ITI106 – Assignment

1 Introduction

In this project, we develop a solution to classify which genre a dry bean belongs to. We will use a fully connected, feedforward neural network with Keras to identify which class a Dry Bean falls under. This is a multi-class classification problem meaning that there are more than two classes to be predicted, in fact there are 7 dry bean species.

2 Dataset

For the purpose of this assignment, we use the "Dry Bean Dataset" from the UCI Machine Learning Repository [1]. The total number of samples in this dataset: **13,611** and the total number of attributes were **17**. The dataset did not contain any missing values. However, there were **68** duplicated rows. Duplicated entries were removed, after which the total remaining samples came to: **13543**.

Table 1 shows the dataset features with their associated descriptions.

Features	Type	Description
Area (A)	int64	The area of a bean zone and the number of pixels within its boundaries
Perimeter (P)	float64	Bean circumference is defined as the length of its border
Major axis length (L)	float64	The distance between the ends of the longest line that can be drawn from a bean
Minor axis length (l)	float64	The longest line that can be drawn from the bean while standing perpendicular to the main axis
Aspect ratio (K)	float64	Defines the relationship between L and l
Eccentricity (Ec)	float64	Eccentricity of the ellipse having the same moments as the region
Convex area (C)	int64	Number of pixels in the smallest convex polygon that can contain the area of a bean seed
Equivalent diameter (Ed)	float64	The diameter of a circle having the same area as a bean seed area
Extent (Ex)	float64	The ratio of the pixels in the bounding box to the bean area
Solidity (S)	float64	Also known as convexity. The ratio of the pixels in the convex shell to those found in beans
Roundness (R)	float64	Calculated with the following formula: (4piA)/(P^2)
Compactness (CO)	float64	Measures the roundness of an object: Ed/L
ShapeFactor1 (SF1)	float64	
ShapeFactor2 (SF2)	float64	
ShapeFactor3 (SF3)	float64	
ShapeFactor4 (SF4)	float64	
Class (Label)	object	One of: Seker, Barbunya, Bombay, Cali, Dermosan, Horoz and Sira

Table 1 – Dry Bean Dataset Features with Description

Figure 1 shows an image of the 7 different types of dry beans that our model will try to classify.

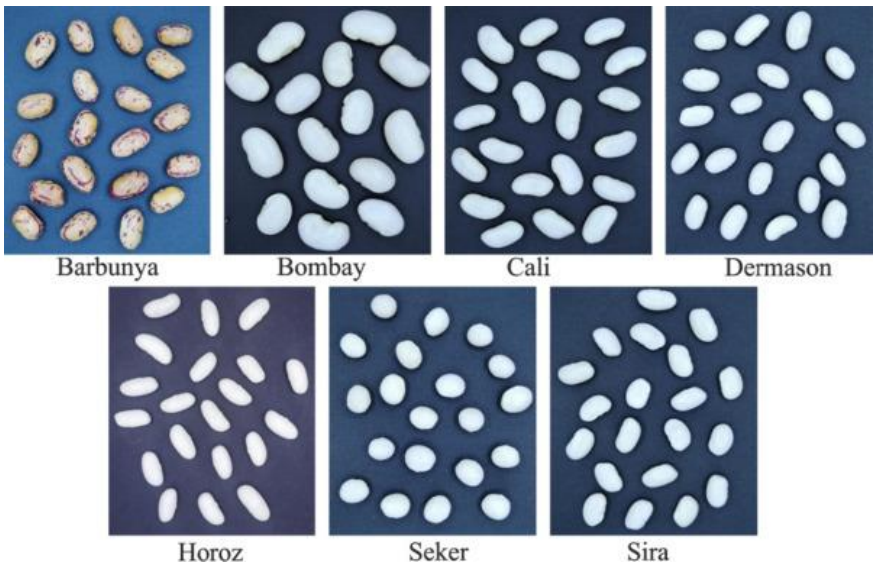


Figure 1 – Different Dry bean Types

Figure 2 shows the plot of the frequency counts of the 7 dry bean types in the dataset. Table 2 shows the frequency counts of the different dry beans.

Dry Bean	Frequency Count
DERMASON	3546
SIRA	2636
SEKER	2027
HOROZ	1860
CALI	1630
BARBUNYA	1322
BOMBAY	522

Table 2 – Frequency Counts of different Dry Beans in Dataset

Since the values of the features vary widely, we apply standard scaling to all the features. This will transform your data such that its distribution will have a mean value 0 and standard deviation of 1. In addition to transforming all the training data, the target column has to be transformed too since this is a multi-class problem. The output attribute has to be reshaped from a vector containing values for each class value to a matrix with a Boolean for each class value and whether or not a given instance has that class value or not. This is done by first encoding the strings consistently to integers using the '**LabelEncoder**' method. The vector of integers is then converted using one hot encoding.

The dataset is then split into train and test sets in an 80:20 ratio.

