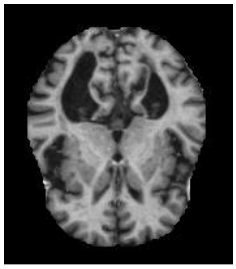

Figure 1.1 Image of mild demented MRI scan

2. Moderate Demented

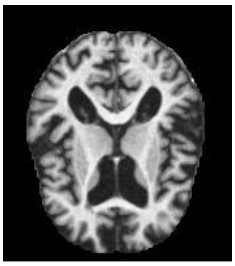

Figure 1.2 Image of moderately demented MRI scan

3. Non-Demented

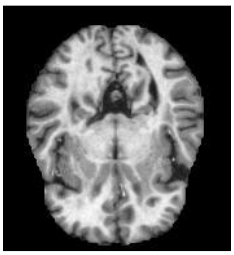

Figure 1.3 Image of Non-Demented MRI scan

4. Very Mild Demented

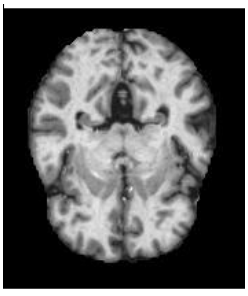

Figure 1.4 Image of Very  mild demented MRI scan

The  training and test sets roughly consist of 34,000 images altogether.  More specifically the training dataset is 27,200 images where the test set is 6800.  For each class there is a corresponding folder where images are labelled from 0 to N . One image from the non-demented folder would look like this *nonDem1002.jpg*. Where 1002 indicates the index of the image within the folder. Each image was 224x224

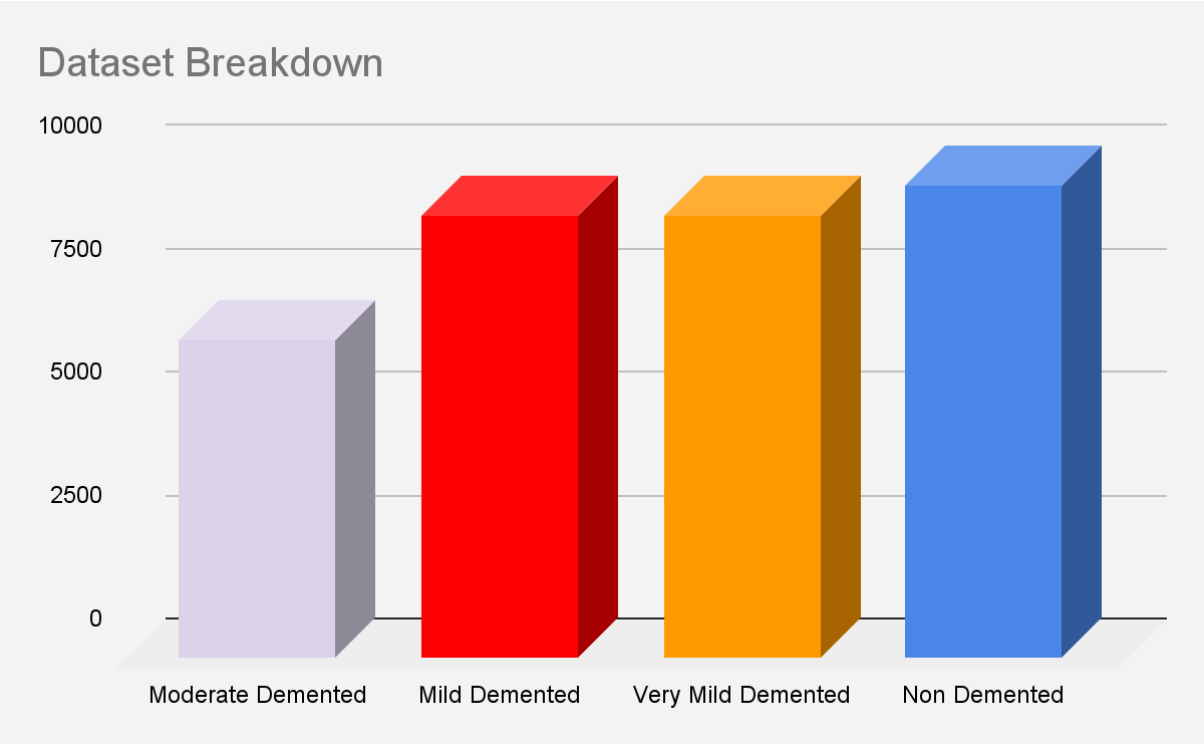


Figure 1.5 Chart representing our Dataset

Once we have obtained the dataset, we uploaded it to our google drive which we linked with jupyter notebook. In our implementation we imported the images by specifying the path such as

```
def get_images(dir_name='/content/drive/MyDrive/CS4287/Alzheimer_s
Dataset', label_to_class=label_to_class):
    """read images / labels from directory"""
```

Figure 1.6 Code snippet  showing importing of data

After this was done we combined the two data sets into one by doing the following:

```
#combine train and test datasets
```

```
    for j in ['/train','/test']:
        for label_name in os.listdir(dir_name+str(j)):
            cls = label_to_class[label_name]

            for img_name in os.listdir('/'.join([dir_name+str(j),
label_name])):
                img = load_img('/'.join([dir_name+str(j), label_name,
img_name]), target_size=(W, H))
                img = img_to_array(img)

                Images.append(img)
                Classes.append(cls)

    Images = np.array(Images, dtype=np.float32)
    Classes = np.array(Classes, dtype=np.float32)
    Images, Classes = shuffle(Images, Classes, random_state=0)

    return Images, Classes
```

Figure 1.7 Code snippet to showing splitting of data

Now all that was left to do was to split our newly combined dataset into two one being test and train. For this we used the *train_test_split* method

```
## split train / test


indices_train, indices_test =
train_test_split(list(range(Images.shape[0])), train_size=0.8,
test_size=0.2, shuffle=False)

x_train = Images[indices_train]
y_train = Classes[indices_train]
x_test = Images[indices_test]
y_test = Classes[indices_test]

x_train.shape, y_train.shape, x_test.shape, y_test.shape
```

Figure 1.8 Code snippet showing train_test_split method

This was done in order to randomly generate a test/train set again.

