## Table of Contents

[**Dataset**](#_1kpyfbwx4k8k) **2**

[(a). Visualisation of key attributes](#_c6g145r3m5mb) 2

[(b) Preprocessing (Normalisation of the data)](#_dlhvsll8ptzt) 4

[**Creating Models**](#_2dof8gjsd6qi) **6**

[Loss Function](#_9yvdb0fmknl6) 6

[Optimizer](#_v1qqnjnves9l) 7

[**Results**](#_k5eu69rd5m8o) **7**

[Evaluation Of Results](#_3radvz5x9sqe) 9

[Varying Hyperparameters](#_3tmxrmlw32of) 10

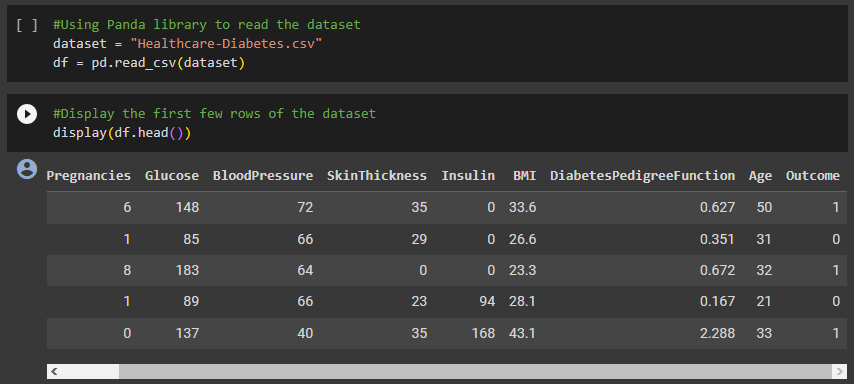
## 

## Dataset

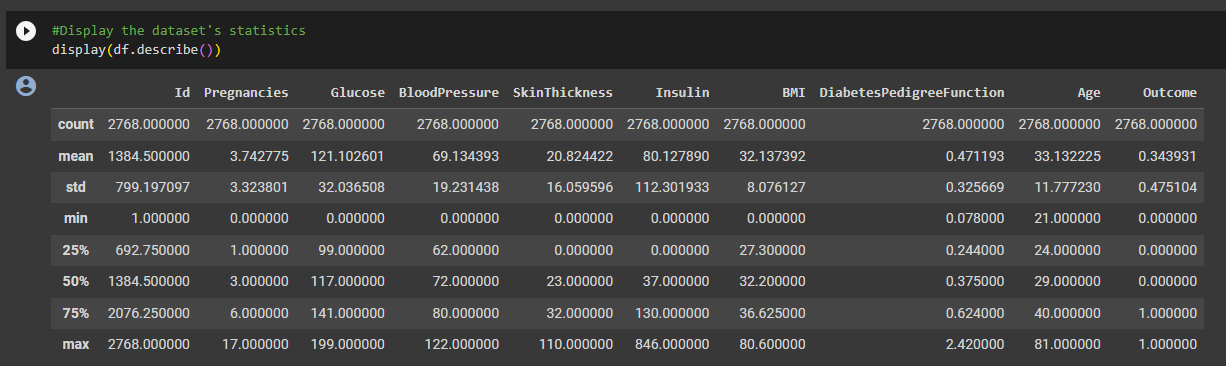
We used a diabetes prediction dataset from Kaggle. The dataset contains a range of health related attributes, that were collected to aid in the development of predictive models for identifying individuals at risk of diabetes. The attributes can be further described here:

1. **Id:** Unique identifier for each data entry.
2. **Pregnancies:** Number of times pregnant.
3. **Glucose:** Plasma glucose concentration over 2 hours in an oral glucose tolerance test.
4. **BloodPressure:** Diastolic blood pressure (mm Hg).
5. **SkinThickness:** Triceps skinfold thickness (mm).
6. **Insulin:** 2-Hour serum insulin (mu U/ml).
7. **BMI:** Body mass index (weight in kg / height in m^2).
8. **DiabetesPedigreeFunction:** Diabetes pedigree function, a genetic score of diabetes.
9. **Age:** Age in years.
10. **Outcome:** Binary classification indicating the presence (1) or absence (0) of diabetes.