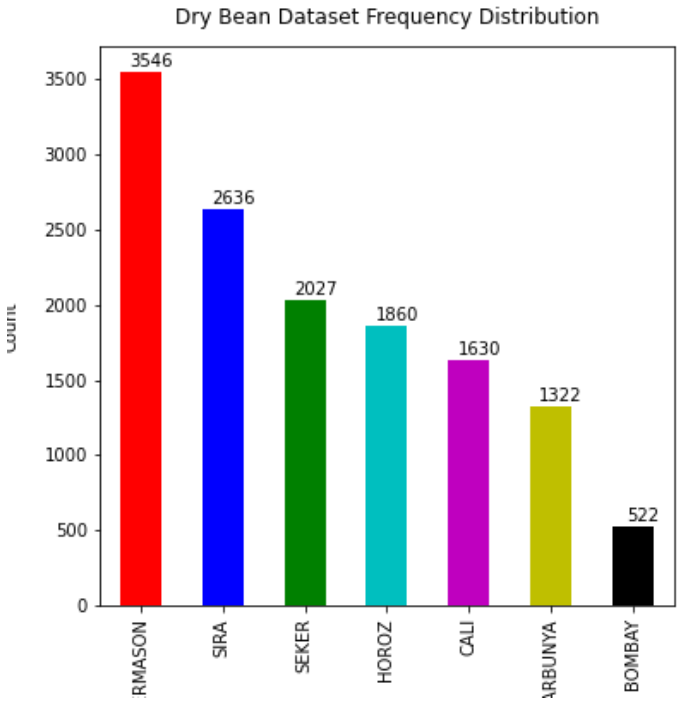


Figure 2 – Count of Dry Bean distribution in dataset

Once exploratory data analysis and pre-processing is performed on the dataset, we proceed to develop the fully connected feedforward neural network to build a classifier to determine which class of dry bean an unknown sample belongs to.

We will use the Tensorflow/Keras 2.x framework to build the classifier.

In order to get reproducible results in Keras, random numbers will have to be seeded. Since Keras gets its source of randomness from Numpy's random number generator, it will have to be seeded by calling the `seed()` function at the top. Likewise for Tensorflow. It has its own random number generator that must also be seeded by calling the `set_random_seed()` function immediately after the NumPy random number generator.

3 Hyperparameter Tuning: Hidden Layers, Number of Neurons in Hidden Layers, Batch Sizes and Activation Function

We start off with a baseline model. An initial feedforward neural network is defined with the correct number of input features. The final layer will have X output classes, one for each output class. The output value with the largest value will be taken as the class predicted by the model. Next, we determine the number of hidden layers and the number of nodes in each of the hidden layers.

The number of hidden layers can be determined using the following rules of thumb [2]:

0 - Only capable of representing linear separable functions or decisions.

1 - Can approximate any function that contains a continuous mapping from one finite space to another.

2 - Can represent an arbitrary decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any accuracy.

In addition to the number of hidden layers, deciding on the number of neurons in the hidden layers is also very important in deciding the overall neural network architecture. If too few neurons are used in the hidden layers, this could result in underfitting – which makes it difficult to adequately detect the signals in a complicated data set. If there are too many neurons in the hidden layers, this can result in overfitting. This occurs when the neural network has so much information processing capacity that the limited amount of information contained in the training set is not enough to train all of the neurons in the hidden layers. Also having a large number of neurons in the hidden layers would increase the computational time for training the network. The following rules of thumb can be used to determine the number of nodes in the hidden layers [3] and these are:

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be $\frac{2}{3}$ the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.

I use grid search in this experiment to determine the number of hidden layers and nodes, activation function and batch size. No cross-validation value was specified so this was defaulted to using the 5-fold 'StratifiedKFold' method as this is a multi-class problem.

# Hidden Layers	# Nodes
1	10
2	12, 8
2	11, 9
3	14, 10, 9

Table 3 – Number of Hidden Layers and Nodes in Hidden Layers

Parameters	Values
Batch Sizes	64, 128, 256
Activation Functions	'relu', 'selu', 'elu'

Table 4 – Batch Sizes and Activation Functions

For the loss function, I use '**categorical_crossentropy**' as this is a multi-class classification problem. The '**softmax'** function is used as the activation function in the output layer of neural network to predict a multinomial probability distribution.

ReLU addresses the vanishing gradient problem. However, it also introduces the 'dead relu' problem, where components of the network are most likely never updated to a new value. It does not avoid the exploding gradient problem

Exponential Linear Unit (ELU) avoids the dead RELU problem. It produces negative outputs, which helps the network push the weights and biases in the right directions. It takes longer to compute because of the exponential operation.

Scaled Exponential Linear Unit (SELU) is one of the newer activation functions and addresses the vanishing and exploding gradient problem.

The resulting run of this experiment is shown in Table 5.:

Parameters	Values

Activation function	SELU
Batch Size	64
Layers	3
Nodes (Hidden Layers)	14,10, 9

Table 5 – Result of Grid Search

A call to **model.summary()** was used to print a useful summary of the model, which includes: the name and type of all layers in the model. This is shown in Figure 3:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 14)                238
_________________________________________________________________
activation (Activation)      (None, 14)                0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                150
_________________________________________________________________
activation_1 (Activation)    (None, 10)                0
_________________________________________________________________
dense_2 (Dense)              (None, 9)                 99
_________________________________________________________________
activation_2 (Activation)    (None, 9)                 0
_________________________________________________________________
dense_3 (Dense)              (None, 7)                 70
=================================================================
Total params: 557
Trainable params: 557
Non-trainable params: 0
_________________________________________________________________
```

Figure 3 – Model summary

Figure 4 shows the resulting neural network architecture after hyper parameter tuning. This was generated using the tool [4], which was recommended in [5].

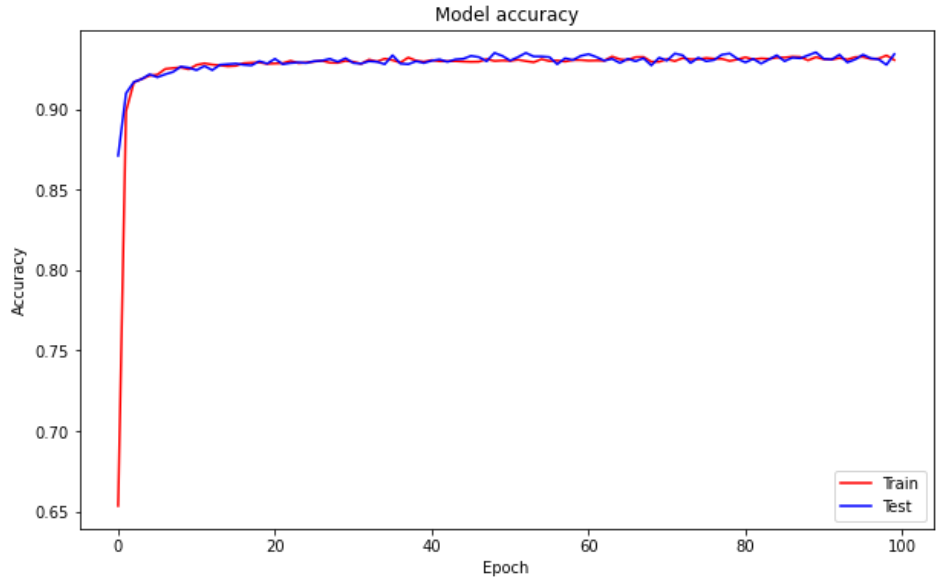
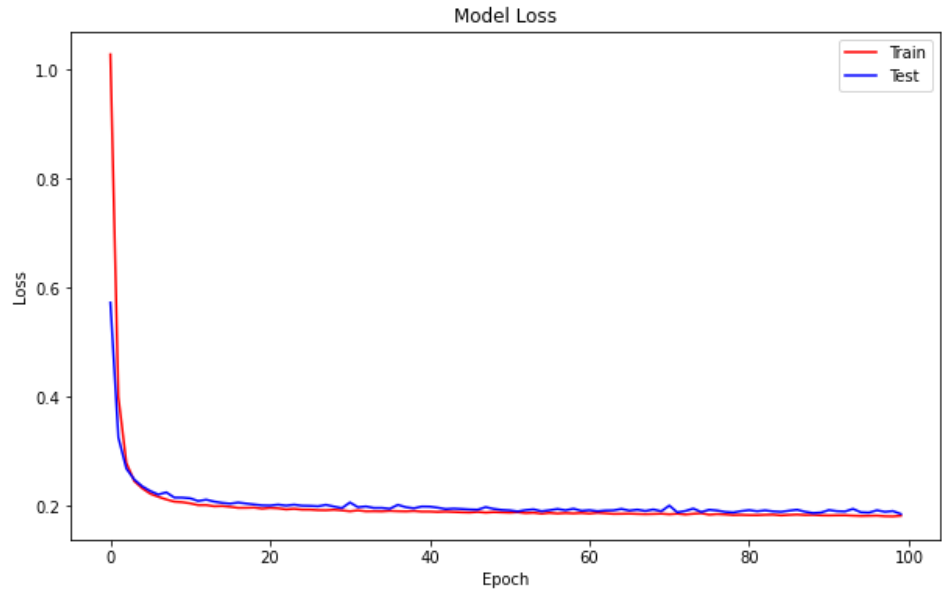
Plot Type	Plot
Epoch vs Accuracy	
Epoch vs Loss	

Table 6 – Plots showing model loss and accuracy after applying the parameters (Table 5) from the experimental run

The gap between training and validation accuracy is a clear indication of overfitting. The larger the gap, the higher the overfitting.

In the model accuracy plot, since the gap between the train and test/validation gap is very small, this means there is little overfitting.

In the case of the loss curve, the goal of the learning algorithm is to get an optimal fit. Model loss will almost always be lower on the training dataset than the validation dataset. This means that we should expect some gap between the train and validation loss learning curves. This gap is denoted as the generalization gap.