Furthermore, we performed pre-processing on the test and train images. We achieved this by using `ImageDataGenerator` then we randomly rotated the images

from 0-40 degrees , width and height shifted them by up to 0.15 and randomly flipped them either horizontally or vertically.

```
## to image data generator
from keras.applications.resnet import ResNet50,preprocess_input

datagen_train = ImageDataGenerator(
    preprocessing_function=preprocess_input, # image preprocessing
function
    rotation_range=40,                       # randomly rotate images
in the range
    width_shift_range=0.15,                   # randomly shift images
horizontally
    height_shift_range=0.15,                  # randomly shift images
vertically
    horizontal_flip=True,                    # randomly flip images
horizontally
    vertical_flip=True,                     # randomly flip images
vertically
)
datagen_test = ImageDataGenerator(
    preprocessing_function=preprocess_input, # image preprocessing
function
)
```

Figure 1.9 Code snippet to showing image pre-processing

## 2. Network Structures and Hyperparameters

### Network Structure

Our network structure utilised the ResNet50 Convolutional Neural Network (CNN) architecture. ResNet stands for Residual Network and this type of neural network was first introduced by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun back in 2015 [2]. We discussed within the team that ResNet would be a CNN to base off of, as it is a very popular and effective CNN. ResNet is also compatible with the Keras API.

Since its introduction to the industry, ResNet has many different variations that are similar in concept but have a different number of total layers, two of the most notable ones being ResNet34 and ResNet50, the main difference being the total number of layers in the model. We chose to base our model off of the ResNet50 variation as

this variation makes use of 3-layer bottleneck blocks rather than the 2-layer bottle neck blocks of the 34 variation. This then allows for ResNet50 to have increased accuracy and a shorter amount of training time required, which is helpful for our dataset.

ResNet50 has 48 convolution layers with another 2 layers, one being a max pooling layer and the other is an average pooling layer [3]. Additional layers implemented in the model included flatten, dense, and batch normalisation. While some hyper-parameters included activation function, early stop, epochs, etc.

**Flatten**

A flatten layer consists of flattening the output of the CNN model into a single vector. This is then connected to the final classification model, where it allows us to classify the data as being one of the Alzheimer values.

**Dense**

Flattening the data was important to do as a dense layer requires the input to be in single-dimensional shape, otherwise we could not use a dense layer in the model. A dense layer is a layer of neurons in which each neuron receives input from all the neurons of the previous layer. For Keras implementation we used the operation **output = activation(dot(input, kernel) + bias)** (shown in Figure 2.1).

```
model.add(Dense(512,activation='relu',kernel_initializer='he_uniform'))
model.add(Dense(128,activation='relu',kernel_initializer='he_uniform'))
model.add(Dense(32,activation='relu',kernel_initializer='he_uniform'))
```

Figure 2.1 Code snippet of dense layers

We have 4 dense layers in our model, the first 3 had 512, 128, and 32 neurons respectively. For the final dense layer, we had 4 neurons - one for each class. Each neuron on this final dense layer would hold the probability that the input image belongs to each respective class. This then allows for the neuron with the highest probability to be chosen as the output.

| | | |
|---|---|---|
| dense (Dense) | (None, 512) | 1049088 |
| dense_1 (Dense) | (None, 128) | 65664 |
| dense_2 (Dense) | (None, 32) | 4128 |
| dense_3 (Dense) | (None, 4) | 132 |

Figure 2.2 Image showing dense layer summary

The final dense layer uses ‘Softmax’ as the activation function rather than ‘relu’. Softmax is used when 2 or more than 2 classes are present [4]. Softmax is the mathematical function in python that converts the vector of numbers into the probability (value between 0 and 1), mentioned in the previous paragraph.

**Batch Normalisation**

Batch normalisation is one of the most common normalisation techniques. It allows for all of the layers to learn more independently by normalising the output data of the previous layer. It does this through mini-batches instead of the full data set.

**Weight Initialisation**

The aim of weight initialisation is to prevent layer activation outputs from exploding or vanishing during the course of a forward pass through a deep neural network. If either occurs, loss gradients will either be too large or too small to flow backwards beneficially [5]. There are two types of weight distribution methods: Normal distribution and Uniform distribution.

$$\text{Normal distribution with mean 0 and variance } \sigma^2 = \frac{2}{fan_{avg}}$$

$$\text{Uniform distribution between } -r \text{ and } +r \text{ with } r = \sqrt{\frac{3}{fan_{avg}}}$$

Figure 2.3 Image showing weight distribution methods

Through some research as a team, we found that normal distribution is normally used to model phenomenon that tend to follow a “bell-curve” shape, while uniform distribution is used when the potential outcome is equally likely [6]. With our dataset

breakdown, shown in Chapter 1, we found that the total amount of images for each value of Alzheimer's were similar, so we thought that a Uniform distribution would be more optimal.

Keras, by default, uses Glorot initialisation with a uniform distribution. However, for this project we changed the initialisation to He initialisation. This can be done through the code shown below, with the option of having either uniform distribution ('he_uniform') or normal distribution ('he_normal').

```
model.add(Dense(512,activation='relu',kernel_initializer='he_uniform'))
```

Figure 2.4 Code snippet of implementation of uniform distribution in Keras

## Residual Learning

As mentioned before, the creators of ResNet (Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun) introduced their findings in their paper "Deep Residual Learning for Image Recognition" in 2015, with residual learning being one of the more distinguishing features of ResNet that they talked about. Their main motivation for introducing residual learning was to further improve the efficiency of their CNNs. This new discovery awarded them with awards such as first place in the ILSVRC classification competition, with 194 object categories won over second place with 4 object categories won and third place with just 2 [7]. This truly showed how revolutionary their paper was to the industry.

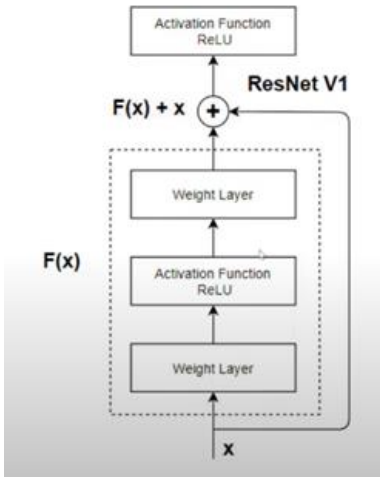


Figure 2.5 Image showing a residual block

Their proposal in the paper was to introduce residual blocks (shown in Figure 2.5), which have skip connections in them. A skip connection is when the model skips one or more layers (F(x)), the output of one-layer (x) acts as the input of the following layer (F(x) + x). Normally, without the skip connection the neural network would have to find the direct mapping of the input to the output, but these residual blocks allow for the difference of the input and output to be found.

The main idea behind skip connections is to skip finding the direct mapping, which can be more difficult and requires more computational performance. So, skip connections allow for a larger number of layers without a major increase in performance issues. This has allowed ResNet variations to become so large in layers, such as ResNet1000.

**Hyper-parameters**

Some of the hyper-parameters of the model included:

1. **Activation function**: Calculates a weighted total and then adds bias to it to decide whether a neuron should be activated or not.

2. **The optimiser and its learning rate**

3. **Number of epochs**: The number of times an entire dataset is passed through the neural network.