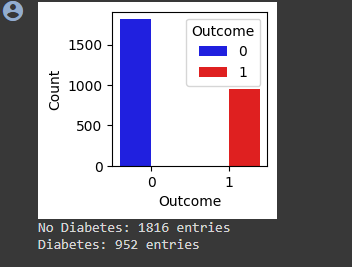
### (a). Visualisation of key attributes



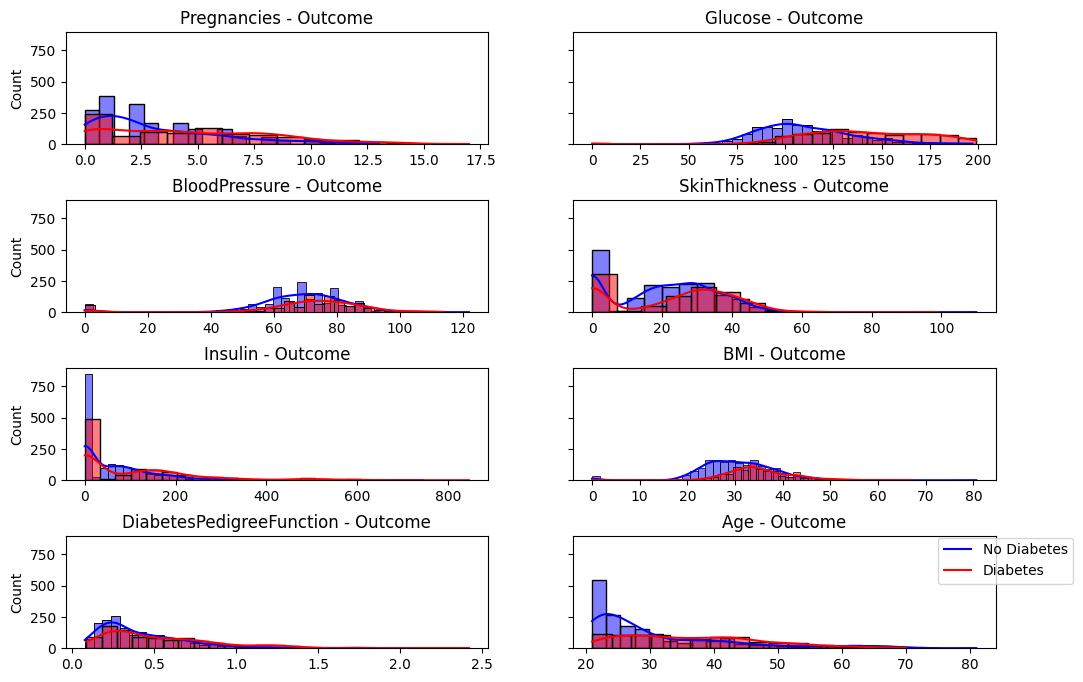
*Figure 1. Using Pandas Library to read the Dataset + Displaying the first few rows of the dataset.*



*Figure 2. Displaying a description of the data in the DataFrame.*



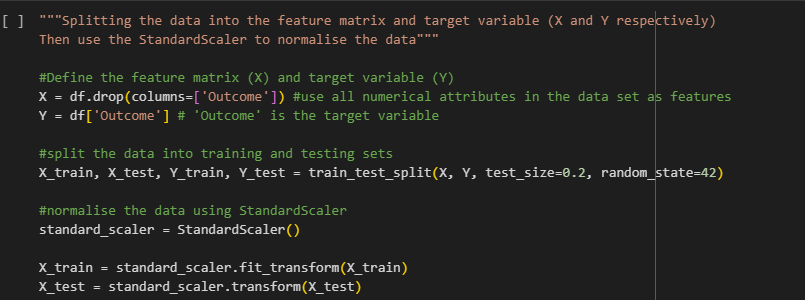
*Figure 3. A Visual Representation (Bar Graph) showing the total count of all outcomes where diabetes is not detected (0) or where it is detected (1)*