An optimal fit [6] is one where:

- The plot of training loss decreases to a point of stability.
- The plot of validation loss decreases to a point of stability.
- The generalization gap is minimal (nearly zero in an ideal situation).

Overfitting is apparent in a loss curve when:

- training loss continues to decrease with increasing training, and
- validation loss has decreased to a minimum and has begun to increase.

In a loss curve, underfitting can show up as:

- a flat line or noisy values of relatively high loss (for the training curve)
- It can also be identified by a training and validation loss that are continuing to decrease at the end of the plot. This means that the model is capable of further learning and that the training process was halted prematurely.

Both learning curves indicate a near optimal fit.

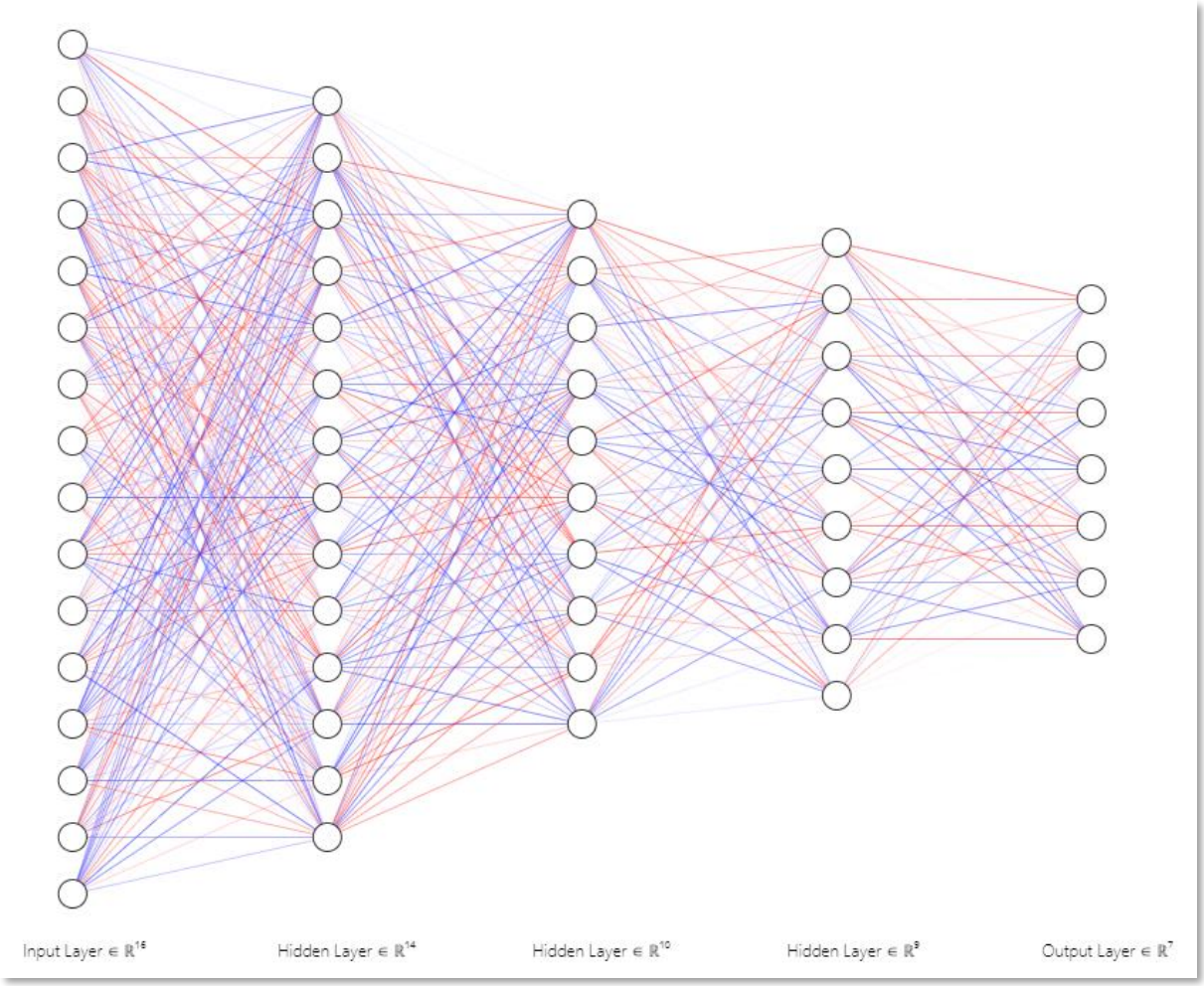


Figure 4 – Neural Network Architecture

4 Hyperparameter Tuning: Determining the Regularization Parameter

The next set of parameters that I wanted to tune was the regularization parameter. The different values used are shown in Table 7.

Regularization Parameters	Values
L1	l1=0.01
L2	l2=0.01
L1_l2	l1=0.01, l2=0.01

Table 7 – Regularization Parameters

Given that all the other parameters are the same (see Table 5), the results of this run recommended the '**L2 Regularizer**'.

Plots of the model loss and accuracy after applying L2 regularization are shown in Table 8.

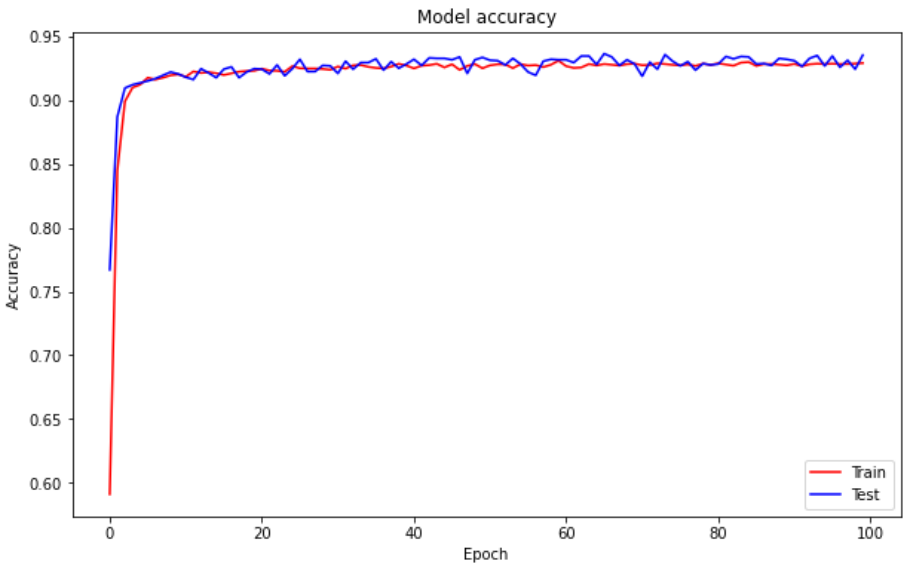
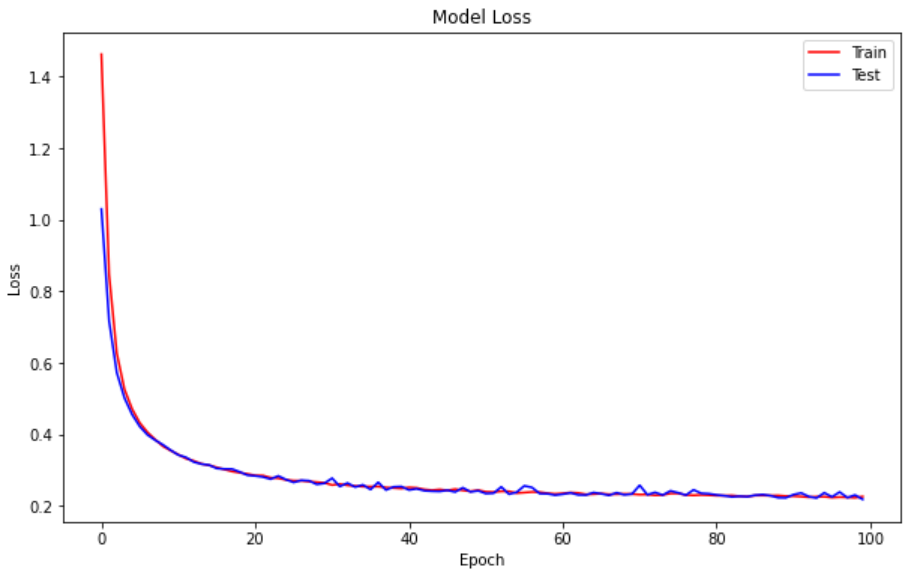
Plot Type	Plots
Epoch vs Accuracy	
Epoch vs Loss	

Table 8 – Plots showing model loss and accuracy after applying L2 Regularization

Again, the learning curves shown in Table 8 show a near optimal fit, with little overfitting or underfitting.

5 Hyperparameter Tuning: Determining the Dropout values

I next tried experimenting with adding dropout. The other parameters remained as determined in Sections 3 and 4 and are summarized here again for easy reference:

Parameters	Values
Activation function	SELU
Batch Size	64
Layers	3
Nodes (Hidden Layers)	14, 10, 9
Regularization Parameters	L2 0.01

Table 9 – 'Constant' Parameters used in this run