4. **Early Stopping**: Stops training once the model's performance has reached its peak and stops improving on the validation dataset.

These hyper-parameters will be investigated further in Chapter 8 and will be tested on their impact when varied.

## 3. Loss Function

Loss function is a method that evaluates how well the algorithm learns the data and produces correct outputs. It computes the distance between our predicted value and the actual value using a mathematical formula[11].

In this implementation the Loss function used was Categorical Cross Entropy, usually this type of loss function performs well  in multi-task classification tasks. These are tasks where an example can only belong to one out of many possible categories, and the model must decide which one in[12].

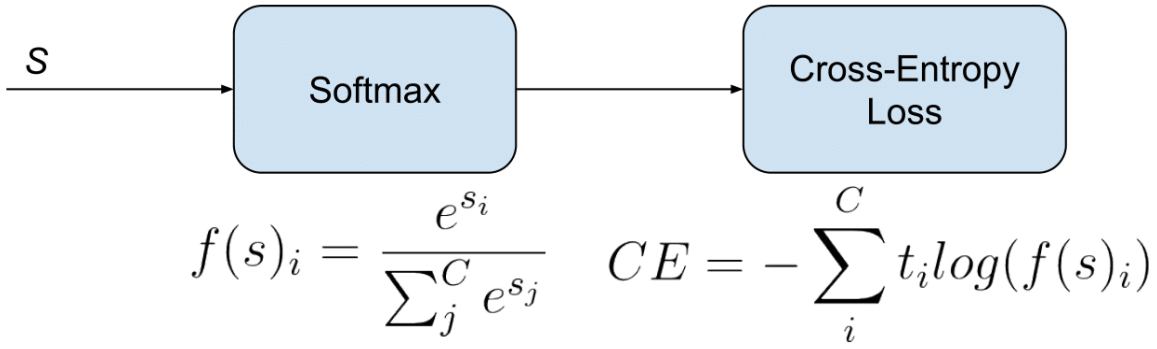


Figure 3.1 Representation of Cross entropy [13]

Within our dataset we have 4 classes therefore `CategoricalCrossentropy` is the best suited Loss Function for this implementation.

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),

loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True), #Loss
Function
              metrics=METRICS)
```

Figure 3.2 Code snippet showing Categorical Cross Entropy Implementation

## 4. The Optimiser

As a team we discussed which optimiser would be the most suitable for our project by being efficient timewise but by also meeting our needs. We looked into many different optimisers but decided on using Adam (Adaptive Moment Estimation). This decision was because of many factors but the main ones were that it is easy to implement using Keras, and we as a team, had previous knowledge of Adam.

Adam is an optimisation algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iteratively based in

training data [8]. In other words, Adam is an updated improvement on other Stochastic Gradient Descent (GSD) variations (e.g. RMSProp) that gives individual adaptive learning rates to different parameters instead of the single learning rate that other GSDs use. Therefore, increasing effectiveness and efficiency due to its low memory requirements.

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
              loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
              metrics=METRICS)
```

Figure 4 Code snippet of implementation of Adam optimiser in Keras

Another major benefit to Adam is that it is very easy to modify its initial learning rate. So, with some tuning, we could further optimise our CNN by increasing its accuracy (see Chapter 8). Any increase in accuracy is important to us, as we thought that when dealing with the topic of building a CNN to identify types of Alzheimer's, then we should try and get the most accurate model that we can achieve.

## 5. Cross Fold Validation

Cross-validation is a resampling procedure used to evaluate machine learning models on a limited data sample [9]. A major disadvantage of splitting our data into one training and testing set is that the test set might not follow the same distribution of classes that the training set did. A way to solve this would be through K-Fold Cross-Validation, where it creates a process where each sample in the data will be included in the test set.

As shown in Figure 5, the data set is equally divided into a number of folds (defined as k). Each unique group is taken as the test set (shown as the green), while the remaining groups act as the training data set. A score is then taken for each grouping, which is then averaged to show the skill of the modelling at estimating how its predictions performed.

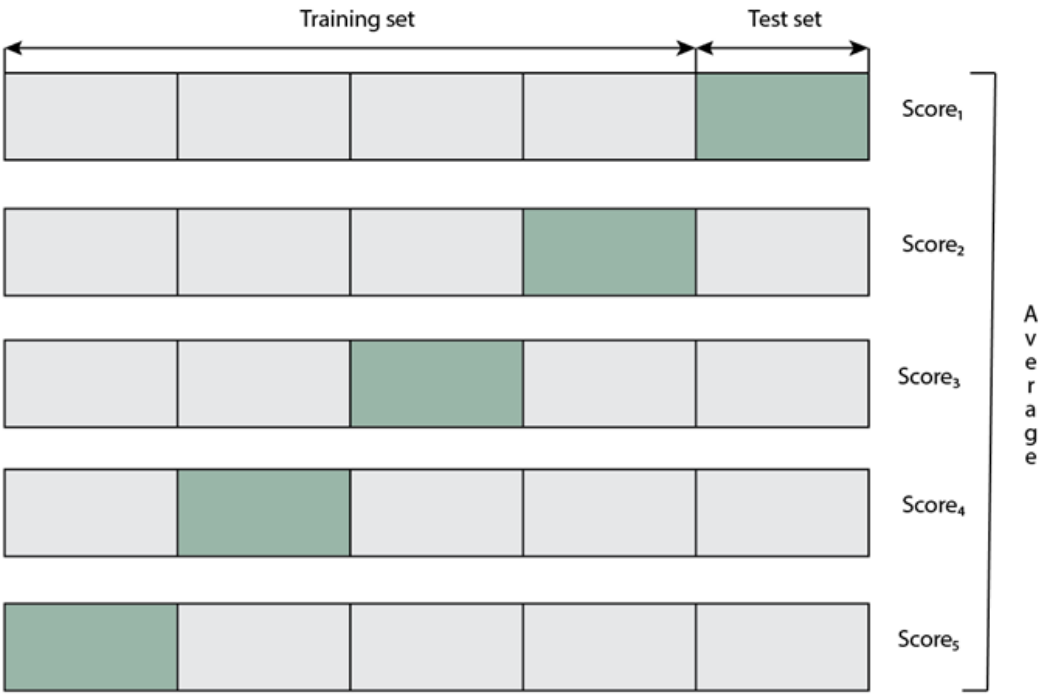


Figure 5 Image of k-fold cross-validation [10]

Furthermore, there are multiple other variations of Cross validation that build on this idea, such as Leave-One-Out Cross-Validation, Repeated K-Fold Cross-Validation, etc.

With all this said, as a team we decided not to include K-Fold Cross-Validation when training the model as it requires a significant amount of computational performance, when our model already had a long running time and thought that this would further increase it.

