­­

Applying Computational intelligence techniques for diabetic retinopathy detection

Module Title: Computational intelligence WRIT1

Module Code: CIS6005

Academic Year: 2022-23

Author: Linton Bryan-Fox

Student Number: 20164763

Contents

[Part 1 - Neural Network without Machine Learning Libraries 3](#_Toc129797931)

[Neural Network Class 3](#_Toc129797932)

[Justification of Activation Function 6](#_Toc129797933)

[Read ARFF file for the Dataset and converted it to CSV – Justify why we dropped the columns 6](#_Toc129797934)

[NULL values and Dropping columns 7](#_Toc129797935)

[Normalising the Data 7](#_Toc129797936)

[Split the data into training sets & Encode the target variable 8](#_Toc129797937)

[Training the neural network 9](#_Toc129797938)

[Results 10](#_Toc129797939)

[Hidden Neuron Tests 11](#_Toc129797940)

[12 Hidden Neurons 11](#_Toc129797941)

[20 Hidden Neurons 12](#_Toc129797942)

[30 Hidden Neurons 12](#_Toc129797943)

[13](#_Toc129797944)

[Analysis of the Neural Network without Libraries 13](#_Toc129797945)

[Part 2 - Neural Network with Machine Learning Libraries 14](#_Toc129797946)

[Hidden Neurons 30 16](#_Toc129797947)

[Hidden Neurons 20 16](#_Toc129797948)

[Hidden Neurons 12 17](#_Toc129797949)

[Comparisons 18](#_Toc129797950)

[Optimisations 19](#_Toc129797951)

## Part 1 - Neural Network without Machine Learning Libraries

### Neural Network Class

Firstly, to create the program we need to consider how the neural network will be organised. This is a simple multi layered feedforward neural network with a hidden player when can be changed depending on the test. It is initialised using the input and output data (X & Y), the number of hidden neurons, the learning rate, and the number of training epochs.

The class overall provides methods to calculate the sigmoid & softmax activation functions, to perform forward and backward propagation, train the network and to make predictions. The forward method calculates the neural network output given some input data, while the backward method updates the weights and biases using backpropagation.

During the training process, the train method will update the weights and the biases at each epoch and prints the loss for that. Finally, the predict method can be used to make predictions for the new input data. Below you will find a picture of the code in question:

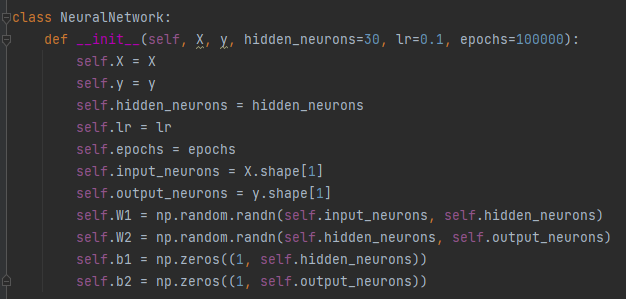


Figure Neural Network from Scratch – init

Text

Description automatically generated

Figure Neural Network - Activation Functions

Text

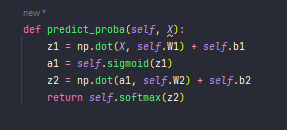
Description automatically generated

Figure neural Network - Forward & Backward propagation

Text

Description automatically generated

Figure Neural Network - Train Method

Text

Description automatically generated

Figure Neural Network - Predict Method

Figure Neural Network - Predict Proba

### Justification of Activation Function

I have the choice of two major activation functions, Sigmoid and softmax. They are two frequently used activation functions in neural networks. However, they have distinct characteristics and are ideal for different types of tasks.

The sigmoid function is utilized to transform any input value into a range between 0 and 1, which can be interpreted as a probability. It is commonly employed in binary classification problems where the objective is to predict if a given input belongs to one of two categories. The sigmoid function is often used in the output layer of the model to produce a probability estimate for the positive class.

In contrast, the softmax function is employed to transform any input value into a range between 0 and 1, and it also normalizes the output so that the sum of all the values is equal to 1. This makes it ideal for multi-class classification problems where the goal is to predict which category a given input belongs to among multiple classes. The softmax function can be utilized in the output layer to generate a probability distribution over all the possible classes. Considering our given problem at hand is binary classification, where the objective is to predict if a patient has diabetic retinopathy or not. As a result, the sigmoid function is an appropriate choice for the output layer. On the other hand, using softmax would be unsuitable since it is intended for multi-class classification, which is not the scenario in this case.