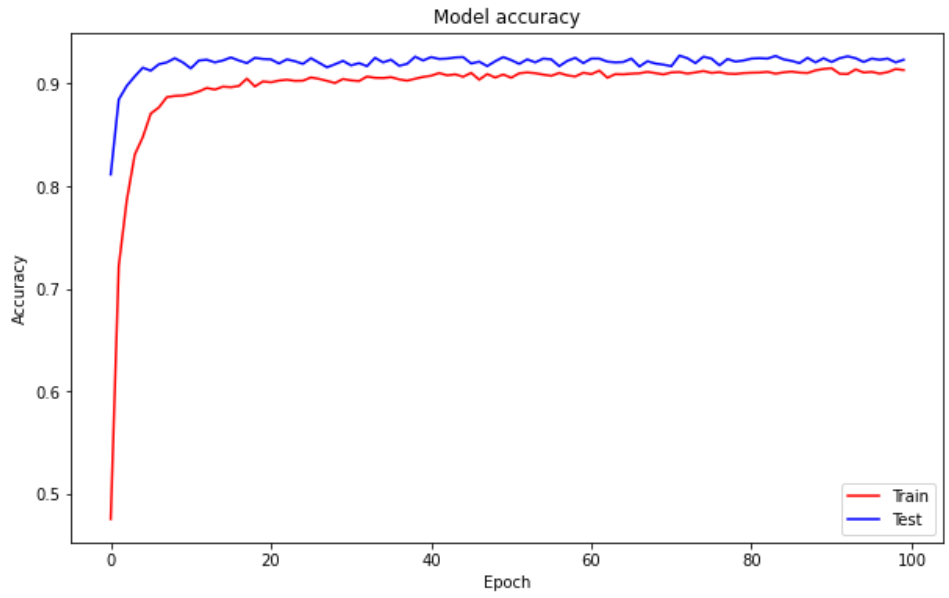
The dropout values used for this experimental run are shown in Table 10

Parameters	Values
Dropout	0.1, 0.3

Table 10 – Dropout Values

The results of running this experiment suggested using a dropout value of **0.1**.

Plots of the model loss and accuracy after applying dropout are shown in Table 11.

Plot Type	Plots
Epoch vs Accuracy	

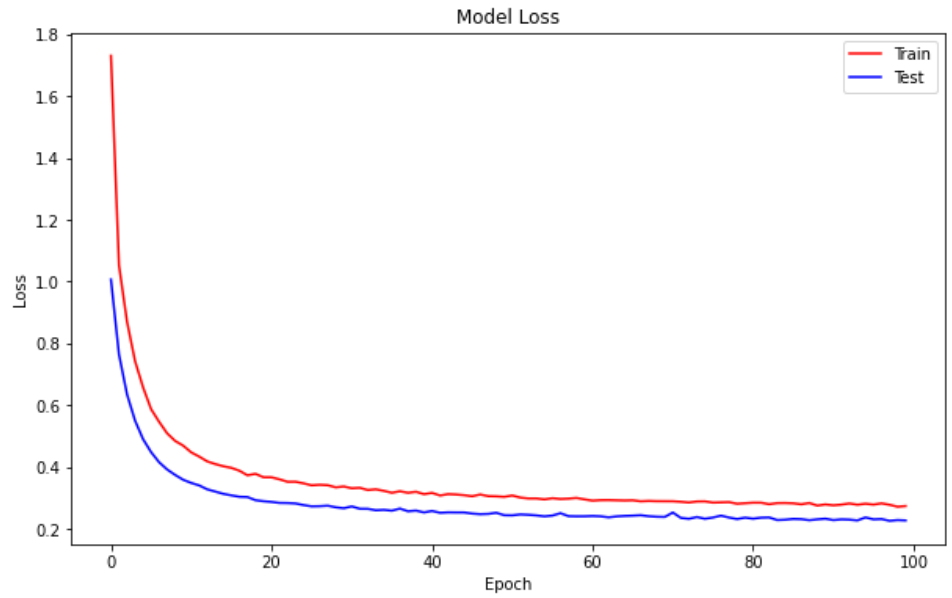
Plot Type	Plots
Epoch vs Loss	

Table 11 – Plots of model accuracy and loss after applying Dropout

Dropout will not be used because the epoch vs loss generalization gap is substantially wider than the other parameters we used for tuning the model. The 'optimal fit' for this parameter is not as good as the others.

6 Hyperparameter Tuning: Initialization Weights

The previous experimental runs used '**glorot_uniform**' as the initializer for the kernel weights matrix. This is the default initializer. In this experiment, I tried 2 other different weight initializers: **'he_uniform'**, **'random_normal'** (See Table 12).

Parameters	Values
kernel_initializer	glorot_uniform he_uniform random_normal

Table 12 – Kernel Initializer Values

glorot_uniform was the recommended kernel initializer.

The other parameters I used are shown in Table 9.

Plots of the model loss and accuracy after applying the '**glorot_uniform**' kernel initializer is shown in Table 13. Again, the learning curves shown in Table 13 display a near optimal fit, with little overfitting or underfitting.

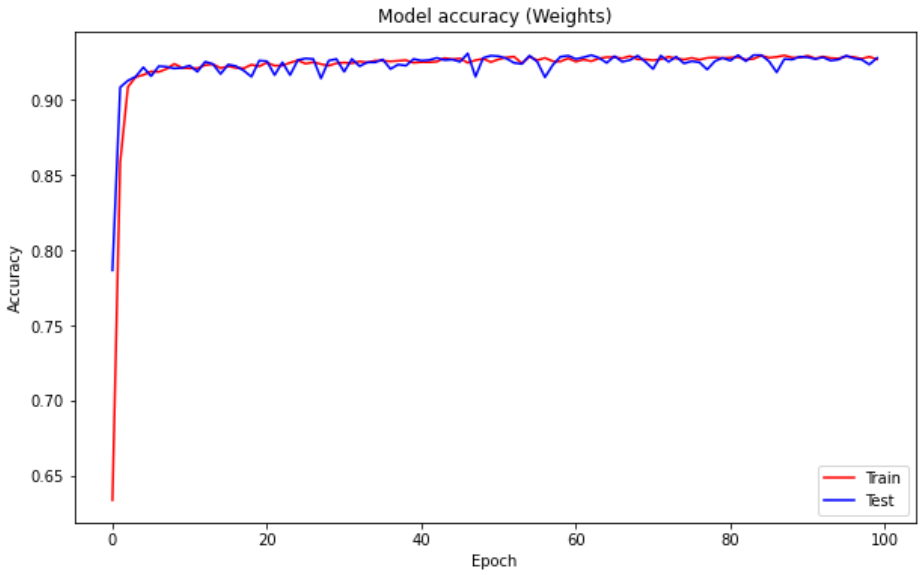
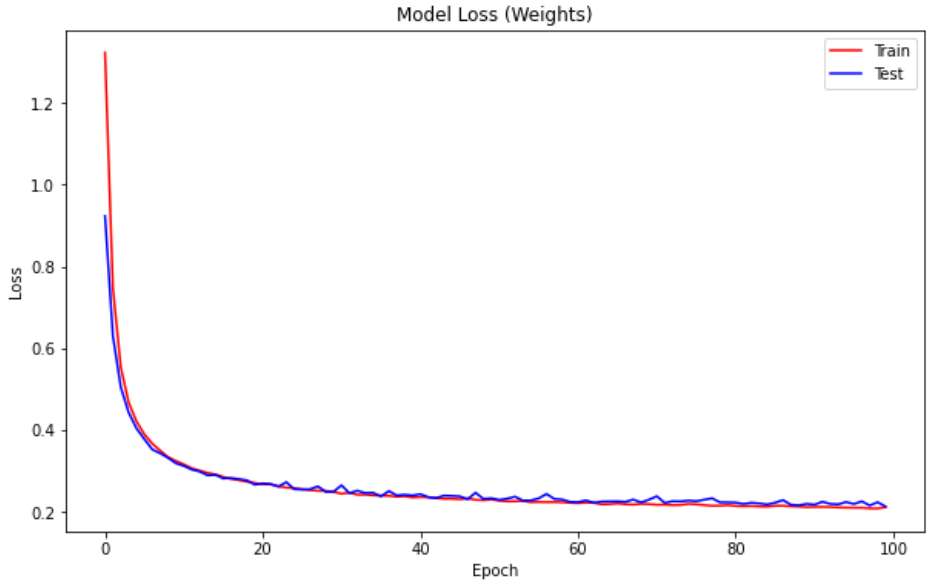
Plot Type	Plots
Epoch vs Accuracy	
Epoch vs Loss	

Table 13 - Plots of model accuracy and loss after applying kernel initialization

7 Hyperparameter Tuning: Optimizers

The next experiment that was carried out was varying the optimizers used (Table 14)

Parameters	Values
optimizers	adam adadelta rmsprop

Table 14 – Different Optimizers

'**adam**' was the optimizer that was suggested.

Plots of the model loss and accuracy after applying the 'adam' optimizer are shown in Table 15. Again, the learning curves shown here display a near optimal fit, with little overfitting or underfitting.

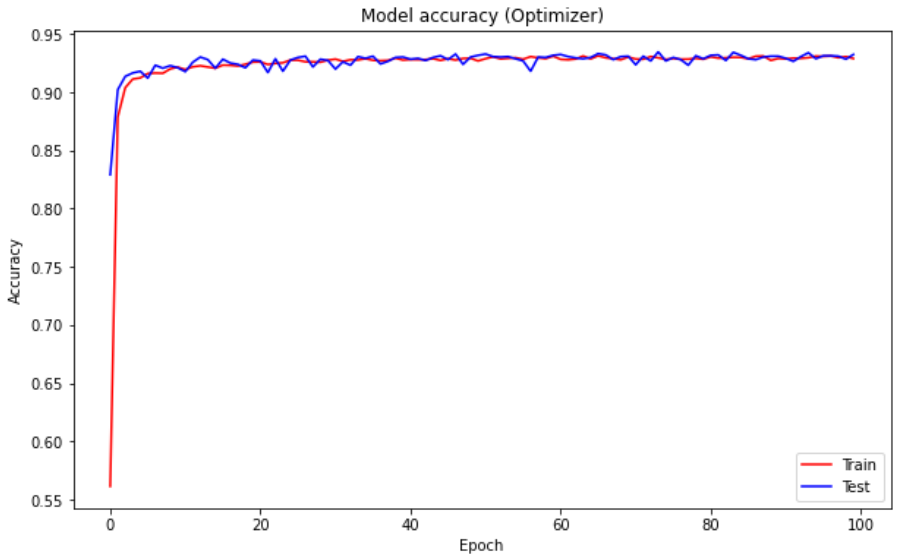
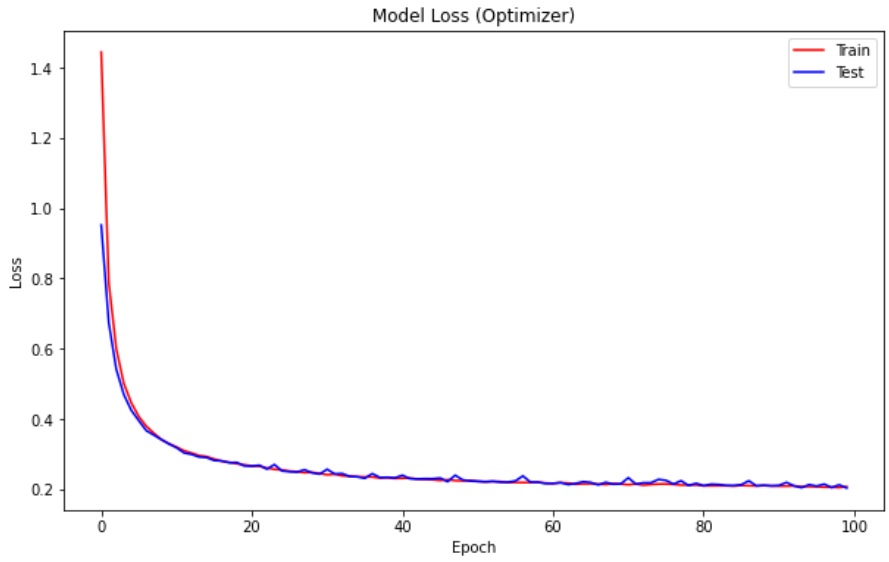