*Figure 4. A Visual Representation done for each visual attribute showing where diabetes is present or not*

### 

### (b) Preprocessing (Normalisation of the data)



*Figure 5. Data normalisation code snippet*

1. **Import Necessary Libraries**The first step in normalising the data was making sure the necessary libraries were imported for the operations. For this block **Pandas** was necessary for manipulating the data in the database’s DataFrame, **Scikit-Learn** or ‘**sklearn**’ for machine learning and more specifically **StandardScaler** to normalise the data.
2. **Defining the Feature Matrix and Target Variable  
    (**Next we define the feature matrix and target variable denoted by ‘X’ and ‘Y’ respectively. The line X = df.drop(columns=['Outcome']) extracts the feature matrix ‘X’ from the DataFrame ‘df’. It drops the ‘Outcome’ column which is recognised as the target variable and keeps the remaining numerical attributes in the DataFrame. ‘X’ will carry the value of all the features used for making predictions.

The line Y = df['Outcome'] extracts the target variable ‘Y’ from the ‘Outcome’ column of the DataFrame ‘df’. ‘Y’ will carry the values we want to predict.

1. **Split Data into Training and Testing Sets**The line train\_test\_split(X, Y, test\_size=0.2, random\_state=42) takes the feature matrix and the target variable (X, Y) and splits them into training and test sets. The parameters are as follows:

* ‘X’ and ‘Y’: The feature matrix and target variable, respectively.
* test\_size=0.2: This declares that 20% of the data will be used for testing, leaving the remaining 80% to be used for training the machine learning model.
* random\_state=42: This sets a random seed to ensure reproducibility of results. It’s used to initialise the random number generator (RNG), ensuring your receive the same split every time the code is run with the same value for ‘random\_state’

The result of this operation is four datasets:

* X\_train: the feature matrix for training
* X\_test: the feature matrix for testing
* Y\_train: the target variable for training
* Y\_test: the target variable for testing

1. **Normalising the Data**

The line standard\_scaler = StandardScaler() creates an instance of the StandardScaler class from Scikit-Learn. StandardScaler is a preprocessing technique used to standardise (normalise) the features, making sure they have a mean of 0 and a standard deviation of 1. This is necessary for a lot of machine learning algorithms to perform well.

The line X\_train = standard\_scaler.fit\_transform(X\_train) here calls the fit\_transform method on the training feature matrix X\_train. This method computes the mean and standard deviation of each feature in the training set, and then standardises the features using these statistics.

Finally the line X\_test = standard\_scaler.transform(X\_test) standardises the test testing feature matrix ‘X\_test’ using the same statistics calculated on the training set. This ensures that the testing data is scaled in the same way as the training data.

Running this block of code, properly separates our feature matrix and target variable. The feature matrix will be normalised, making it ready for use in the machine learning algorithm. The split into training and testing sets is crucial for evaluating the performance of the model.