### Read ARFF file for the Dataset and converted it to CSV – Justify why we dropped the columns

This code first converts a ARFF file using Pandas, with some additional parameters such as `header=None` & `comment=@`. Since the ARFF file is a file used for machine learning it first needs to be converted for this reason, after reading in the ARFF file, it then gives it Header names that will be used for the output CSV file. This is because the ARFF file does not have any column names and instead uses metadata to define the attributes. When declaring the header names, as I did not know the dataset, I used the metadata it gave. Once I got that metadata using WEKA then I was able to create a list of header names appropriate for file.

The code then reads in the newly created CSV file using Panadas and assigns the values of the `Output` column to the target variable `y` as it will be used as a target variable for the machine learning model and not a feature.

Text

Description automatically generated

Figure Neural Network - ARFF to CSV & Drop target Variable from data

### NULL values and Dropping columns

The code snippet here starts using a loop to iterate over each column in the dataframe, the loop uses `iloc` to select all the columns expect the first 2 and the last column. The reason it does this was that the last column is just the target variable and the first two columns had nothing to do with the dataset and its overall outcome. In the remaining columns the code counts the number of null values inside of it and if those null values is greater than 25% of the total number of rows , which s calculated as 1151 \* 0.25, the column is then dropped as this data is not good enough to use for the machine learning purposes as we cannot fill the null values with the mean score as the data will be filled with mean scores of 25% and won’t actually represent the data very well and could lead to inaccuracies.

Once all relevant columns are dropped the code snippet then iterates over each row and if the value is null & the number of nulls is less than 25% it will calculate the mean score for that column and then insert the mean score into it. This snippet identifies and handles the missing values in the dataframe, by implementing this code, the accuracy and the reliability of the results can be improved, making it a valuable tool for the data cleaning and analysis.

Text

Description automatically generated

Figure Neural Network - NULL value and Dropping Columns

### Normalising the Data

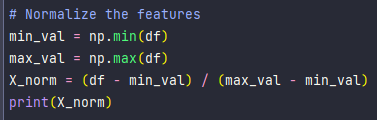
This snippet shows the Normalizing of the data, this is done to scale the data to a common range, we do this to avoid any biases towards the features. The code itself calculates the minimum and maximum values for each column in the dataframe using numpy `min` & `max` functions. The it applies the normalization formula of `(x + min)/(max – min)`, to each value in the dataframe. The result is the new dataframe of `X\_norm` with the same shape as `df` but with each feature to the scale between 0 and 1. By normalising the data, we can ensure that each feature contributes equally to the proposed models’ predictions.

Figure Neural Network - Normalising Data

### Split the data into training sets & Encode the target variable

This is basic method/function for a machine learning program – by splitting the dataset into training and test sets and one-hot encoding the target variable.

The X\_train and y\_train contains the corresponding target variable values. Similary, X\_test and y\_test represent the test data, where X\_test contains the inout features, and y\_test contains the corresponding target variable values. The code itself uses the Pandas `loc` method to select the target variable values for the training and test sets from the original dataset using the corresponding indices.

The next part is the OneHotEncoder function, which is used to convert the categorical target variable into a binary vector representation, where each category is represented by a column and each row has a 0 or 1 indicating the presence or absence of that category. The resulting encoded target variables are stored in y\_train and y\_test, this is to convert the target variables into numerical format that can be used to train and evaluate the machine learning model. The resulting encoded variables are used as target variables in the training and testing of the machine learning model.

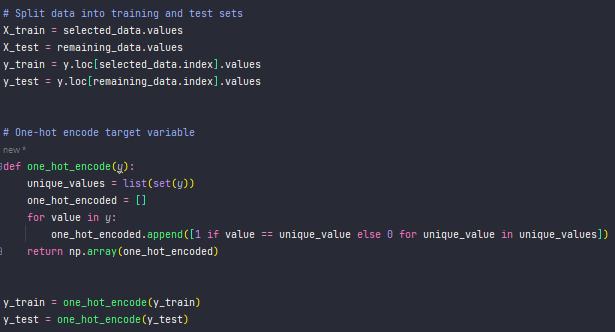


Figure Neural Network Splitting data & Encoding the target variable

### Training the neural network

This is one of the most important parts of the code, it trains the neural network model using the input features in X\_train and the target variable in y\_train. The train method of the NeuralNetwork class is called to perform the training, once the neural network is trained it is used to make predictions on the test data in X\_test using the predict method, and the resulting predictions are stored in y\_pred to be used later.