Plot Type	Plots
Epoch vs Accuracy	
Epoch vs Loss	

Table 15 - Plots of model accuracy and loss after applying Optimizers

8 Early Stopping

We use early stopping to determine the number of training epochs to use. It stops training when a monitored metric has stopped improving.

Parameters	Description	Value
monitor	Quantity to be monitored. If the goal of training is to minimize the loss, the metric to be monitored would be 'loss', and mode would be 'min'	loss
patience	Number of epochs with no improvement after which training will be stopped.	100
mode	In min mode, training will stop when the quantity monitored has stopped decreasing.	min

Table 16 – Early Stopping Parameters

The number of epochs used for the early stopping training is **2000**. Training stopped at the **715**th epoch.

Plots of the model loss and accuracy after applying the early stopping are shown in Table 17. Again, the learning curves shown here display a near optimal fit, with little overfitting or underfitting.

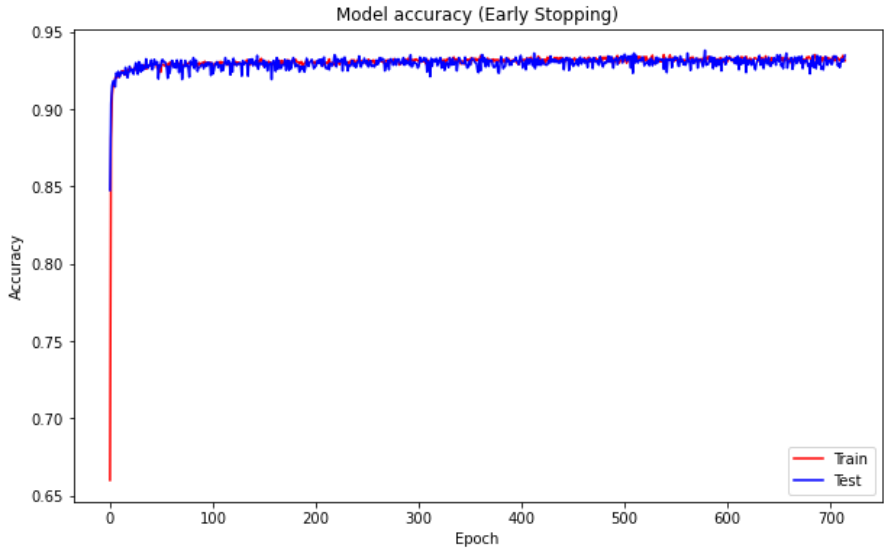
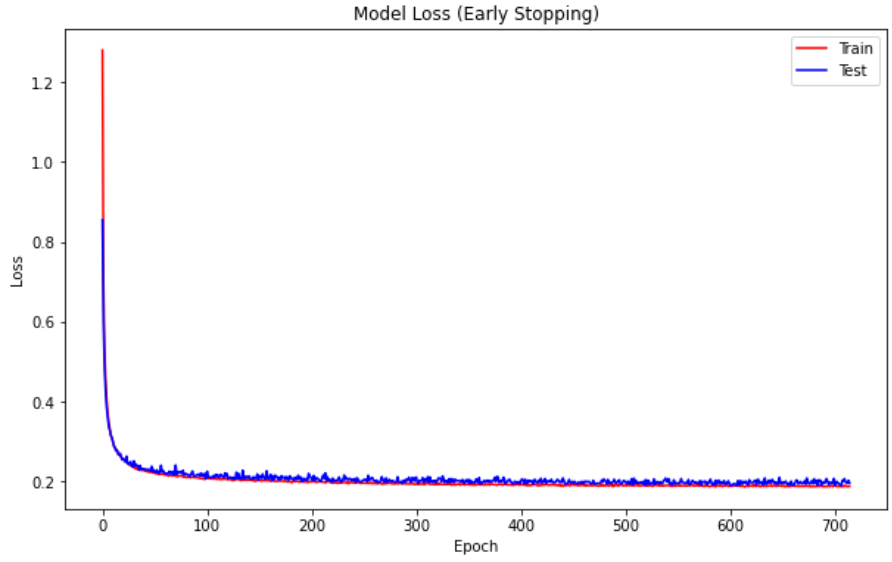
Plot Type	Plots
Epoch vs Accuracy	
Epoch vs Loss	

Table 17 - Plots of model accuracy and loss after applying Early Stopping

9 Results

In the learning curves (epoch vs. loss) that we have plotted, all show a near optimal fit with the exception when Dropout was used.

Since using Dropout resulted in a much wider generalization gap, we will not use dropout in our model.

After applying hyper-parameter tuning, we arrive at the final set of parameters for the dry bean classifier model (Table 18):

Parameters	Values
Activation function	SELU
Batch Size	64

Parameters	Values
Layers	3
Nodes (Hidden Layers)	14, 10, 9
Regularization Parameters	L2 0.01
Kernel Initializer (Weights)	glorot uniform
Optimizer	Adam

Table 18 – Final Set of Hyperparameters

We ran the evaluation method on the training data and the following results were produced:

Evaluation Metrics	Values
Loss	0.815
Accuracy	0.934

Table 19 – Model Evaluation Results

We also ran model prediction on the test set and computed the model accuracy and the accuracy was determined to be: **93.47%**

10 Comparison between deep learning and classical machine algorithms