## 6. Results

The model was ran using Google Collab , we switched the “Runtime time” to GPU as we found it greatly decreases the time taken to train our model. After this the time taken per epoch was anywhere from 59s to 75s. The model ran for 42 epochs to get the results below.

```
Epoch 1/100
40/40 [==============================] - ETA: 0s - loss: 1.0282 - accuracy: 0.7855 - precision: 0.5927 - recall: 0.4547 - auc: 0.8193
Epoch 1: val_accuracy improved from -inf to 0.78066, saving model to 01.h5
40/40 [==============================] - 76s 2s/step - loss: 1.0282 - accuracy: 0.7855 - precision: 0.5927 - recall: 0.4547 - auc: 0.8193 -
```

Figure 6.1 Image showing initial results when training our model

Our model started with an accuracy of 79% , precision of 59% and a recall of 45%. The model finished with an accuracy of 95% , precision of 92% and recall of 91%. After this the validation accuracy, training accuracy , loss, and AUC were plotted. Which demonstrated how well the model worked on both training and test data sets.

```
Epoch 42/100
40/40 [==============================] - ETA: 0s - loss: 0.2187 - accuracy: 0.9578 - precision: 0.9234 - recall: 0.9062 - auc: 0.9916
Epoch 42: val_accuracy did not improve from 0.89668
40/40 [==============================] - 59s 1s/step - loss: 0.2187 - accuracy: 0.9578 - precision: 0.9234 - recall: 0.9062 - auc: 0.9916 -
Epoch 42: early stopping
```

Figure 6.2 Image showing end results when training our model

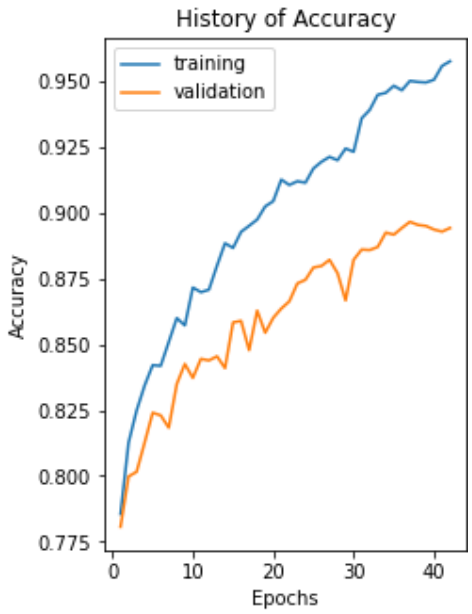


Figure 6.3 Graph showing History of Accuracy
for training and validation data

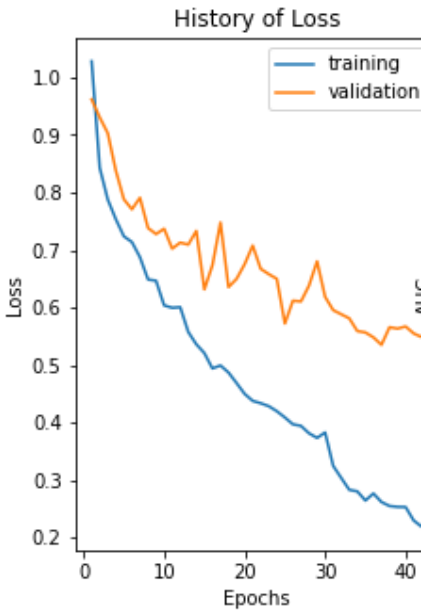


Figure 6.4 Graph showing History of Loss
for training and validation data

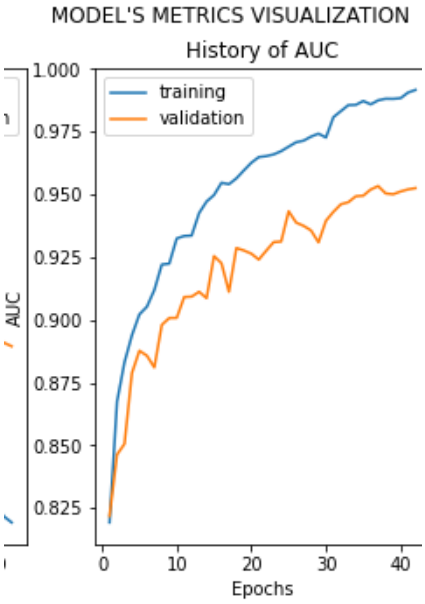


Figure 6.5 Graph showing History of AUC
for training and validation data

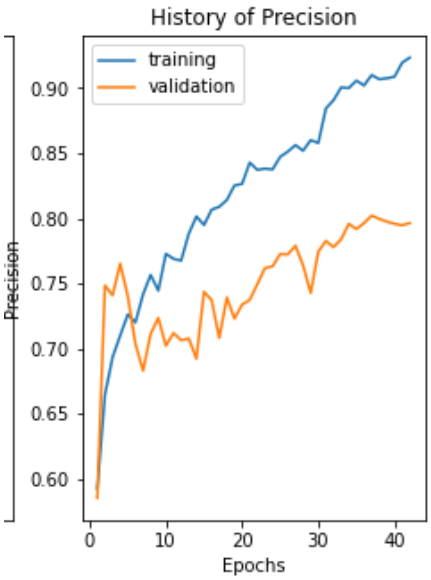


Figure 6.6 Graph showing History of Precision
for training and validation data

The confusion matrix describes how well the performance of a classification model. In our case it is made up of 16 fields. On the Y axis we have the True label and on the X axis we have the Predicted label. This shows false positives and false negatives with ease.