The one-hot encoded predictions in y\_pred and true labels in y\_test are converted back to class labels using the argmax method of numpy. A confusion matrix is then constructed using the confusion\_matrix method from scikit-learn. The confusion matrix is used to calculate the true positive rate, false positive rate, and accuracy of the model on the test data. These performance indices are stored in the tp\_rate (True Positive Rate), fp\_rate (False Positive Rate), and accuracy variables, respectively.

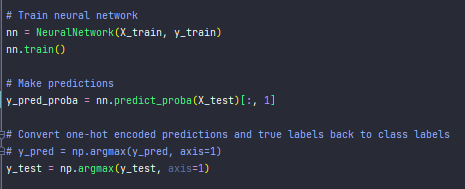


Figure Training the Neural Network

### Results

The final code snippet is the evaluation metrics of the trained neural network, including the confusion matrix. It uses the tabulate library to create a table to organise the metrics.

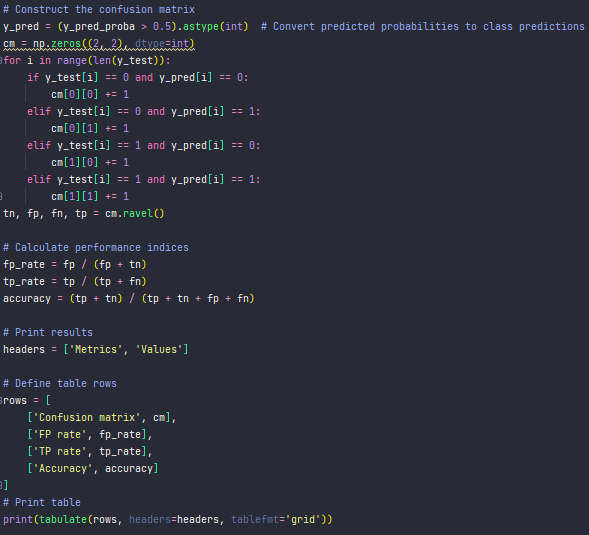


Figure Neural Network – Results

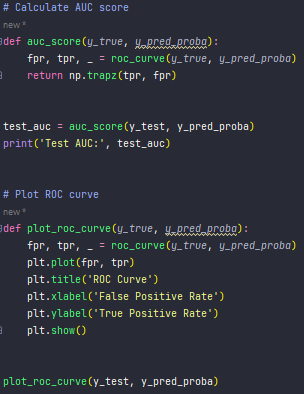


Figure Neural Network without libs - Calculate ROC score

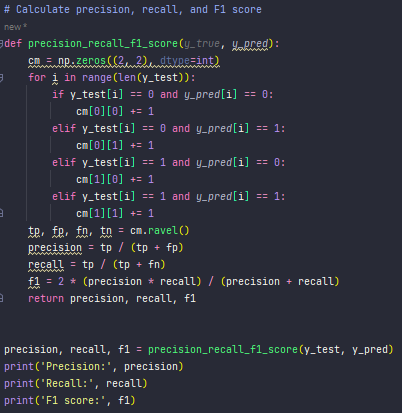


Figure neural Network without LIBs- Confusion matrix

## Hidden Neuron Tests

To compare the performance of three neural network classifiers with different numbers of hidden neurons, I needed to evaluate their accuracy on the dataset we are using. We can use metrics such as true positive rate, false positive rate, confusion matrix, and accuracy to compare their performance against each other.

After training the three classifiers and evaluating them on the dataset, we can compare their performance based on the chosen metrics. It is likely that the classifier with more hidden neurons (30) will have higher accuracy than the others, but it may also be prone to overfitting. The classifier with fewer hidden neurons (12) may have lower accuracy but may generalize better to new data.

To justify my observations, I can analyse the training and validation curves of the classifiers and check for signs of overfitting or underfitting. We can also try tuning the hyperparameters of the classifiers, such as learning rate and regularization, to improve their performance as mentioned before. Below you will find 3 pictures of the Neural Network being tested with 12, 20 & 30 hidden neurons all with the learning rate of 0.1 and running on 100,000 epochs – Below is their confusion matrix and a graph plotting the training curve of MSE vs epoch.