### 

### 

### 

## Creating Models

#### Loss Function

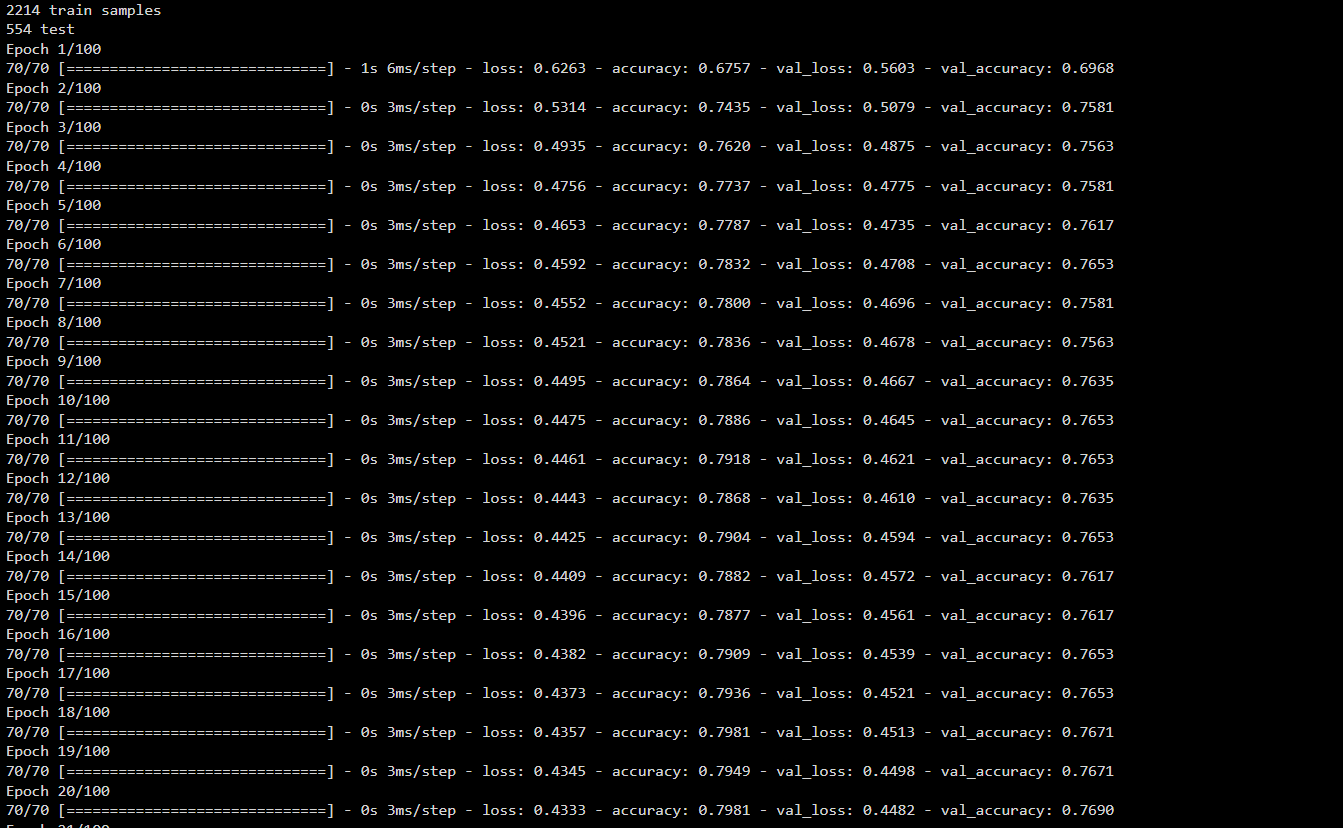
For our loss function we utilised binary cross entropy to calculate the loss between the true labels and the predicted probabilities.

In the Neural Network model we have 2 layers. In the first layer(hidden layer) we utilise the Rectified Linear Unit (RELU) Activation Function, that is commonly used in hidden layers of Neural Networks. In the output layer we use the Sigmoid Function , which allows the output to be between the values of 0 and 1.