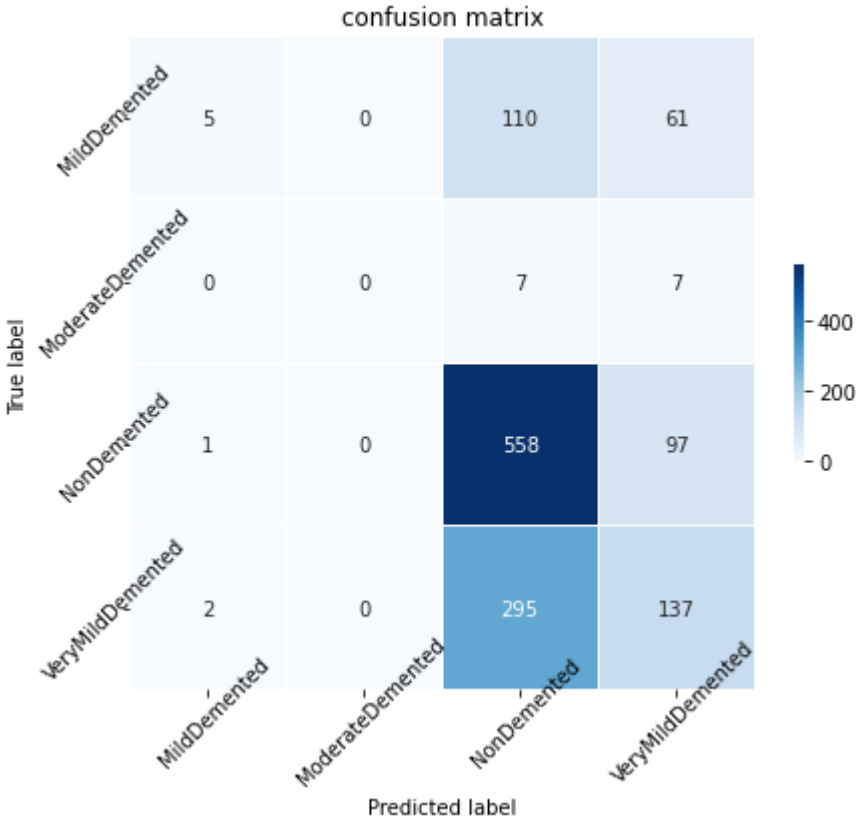


Figure 6.7
Graph showing Confusion Matrix
testing

## 7. Evaluation of Results

After acquiring the results above, we were content with how well our model performed during training although we concluded that our model was performing better on the training data.

Further examination of our results we saw that our model was overfitting. As we established above, models' accuracy during training was 95% whereas during testing it turned out to be 55% . This meant there was a 40% difference in accuracy between training and testing accuracy.

Furthermore, when it came to precision and recall the test data got a weight average of 53% and 55% respectively . Compared to precision of 92% and recall of 91% for the training dataset. This gave us a clear indication that our model is well outperforming on the training set while underperforming on the test set. After establishing that our model was overfitting, we knew we had found a way to treat it, some methods may include Data Augmentation and Adding more data . Most likely our dataset wasn't the root cause of our issue as we had over 34,000 images so we decided to try Data Augmentation  which we will discuss in the next section.

```
accuracy (testing): 54.69%
```

Figure 7.1 Snippet showing accuracy result for testing

```
F1 Score (testing): 48.55%
```

Figure 7.2 Showing F1 score for testing

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.62 | 0.03 | 0.05 | 176 |
| 1 | 0.00 | 0.00 | 0.00 | 14 |
| 2 | 0.58 | 0.85 | 0.69 | 656 |
| 3 | 0.45 | 0.32 | 0.37 | 434 |
| accuracy | | | 0.55 | 1280 |
| macro avg | 0.41 | 0.30 | 0.28 | 1280 |
| weighted avg | 0.53 | 0.55 | 0.49 | 1280 |

Figure 7.3 Table showing classification results

## 8. Impact of Varying Hyperparameters

### Data augmentation

After detecting overfitting, the first thing we tried was Data augmentation. Our initial values for image pre-processing were as follows `rotation_range=40,width_shift_range=0.15` and `height_shift_range=0.15`.These values were changed to 60, 0.25 and 0.25 respectively.

```
 preprocessing_function=preprocess_input, # image preprocessing
function

    rotation_range=60,                          # randomly rotate images
in the range

    width_shift_range=0.25,                      # randomly shift images
horizontally

    height_shift_range=0.25,                     # randomly shift images
vertically

    horizontal_flip=True,                       # randomly flip images
horizontally

    vertical_flip=True,                        # randomly flip images
vertically
```

Figure 8.1 Code snippet showing Data Augmentation parameter/pre-processing variables

After running the model, we got our results back. In this case varying the parameters for data augmentation did not have much of an effect on our test results which came to an accuracy of 52% which meant a decrease of nearly 3% from our initial value in the previous section. It did however lower the accuracy for training from 95% to 89%. We should also mention it only ran for 30 epochs as we have early stopping in our implementation.

```
accuracy (testing): 51.95%
```

Figure 8.2 Results for testing after Data Augmentation

```
Epoch 30: val_accuracy did not improve from 0.85488
40/40 [==============================] - 60s 1s/step - loss: 0.5140 - accuracy: 0.8906 - precision: 0.8116 - recall: 0.7326
Epoch 30: early stopping
```

Figure 8.3 Final values during training after Data Augmentation

The loss was 54% , precision and recall were 81% and 73% respectively. This meant that loss saw a 30% increase , precision a 10% decrease and recall a 16% increase. To conclude this did not solve our overfitting problem but it made our training set perform worse while also making our test worse as well.

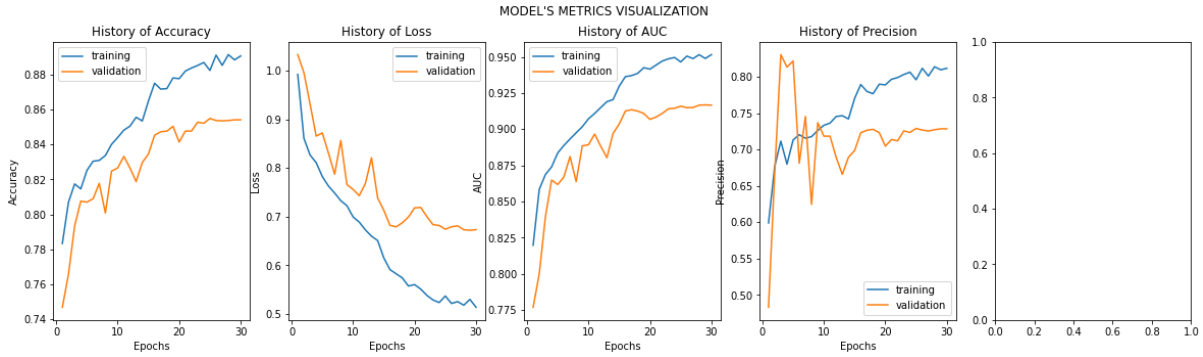


Figure 8.4 Graphs showing History of AUC , Accuracy , Loss and Precision for training and validation data after Data Augmentation

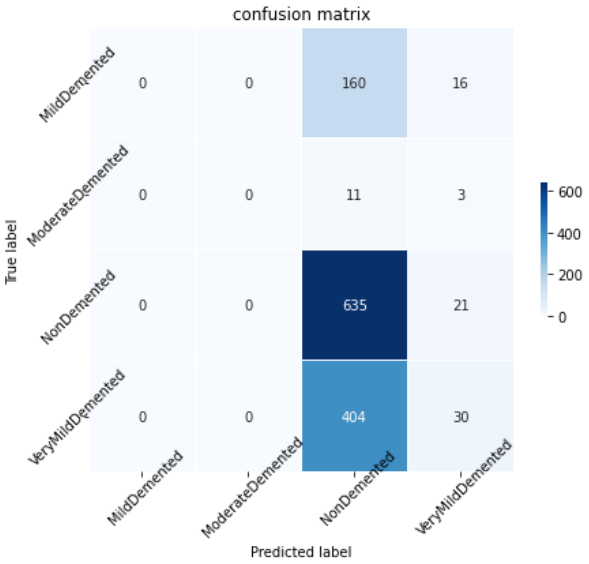


Figure 8.5 Image of Confusion matrix after Data Augmentation

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.00 | 0.00 | 0.00 | 176 |
| 1 | 0.00 | 0.00 | 0.00 | 14 |
| 2 | 0.52 | 0.97 | 0.68 | 656 |
| 3 | 0.43 | 0.07 | 0.12 | 434 |
| accuracy | | | 0.52 | 1280 |
| macro avg | 0.24 | 0.26 | 0.20 | 1280 |
| weighted avg | 0.41 | 0.52 | 0.39 | 1280 |