### 12 Hidden Neurons

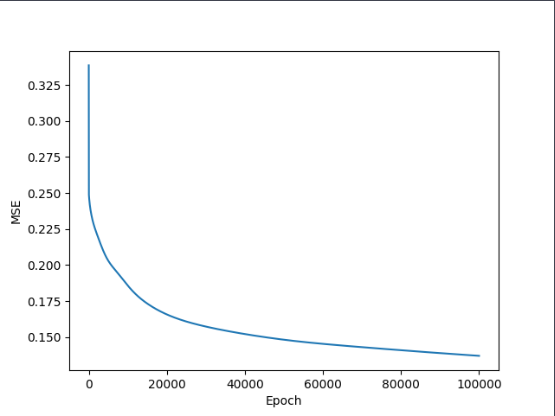
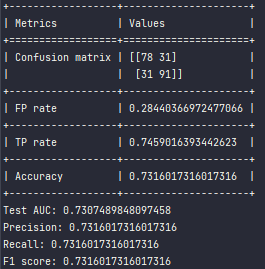
 For the neural network with 12 hidden neurons, the accuracy is 0.73, which indicates that it correctly classified 73% of the instances. The false positive rate (FP rate) is 0.28, which means that 28% of the instances that did not have diabetic retinopathy were incorrectly classified as having the condition. The true positive rate (TP rate) is 0.74, which means that 74% of the instances that actually had diabetic retinopathy were correctly identified as having the condition. Finally, the precision, recall, and F1 score are all measures of the model's performance. In this case, all three metrics have the same value of 0.7316 or 73.16%. This indicates that the model has similar performance in terms of precision, recall, and F1 score.

Figure MSE vs Epoch of 12 Hidden Neurons

Figure Confusion Matrix of 12 Hidden Neurons

### 20 Hidden Neurons

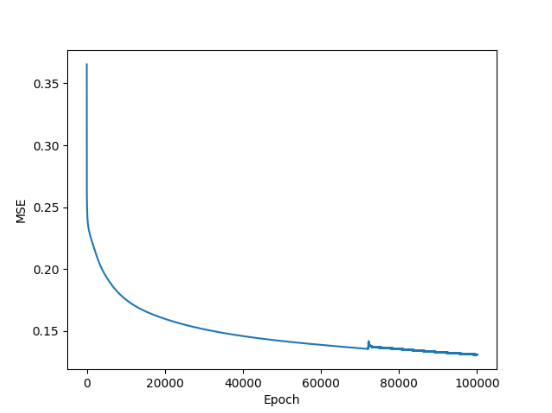
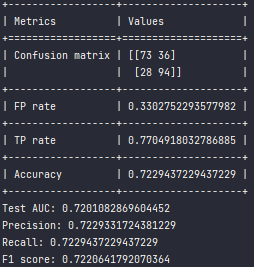
 For the neural network with 20 hidden neurons, the accuracy is 0.72, which is slightly worse than the neural network with 12 hidden neurons. The false positive rate is higher at 0.33, indicating that more instances without diabetic retinopathy were incorrectly classified as having the condition. However, the TP rate is slightly higher at 0.77, indicating that some instances of diabetic retinopathy were not correctly identified. The precision of the model is the ratio of true positives to the total number of samples predicted as positive, which is 0.72 in this case. The recall of the model is the same as the TP rate (0.77), which is the ratio of true positives to the total number of positive samples. The F1 score is the harmonic mean of precision and recall, which is 0.72 in this case. Overall, the metrics suggest that the model is performing reasonably well, but it may be useful to further investigate the misclassified samples and consider ways to improve the model's performance.

Figure MSE vs Epoch of 20 Hidden Neurons

Figure Confusion Matrix of 20 Hidden Neurons

### 30 Hidden Neurons

For the neural network with 30 hidden neurons, the accuracy is better than the neural network with 20 hidden neurons only slightly better than the one with 12. The false positive rate is set at 0.27 and the TP rate is set at 0.75. The model itself had an accuracy of 0.74 which makes it the best of the three. Overall, the metrics suggest that the model is perofrming reasonalbly well. This suggests that lowering the number of hidden neurons beyond 30 did not improve the performance of the classifier at all making it the ideal choice.