#### Optimizer

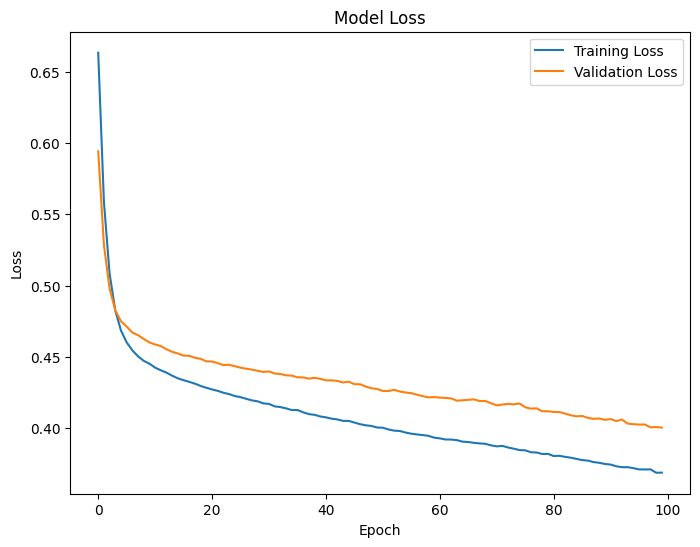
When compiling the model we used the Adam Optimizer function.This is an extension of the stochastic gradient descent (SGD) algorithm and is used to update the weights of the neural network during training.

## Results

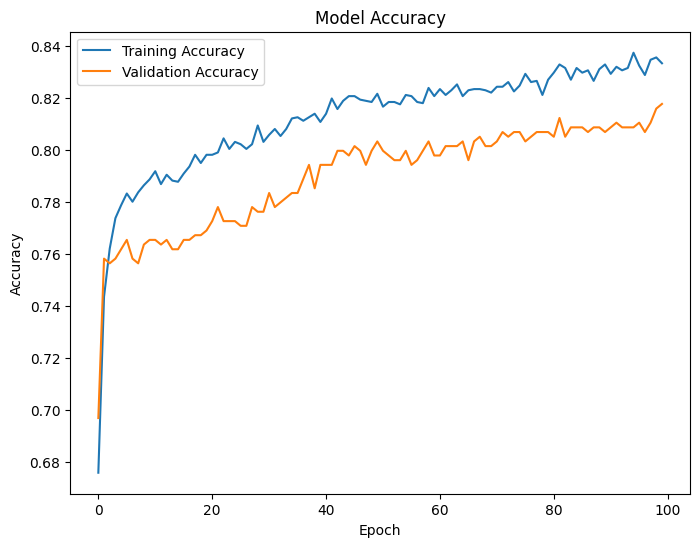


*Figure 6. The Results showing the number of training/test samples, the loss, accuracy etc.*

1. Plot of Results For Training/Validation Loss + Training/Validation Accuracy



*Figure 7. Plot of Training/Validation Loss Values*



*Figure 8. Plot of Training/Validation Accuracy Values*

#### Evaluation Of Results

HYPERPARAMETERS:

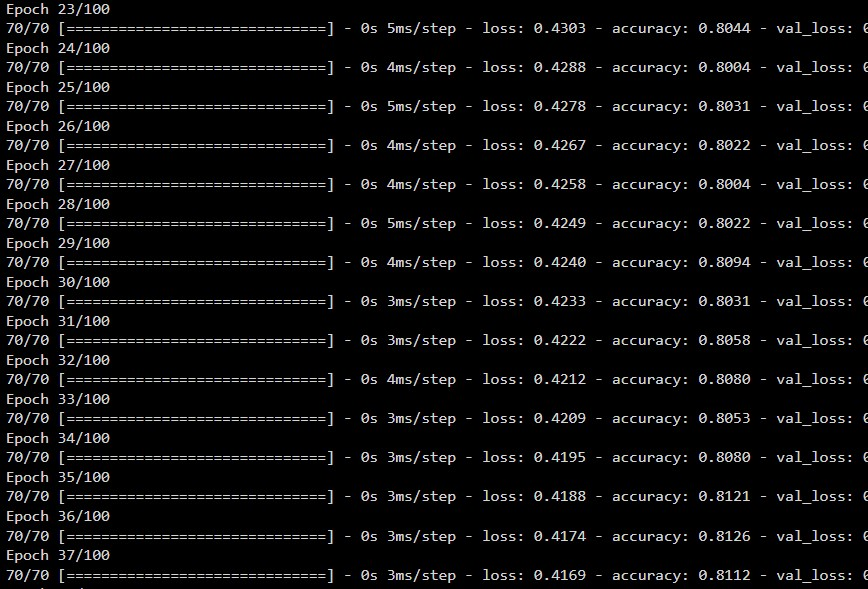
EPOCHS=100

BATCH\_SIZE=32

LOSS FUNCTION = BINARY CROSSENTROPY

If we analyse the curve in the Model Loss graph, we can see that the model initially has a high learning rate.