Figure 8.6 Table showing classification results after Data Augmentation

## Learning Rate

The learning rate of the Adam optimiser was a hyper-parameter that we looked into tuning to get the most accurate model. The initial learning rate value was 1e-3 (0.001), so we changed this to ie-4 (0.0001) and 1e-2 (0.01) to evaluate the most optimal model.

The Keras callback API, ReduceLROnPlateau, was also incorporated into the CNN. This would reduce the learning rate when there has been a lack of improvement. We monitored to ‘val_accuracy’ and reduced the Learning Rate by a factor of 0.2, if 3 epochs passed without improvement.

```
reduce_lr = ReduceLROnPlateau(monitor = 'val_accuracy',
                              factor = 0.2,
                              patience = 3,
                              verbose = 1,
                              min_delta = 0.0001)
```

Figure 8.6 Code snippet of ReduceLROnPlateau in Keras

## Learning Rate of 1e-4

This learning rate seems to have not improved our model, as it caused an early stop at epoch 15, with a loss of 83%, accuracy: 81%, precision: 68%, and recall: 49%. This learning rate change has not been effective, mainly shown by the decrease in accuracy from the 95% previously.

```
Epoch 15/100
40/40 [==============================] - ETA: 0s - loss: 0.8357 - accuracy: 0.8158 - precision: 0.6844 - recall: 0.4887 - auc: 0.8688
Epoch 15: val_accuracy did not improve from 0.79434
40/40 [==============================] - 60s 1s/step - loss: 0.8357 - accuracy: 0.8158 - precision: 0.6844 - recall: 0.4887 - auc: 0.8688 - val_loss: 0.9234 - val_accuracy: 0.7939
Epoch 15: early stopping
```

Figure 8.7 Image showing end results when training our model after varying learning rate to 1e-4

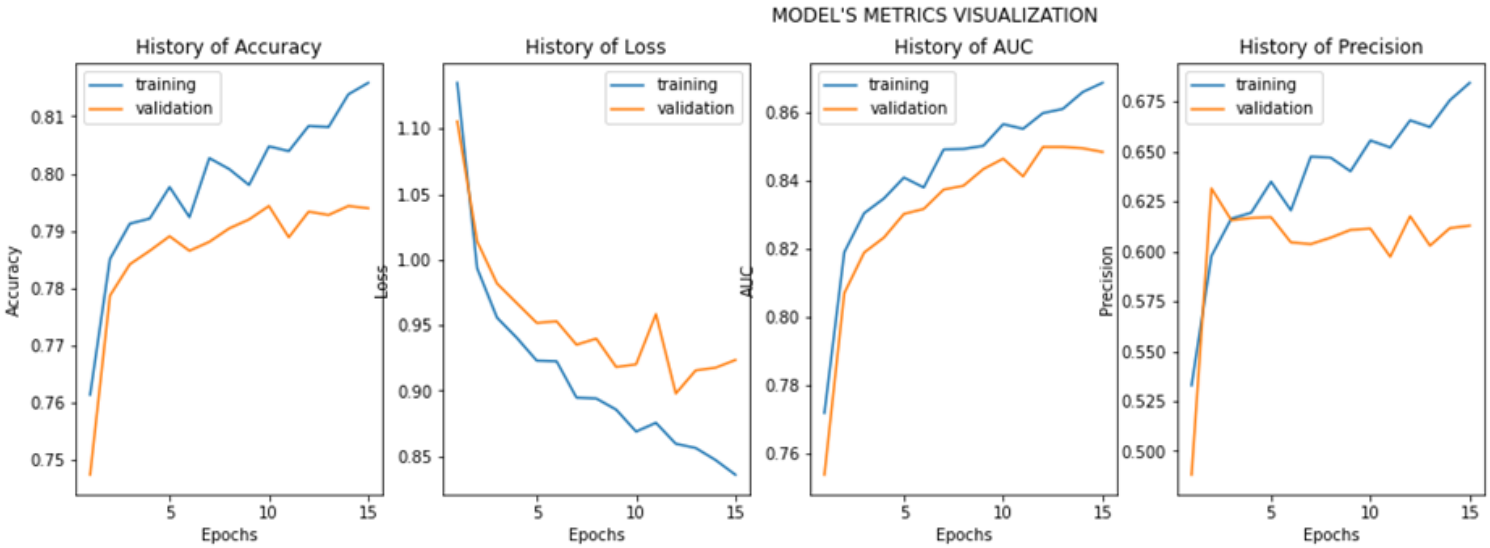


Figure 8.8 Graphs showing the history of accuracy, loss, AUC, and precision for training and validation data after varying learning rate to 1e-4

## Learning Rate of 1e-2

This learning rate seems to have not improved our model again, as it caused an early stop at epoch 19, with a loss of 82%, accuracy: 80%, precision: 76%, and recall: 32%. This learning rate change has not been effective, mainly shown by the decrease in accuracy from the 95% previously.

```
Epoch 19/100
40/40 [==============================] - ETA: 0s - loss: 0.8250 - accuracy: 0.8053 - precision: 0.7625 - recall: 0.3211 - a
Epoch 19: val_accuracy did not improve from 0.80313
40/40 [==============================] - 59s 1s/step - loss: 0.8250 - accuracy: 0.8053 - precision: 0.7625 - recall: 0.3211
Epoch 19: early stopping
```

Figure 8.9 Image showing end results when training our model after varying learning rate to 1e-2