This section discusses the differences between deep learning and classical machine learning.

Deep Learning requires huge datasets for training. In addition, it also needs high end GPUs in order for training to happen within a reasonable amount of time. High end GPUs are very costly without which it becomes impractical to train deep networks within a reasonable amount of time. In addition to fast GPU, a fast CPU processor, SSD, storage and a fast and large RAM are also required. In contrast, classical ML algorithms can be trained using a decent CPU, without the need for really high-end hardware. Classical ML algorithms can also be trained using a smaller amount of data.

Classical machine learning algorithms are also easier to understand and interpret unlike deep learning networks which is like a 'black box' and is difficult to understand what is really going on inside the hidden layers of deep networks. Hyper-parameter tuning can be done more easily on classical machine learning algorithms compared to deep learning networks. Network design encompassing for example, the number of hidden layers to use and the number nodes in each hidden layer is also a challenge for deep learning. It is more an art than science.

In spite of the disadvantages, deep learning networks have achieved accuracies that far surpass classical machine learning algorithms especially in areas such as computer vision, speech, natural language. Deep networks scale much better with more data than classical machine learning algorithms, i.e., the accuracy of deep learning models can be improved by providing more data. However, this is not true for classical machine learning. There is also no need for feature engineering when using deep learning, unlike in classical machine learning. Deep learning techniques are adaptable and transferrable. An example of this is transfer learning. Here, pre-trained networks can be used for different applications within the same domain. This is not possible in classical machine learning.

The pros and cons of deep learning vs classical machine is summarized below in Table 20:

Pros	Cons
No need for feature engineering. It is able to discover and learn features from raw data; this means there will be less reliance on subject matter experts.	Deep learning requires large datasets for training. Small datasets may not work well for deep learning.