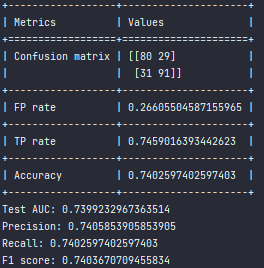
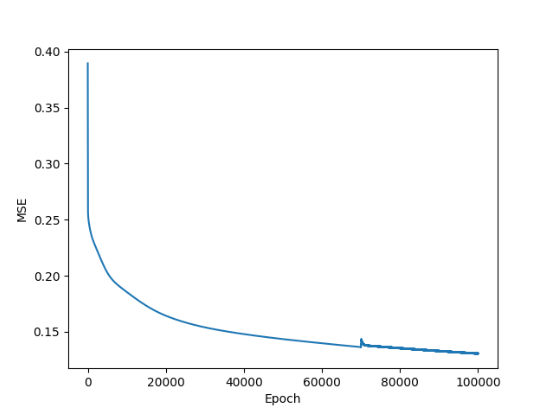


Figure Confusion Matrix of 30 Hidden Neurons

Figure MSE vs Epoch of 30 Hidden Neurons

## Analysis of the Neural Network without Libraries

The three outputs are presenting the results of different neural network classifiers with different number of hidden neurons.

When comparing the three models, we can observe that the model with 30 hidden neurons achieved the highest accuracy (74%), while the model with 20 hidden neurons achieved the lowest accuracy (72%). The model with 12 hidden neurons had an accuracy of 73% as well.

In terms of false positive (FP) rate and true positive (TP) rate, the model with 20 hidden neurons had the highest TP rate (77%) but also the highest FP rate (33%), which indicates that it is more likely to classify a normal case as positive. The model with 30 hidden neurons had the lowest TP rate (74%), indicating that it is less likely to classify a diabetic retinopathy case as positive.

Overall, the model with 30 hidden neurons seems to be the best choice, as it achieved the highest accuracy and a good balance between FP rate and TP rate.

## Part 2 - Neural Network with Machine Learning Libraries

For the final part of this report, I was tasked with the construction, training and testing another multilayer feedforward neural network with the same datasets prepared in Part 1 using any number machine learning libraries such as scikitlearn or Keras. I will make sure to detail the hyperparameters and training rules/algorithms adopted in the report. In doing so, this let’s me compare the performance of this model with the neural network classifier trained in Part 1 above. The performance metrics used include MSE, confusion matrix, TP rate, FP rate, and accuracy will be compared to this neural network. On top of this I will explore how some advanced measures like a AUC curve could have been used for performance evaluation of these neural network classifiers.

I will then tune the parameters, training rules/algorithms to re-train the model so that the testing performance of the model is improved. Once we have re-trained the model, I will explore more machine learning libraries and construct a new machine learning model using a different neural network / classifier architecture, like support vectors machines, decision tree or random forest, then I will compare the model I built to a machine library one.

#### Neural Network with Machine Learning Libraries CODE

The start of this code is the same as the other model until you get to the normalization of the data. I normalised the data here using the sklearn library using `StandardScaler`, which is a common technique to ensure that all the features have the same scale. The next part of this code snippet is how I split the data and then sampled it as per the specification of the assignment.

As the specification said we needed to reserve 80% of the instances for the neural network training and the other 20% of the instances are reserved for testing the performance of the trained neural network. With the data we first had to determine 80% of it, this was easy as we knew 80% of the total instances need to be selected therefore we use `train\_test\_split` of `sklearn.model\_selection` library to achieve this, in the code this was completed as specified by the `test\_size=0.2` argument in the `train\_test\_split function`.

Then we create a sequential model consisting of two dense layers. The first layer has 20 units and uses the Relu activation function which is different from the Neural Network without libraries. This was done as it helps the model to learn more complex and non-linear relationships between the input features and the target variable. The input shape for this layer is the number of features in the input data. The second layer has a single unit and uses the sigmoid activation function. This is a binary classification problem, so the output of the model should be a single probability value between 0 and 1, which is why the sigmoid activation function here.

Next was to compile the model and then train it, in the code below this line compiles the model by specifying the optimizer, loss function, and evaluation metrics `model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])`. We used the `Adam optimizer` and we also used the binary class-entropy loss function, this was used as it is a binary class problem and the evaluation metric is accuracy. The most important part of this code is the training method, the X\_train and y\_train data are used for training here, and the validation\_split parameter is set to 0.2, which means that 20% of the data is used for validation during training. The verbose parameter is set to 1, which means that progress bars are displayed during training. The model is trained for 10,000 epochs with a batch size of 32.

The last parts of the code evaluate the model which simply returns the test loss and test accuracy. Then we make predictions on the test set using the predict method. Also, then I would calculate the confusion matrix and accuracy score using scikit-learns confusion\_matrix and accuracy\_score functions. The final part of the code plots the training and validation loss versus epochs using the mapplotlib and, I then calculated the AUC score and plotted the ROC curve using scikit-learn’s roc\_auc\_score & roc\_curve functions.