As we can see from the plotted graphs, the accuracy of the data gradually increases as its model runs. The accuracy is consistently above 70%, as you can see in the image below, with each instance being above .8 in this case. The accuracy peaks at 83%.



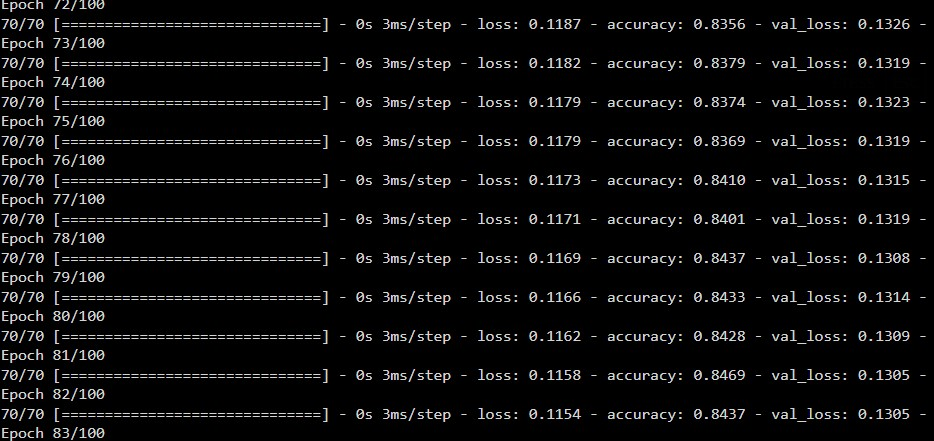
*Figure 9. The Loss and Accuracy Values from each Epoch*

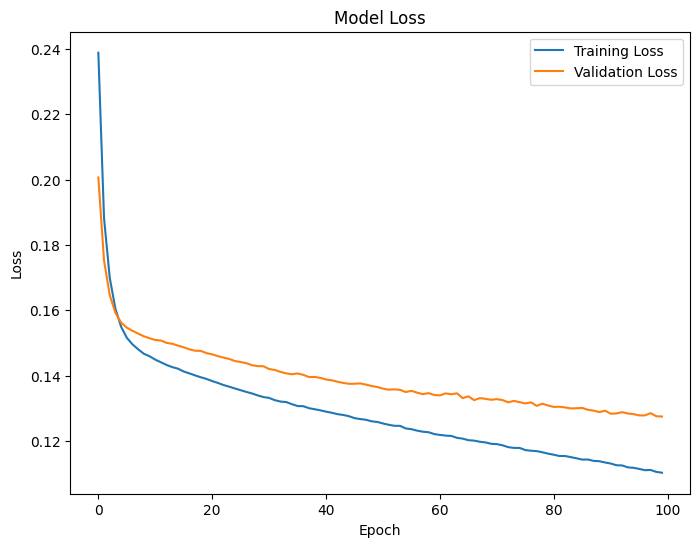
If we analyse the curve in the Model Loss graph, we can see that the model initially has a high learning rate. The model avoided a case of overfitting.

#### Varying Hyperparameters

**MSE VS Binary Crossentropy**

The first hyperparameter we decided to change was the loss function.We changed the loss function from Binary Cross Entropy to Mean Square Error and noticed a significant decrease in training and validation loss.