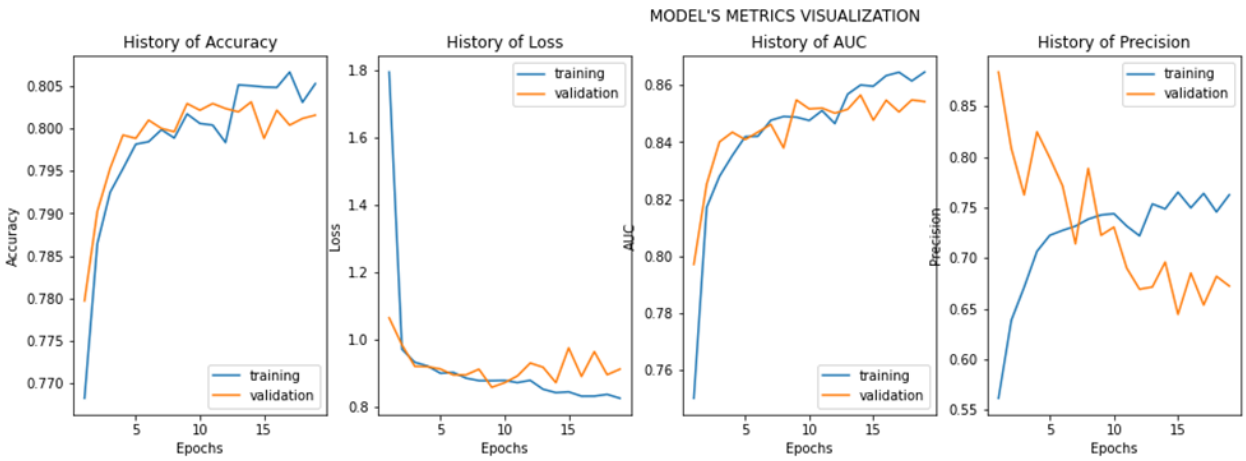


Figure 8.10 Graphs showing the history of accuracy, loss, AUC, and precision for training and validation data after varying learning rate to 1e-2

### Uniform vs Normal distribution

Changing to normal distribution seems to have not improved our model, as it caused an early stop at epoch 19, with a loss of 47%, accuracy: 90%, precision: 82%, and recall: 77%. This learning rate change has not been effective, mainly shown by the decrease in accuracy from the 95% previously.

```
Epoch 25/100
40/40 [==============================] - ETA: 0s - loss: 0.4707 - accuracy: 0.9017 - precision: 0.8275 - recall: 0.7666 - a
Epoch 25: val_accuracy did not improve from 0.85449
40/40 [==============================] - 56s 1s/step - loss: 0.4707 - accuracy: 0.9017 - precision: 0.8275 - recall: 0.7666
Epoch 25: early stopping
```

Figure 8.11 Image showing end results when training our mode after using normal distribution

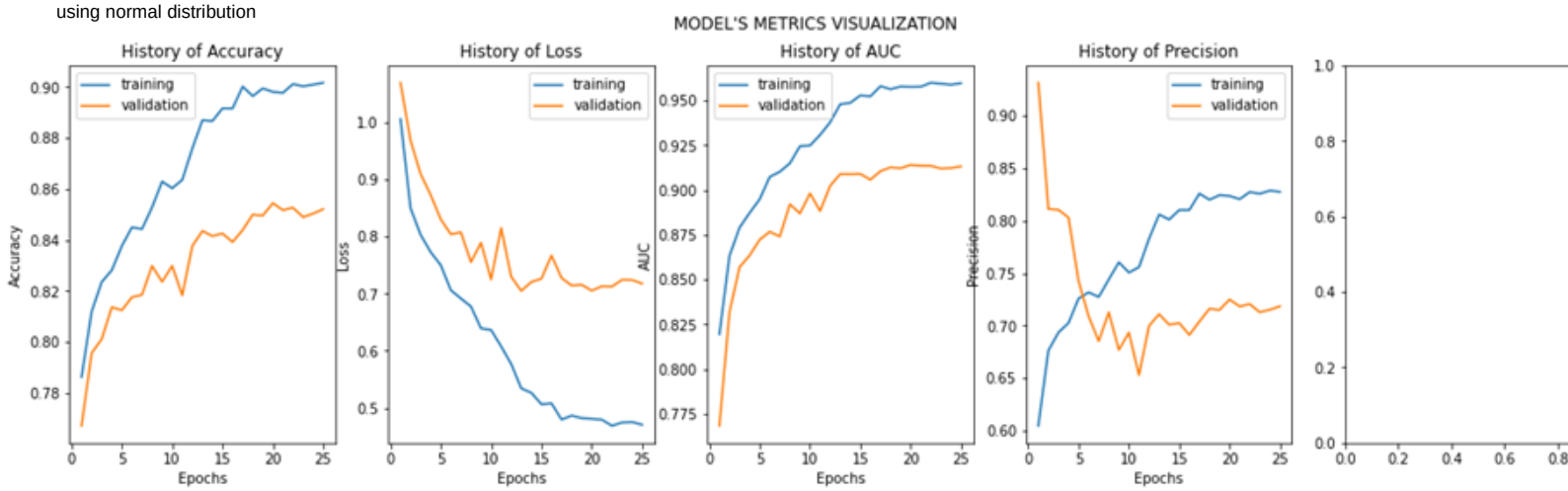


Figure 8.12 Graphs showing the history of accuracy, loss, AUC, and precision for training and validation data after using normal distribution

However, it seems to have helped in regards to the overfitting of the model, with a more accurate confusion matrix and testing accuracy of 54.3% (shown in Figure 8.x). So, we think that changing it to normal distribution would be helpful.

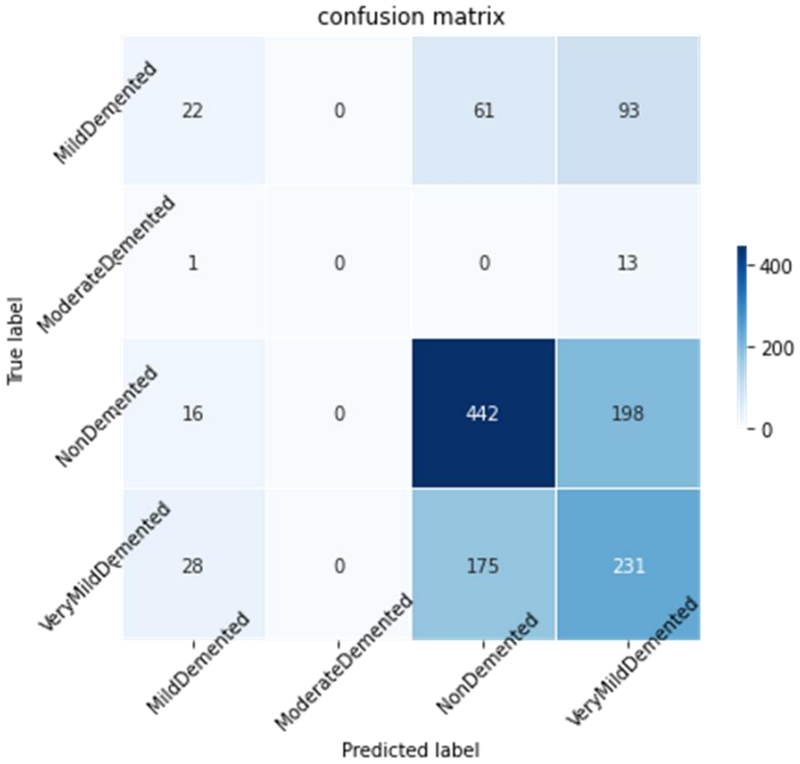


Figure 8.13 Confusion matrix after using normal distribution


Figure 8.14 Image showing F1 score for testing using normal distribution


Figure 8.15 Image showing accuracy score for testing using normal distribution

### Number of epochs

For the number of epochs, early stopping (which was described in Chapter 2) made it so that changing the number of epochs was obsolete as long as it is over the epoch where early stopping occurs. So, as a team we chose 100 epochs as no run ever made it that far or close to that.