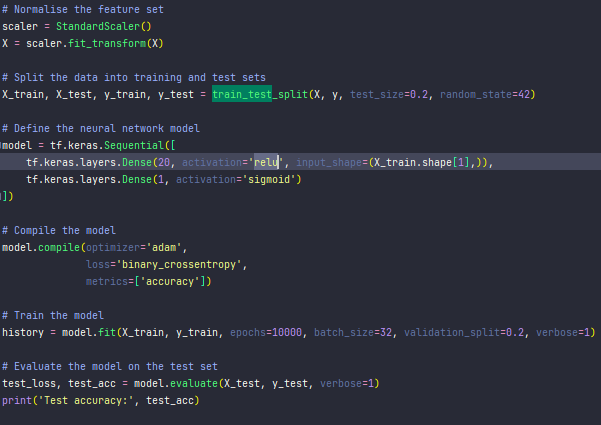


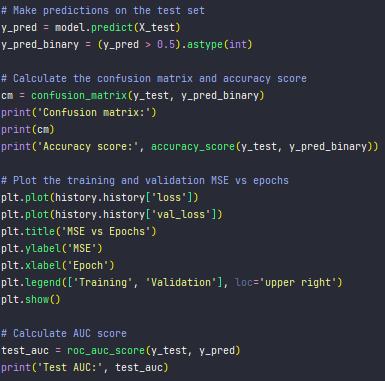
Figure Neural Network with Machine learning libraries

Figure Neural Network with Machine learning libraries

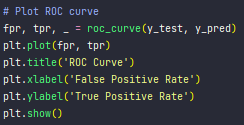


Figure Neural Network with Machine learning libraries

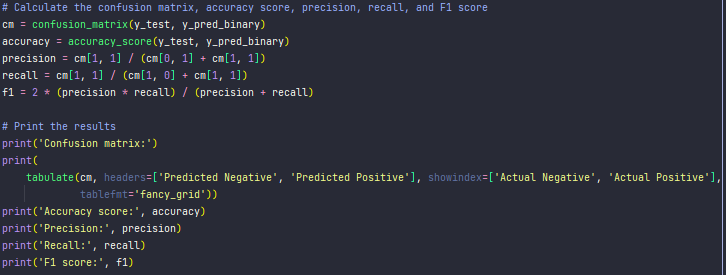


Figure neural Network with Machine Learning libraries confusion matix

#### Results

### Hidden Neurons 30

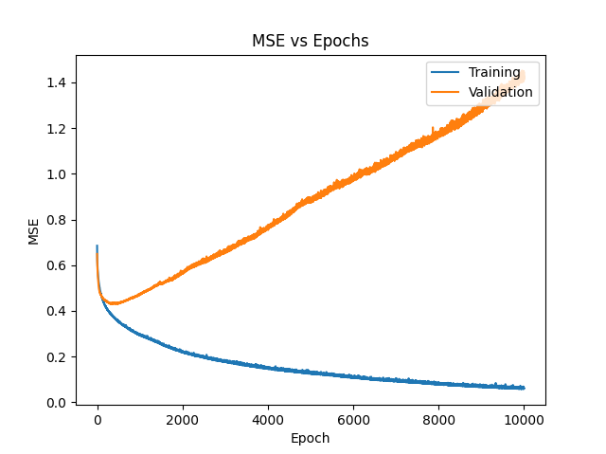
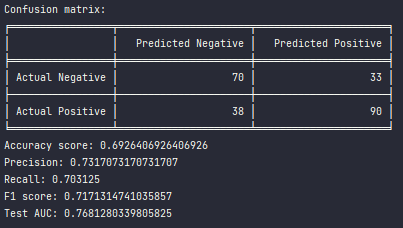
As you can see MSE vs Epochs for 30 hidden neurons looks great compared to the validation set that wasn’t trained and as you can see the MSE continues to rise as the epochs continue to increase. In the confusion matrix it HAS predicted 70 true negative cases and 33 false positive cases, and we have predicted 38 false negatives and 90 true positive cases, the accuracy for this was 0.6926, which means the model correctly classified 69.26% of the total cases.