*Figure 10. The Training and Loss Validation Values using MSE as Loss Function*

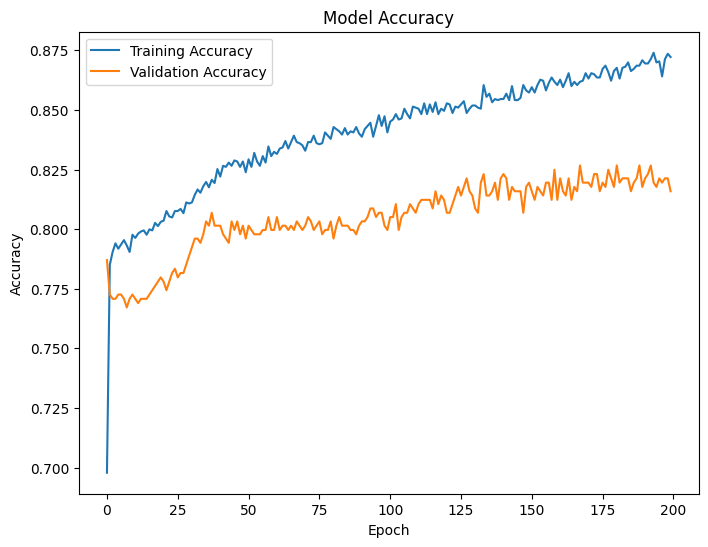
***Figure 11. Graph displaying Validation/Training Loss using MSE*

Comparing the values between Figure 9 and Figure 10, we can see the significant decrease in Loss value when using the MSE loss function compared to when we use Binary Crossentropy as the loss function.

We did not observe a significant difference in terms of accuracy when we changed the loss function.This would indicate that the model is better-trained and generalises better when using Mean Squared Error as the Loss Function.

**Epochs 100 vs Epochs 200**

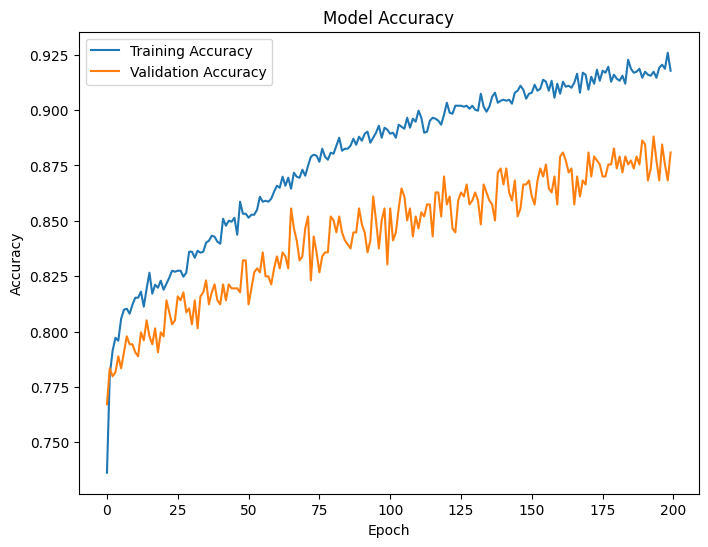
In this condition we increased the number of epochs compared to the original results and took note of the results. We noticed that with the increase in epochs, the system generates a higher accuracy for the training and validation data which indicates an improved learning. Yet the model loss remained the same.



*Figure 12. Graph displaying Validation/Training Accuracy using Epochs of 200*

**Batch Size = 32 vs Batch Size = 2**

Here we decided to change the hyperparameter of Batch Size from the original 32 used in the results to 2 and investigate the outcome.We had originally changed the Batch Size to 100 but did not notice a significant difference in performance.However when we changed the Batch Size to 2 there was significant changes. These were both ran using an epochs of 200.



*Figure 13. Graph displaying Validation/Training Accuracy using Batch Size 2*

As you can see if you compare Figure 12 and Figure 13, we notice a significant increase in the training and validation accuracy of the data with the Batch Size of 2 while the Batch Size of 32.However this came at a cost as the larger batch was 10 times quicker in executing than the smaller batch.