## Regularisation

We decided to try L2 regularisation as In L2 regularisation we take the sum of all the parameters squared and add it with the square difference of the actual output and predictions. Similar to that in L1 regularisation the difference is that the weights will not be sparse, and we will get much better accuracy values compared to L1.[14]

We did have trouble with training our model as it frequently ran out of RAM while we had Regularization implemented. We did comment the code responsible for L2 out as we needed  our program to run for our final submission.

```
model.add(Dense(32, kernel_regularizer=l2(0.02), bias_regularizer=l2(0.
02)))
```

Figure 8.16 Code Snippet showing weight used for L2 Regularization

After implementing L2 regularization our model started of with 77% training accuracy and 69% validation accuracy.

```
Epoch 1: val_accuracy improved from -inf to 0.69316, saving model to
01.h5
40/40 [==============================] - 76s 2s/step - loss: 2.1754 -
accuracy: 0.7770 - precision: 0.5718 - recall: 0.4295 - auc: 0.8113 -
val_loss: 2.4000 - val_accuracy: 0.6932 - val_precision: 0.2186 -
val_recall: 0.0883 - val_auc: 0.5911 - lr: 0.0010
```

Figure 8.17 Image Showing our initial results during training after implementing L2 Regularization

It ran for  28 epochs before early stopping . Our final results were 90% accuracy for the training set and 86% for the validation. Although it didn't improve our overfitting issues for that metric it did for AUC , precision , loss and recall.

```
Epoch 28: val_accuracy did not improve from 0.86523
40/40 [==============================] - 56s 1s/step - loss: 0.5614 -
accuracy: 0.9058 - precision: 0.8388 - recall: 0.7713 - auc: 0.9635 -
val_loss: 0.7399 - val_accuracy: 0.8623 - val_precision: 0.7439 -
val_recall: 0.6852 - val_auc: 0.9274 - lr: 4.0000e-05
```

Figure 8.18 Image Showing our final results during training after implementing L2 Regularization

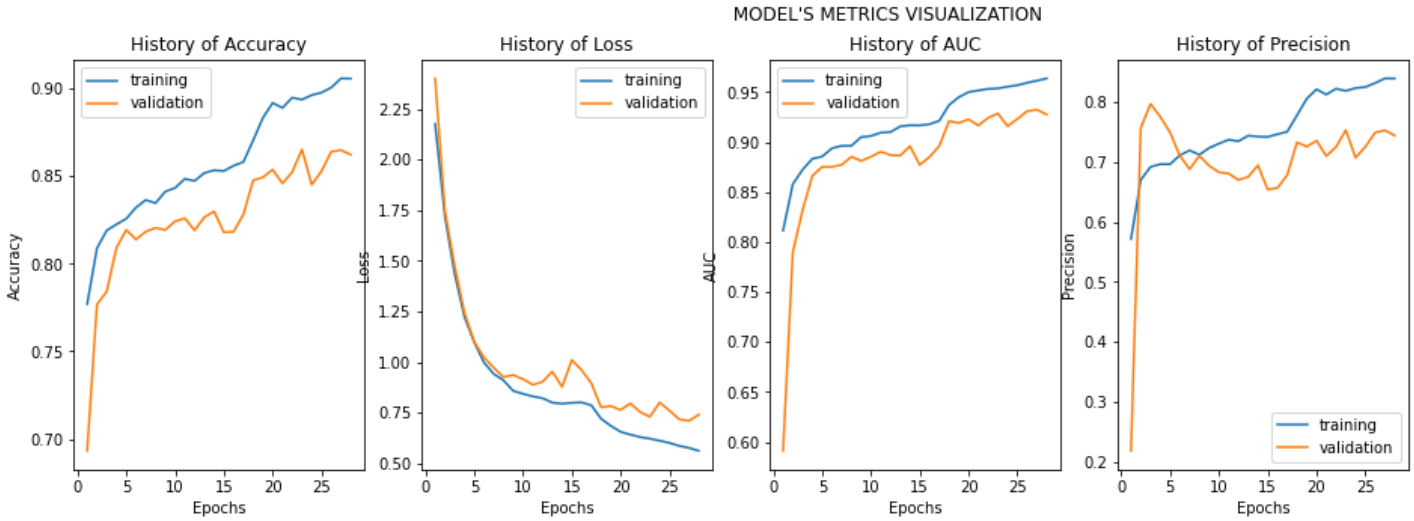


Figure 8.19 Graphs showing results for Accuracy, Loss, AUC and Precision for training and validation.

## References

https://www.kaggle.com/datasets/uraninjo/augmented-alzheimer-mri-dataset [1]

https://viso.ai/deep-learning/resnet-residual-neural-network/ [2]

https://datagen.tech/guides/computer-vision/resnet-50/ [3]

https://towardsdatascience.com/introduction-to-convolutional-neural-network-cnn-de73f69c5b83 [4]

https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79 [5]

https://www.statology.org/normal-vs-uniform-distribution/#:~:text=The%20normal%20distribution%20is%20bell,is%20equally%20likely%20to%20occur [6]

https://image-net.org/challenges/LSVRC/2015/results [7]

https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/ [8]

https://machinelearningmastery.com/k-fold-cross-validation/ [9]

https://www.baeldung.com/cs/k-fold-cross-validation#:~:text=Repeated%20k%2Dfold%20cross%2Dvalidation%20is%20a%20simple%20strategy%20that,times [10]

https://www.kdnuggets.com/2022/03/loss-functions-explainer.html [11]

https://peltarion.com/knowledge-center/modeling-view/build-an-ai-model/loss-functions/categorical-crossentropy[12]

https://gombru.github.io/2018/05/23/cross_entropy_loss/[13]

https://medium.com/intelligentmachines/convolutional-neural-network-and-regularization-techniques-with-tensorflow-and-keras-5a09e6e65dc7#:~:text=In%20L2%20regularization%20we%20take,L2%20will%20penalize%20the%20parameters.[14]