Accuracy of deep learning improves by providing more data	With more data, deep learning will become computationally expensive. In addition to needing high end GPUs, huge and fast storage requirements, makes training deep learning models more costly.
Deep learning techniques are transferrable and adaptable unlike classical machine learning	Interpreting the models generated from deep learning is very difficult.
It is useful for solving complex problems such as computer vision, image classification, NLP and speech recognition.	

Table 20 – Deep Learning vs. Classical Machine Learning

11 Conclusion

To summarize, we have built a fully connected feed forward neural network to classify dry beans. There are 7 species of dry beans to be identified. This is a multi-class classification problem. First, we carried out exploratory data analysis and performed data pre-processing. Then, we built the neural network by first determining the number of hidden layers, the number of neurons in each hidden layer, batch size and activation function to use. We use grid searching to determine the best set of parameters for this step of the tuning. For the next few steps, we determine the regularization, dropout, initialization weights and optimizers to use. Early stopping was also applied as the initial number of epochs used was 100.This was increased to 2000. If we let the training run the full number of epochs, this can lead to overfitting of the training dataset. Hence, we stop training (at the 715th epoch) once the model performance stops improving on a hold-out validation dataset. The final model accuracy was achieved at **93.47%.**

12 Reference

[1]. https://archive.ics.uci.edu/ml/machine-learning-databases/00602/DryBeanDataset.zip

[2]. Introduction to Neural Networks for Java (second edition) by Jeff Heaton

[3]. https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw

[4]. http://alexlenail.me/NN-SVG/index.html

[5]. https://github.com/ashishpatel26/Tools-to-Design-or-Visualize-Architecture-of-Neural-Network

[6]. https://rstudio-conf-2020.github.io/dl-keras-tf/notebooks/learning-curve-diagnostics.nb.html