Figure MSE vs Epochs for 30 hidden Neurons

Figure Confusion Matrix for 30 Hidden Neurons

### Hidden Neurons 20

Out of the total number of samples the confusion matrix shows that it correctly predicted 78 negative samples and 92 positive samples, while it incorrectly predicted 36 positive samples as negative and 25 negative samples as positive. The total accuracy of the classifier was 0.73, which means that it correctly classified 73% of the total samples. Therefore, the performance of this classifier is better than the one with 30 hidden neurons, however further improvements can be made here also by adjusting the parameters of the classifier or just simply using different algorithms.

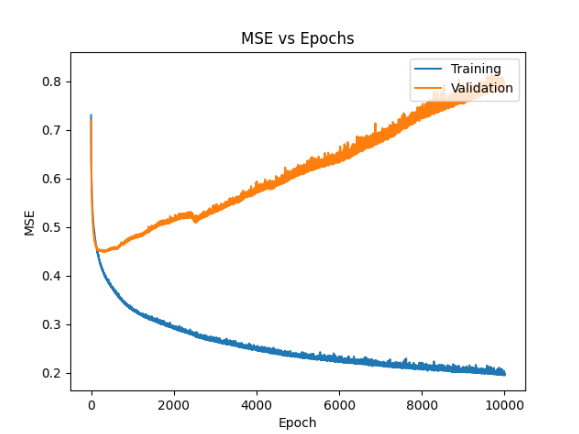
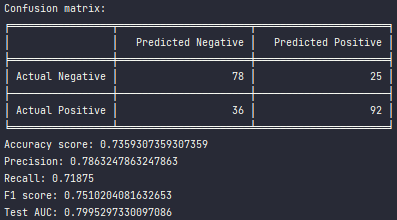


Figure Confusion Matrix for 20 Hidden Neurons

Figure MSE vs Epochs for 20 Hidden Neurons

### Hidden Neurons 12

The overall accuracy was 0.74 which means it correctly predicted the class for 74% of instances. In the confusion matrix in the figure below it shows the classifier predicted positive for 122 instances and negative for 113 instances. Out of the 78 actual negative instances, the classifier correctly predicted 78, but incorrectly predicted 25 as positive. Out of the actual positive instances, the classifier correctly predicted 93, but incorrectly predicted 35 as negative. Overall, this is the best classifier to use as it has a great accuracy than the other two and its FP and TP rates are as balances as they can be between the 3 of them, as always there is room for improvement, especially in reducing the false positive prediction which in this case was quite high.

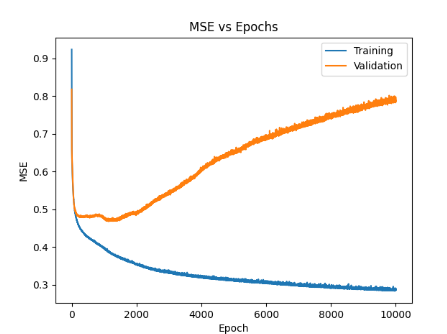
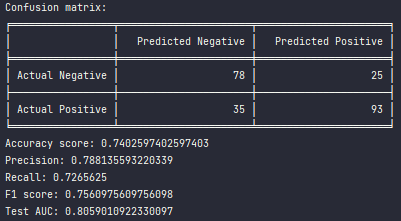


Figure Confusion Matrix for 12 Hidden Neurons

Figure MSE vs Epochs for 12 Hidden Neurons

## Comparisons

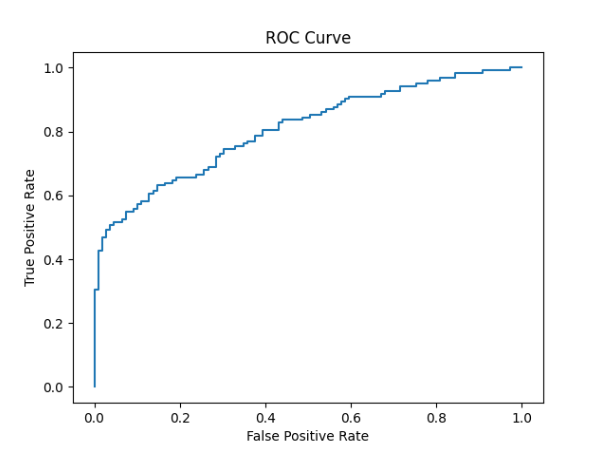
In this section we will compare the performance of the model with Libraries to the model without libraries, we will take the best model trained over X number of Hidden neurons and compare those to the machine learning counterpart using the MSE vs Epoch Curve, the confusion matrix and finally an added test of the ROC curve.

In FIG [16] in Part 1 we can see that this was the best model to use as its accuracy and confusion matrix was the best out of the three that were tested and used, the model has a true positive rate of 74% and a false positive rate of 26%. The accuracy of the model is 74%. Whereas the best model that was using Machine Learning libraries is FIG [26], this model had accuracy of 71%.

Comparing the confusion matrices:

* FIG [16] matrices have the are better accuracy of 074% over the 71% for FIG [26].
* FIG [26] has a higher precision (0.7881) and F1 score (0.75609) compared to FIG [16] (precision: 0.7405, F1 score: 0.7403).
* FIG [16] has a higher recall (0.74025) compared to FIG [26] (recall: 0.7265625).

#### ROC Curve

In FIG [29] is the Neural Network with machine learning libraries, as you can see when the ROC curve started out it hit a TPR of around 0.4 – 0.5 slowly and sluggishly, it started to even out when it hits around 0.8 TPR & 0.6 FPR. Whereas the Neural network without libraries hit 0.6 TPR within the 0.2 of the FPR and starts to even out at 0.4 FPR. Overall, the Model without libraries is the best choice as it is closer to 1 of the AUC than Fig [32] is, and because FIG [32] is closer to 0.5 it is the less effective classifier disguising between them both.

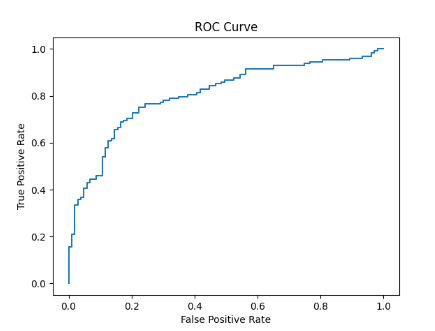


Figure Neural Network with NO libraries & 30 hidden neurons

Figure neural Network with Machine learning libraries & 12 Hidden Neurons

Since the primary goal is to accurately identify as many cases of diabetic retinopathy as possible, a model with a high recall may be preferred – and going by the ROC curve the neural network with no machine learning trained with 30 hidden neurons is the best. This overall down to be a better choice due to its higher recall. But what if the goal was to minimize the number of false positive predictions, then FIG [26] with a high precision may be preferred.

# Optimisations

The performance metric to optimize is the binary cross-entropy loss function, specified as the "loss" argument in the model compilation step of the neural network:

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

The binary cross-entropy loss function is commonly used in binary classification problems, where the goal is to minimize the difference between the predicted probability distribution and the true probability distribution of the target variable. And then model is also optimizing for accuracy, as specified in the "metrics" argument of the model compilation step. Therefore, the objective of the neural network model in this code is to minimize the binary cross-entropy loss function while maximizing the accuracy of the model.

As it goes to optimising variables, there’s a few things to consider here below is a list variable we can change / adjust to improve the performance of the network:

1. Learning rate: This evidently controls the weights tat are updated during the training, a higher learning rate can result in faster tests, but it could cause the model to overshoot the optimal weights.
2. Number of Neurons: Increasing the number of neurons in a layer can allow the model to lean more complex patterns in the data, but to many neurons will cause overfitting.
3. Activation functions: Different activation functions like ReLu, Sigmoid or tanh can be used in different layers to introduce nonlinearity into the model meaning our network can better approximate complex functions and model nonlinear relationships between inputs and outputs.
4. Dropout: This is a function where it randomly drops out some of the neurons during the training to prevent overfitting.
5. Batch Size: The number of samples processed in each training batch can affect the overall speed and stability of the actual training process.

By adjusting these variables, we can find the optimal combination that results in the best performance of the neural network.

The final optimisation I would make would be to use a crossover operator and mutation operator, which are two key genetic operators used in genetic algorithms, these operators can be used to optimize the neural network architecture and hyperparameters, such as the number of hidden layers, number of nodes per layer, learning rate, and activation functions.

The crossover operator can be used to combine the genetic information of two or more neural network models, such as their weights and biases, to create a new and potentially better performing model. This can be done by randomly selecting a crossover point in the chromosome and swapping the corresponding weights and biases between the parent models to create a child model.

The mutation operator can be used to introduce random changes to the weights and biases of the neural network model. This can be done by randomly selecting a weight or bias in the chromosome and modifying it by a small random value.

By applying these genetic operators to the dataset, we can search for the best performing neural network model that can accurately classify retinopathy cases. This can help in early detection and prevention of diabetic retinopathy, which can significantly improve patient outcomes.