Assignment on Neural Networks № 3

Christos Chrysikos THMMY AEM 9432, Aristotle University of Thessaloniki 4/1/2023

At this project we chose to implement a Radial Basis Function Neural Network. We chose Iris Data-set as a train and test Data-set which consists of **150** different sample flowers, that contain information about 4 different characteristics:

- 1.sepal length
- 2.sepal width
- 3.petal length
- 4.petal width

And can be categorized in 3 different classes:

- 1.Iris Setosa
- 2.Iris Versicolour
- 3.Iris Virginica

Classification Algorithm using k-means and choosing random centroids

Listing 1: The code used for the implementaion of k-means with random selection of centroids.

```python
from keras.initializers import Initializer
from sklearn.cluster import KMeans


class InitCentersKMeans(Initializer):
    """ Initializer for initialization of centers of RBF network
        by clustering the given data set.
    # Arguments
        X: matrix, dataset
    """

    def __init__(self, X, max_iter=100):
        self.X = X
        self.max_iter = max_iter

    def __call__(self, shape, dtype=None):
        assert shape[1] == self.X.shape[1]

        n_centers = shape[0]
        km = KMeans(n_clusters=n_centers, max_iter=self.max_iter, verbose↩
            =0)
        km.fit(self.X)
        return km.cluster_centers_



class InitCentersRandom(Initializer):
    """ Initializer for initialization of centers of RBF network
        as random samples from the given data set.
    # Arguments
        X: matrix, dataset to choose the centers from (random rows
          are taken as centers)
    """

    def __init__(self, X):
        self.X = X

    def __call__(self, shape, dtype=None):
      assert shape[1] == self.X.shape[1]
      idx = np.random.randint(self.X.shape[0], size=shape[0])

      # type checking to access elements of data correctly
      if type(self.X) == np.ndarray:
          return self.X[idx, :]
      elif type(self.X) == pd.core.frame.DataFrame:
          return self.X.iloc[idx, :]
```

Radial Basis Function Algorithm

Listing 2: The code used for the implementaion of Radial Basis Function.

```python
from keras import backend as K
from keras.layers import Layer
from keras.initializers import RandomUniform, Initializer, Constant
import numpy as np



class RBFLayer(Layer):
    """ Layer of Gaussian RBF units.
    # Example
    ‘‘‘python
        model = Sequential()
        model.add(RBFLayer(10,
                           initializer=InitCentersRandom(X),
                           betas=1.0,
                           input_shape=(1,)))
        model.add(Dense(1))
    ‘‘‘
    # Arguments
        output_dim: number of hidden units (i.e. number of outputs of the
                    layer)
        initializer: instance of initiliazer to initialize centers
        betas: float, initial value for betas
    """

    def __init__(self, output_dim, initializer=None, betas=1.0, **kwargs):
        self.output_dim = output_dim
        self.init_betas = betas
        if not initializer:
            self.initializer = RandomUniform(0.0, 1.0)
        else:
            self.initializer = initializer
        super(RBFLayer, self).__init__(**kwargs)

    def build(self, input_shape):

        self.centers = self.add_weight(name='centers',
                                       shape=(self.output_dim, input_shape↩
                                           [1]),
                                       initializer=self.initializer,
                                       trainable=True)
        self.betas = self.add_weight(name='betas',
                                     shape=(self.output_dim,),
                                     initializer=Constant(
                                         value=self.init_betas),
                                     # initializer='ones',
                                     trainable=True)

        super(RBFLayer, self).build(input_shape)

    def call(self, x):

        C = K.expand_dims(self.centers)
        H = K.transpose(C-K.transpose(x))
        return K.exp(-self.betas * K.sum(H**2, axis=1))

        # C = self.centers[np.newaxis, :, :]
        # X = x[:, np.newaxis, :]

        # diffnorm = K.sum((C-X)**2, axis=-1)
        # ret = K.exp( - self.betas * diffnorm)
        # return ret

    def compute_output_shape(self, input_shape):
        return (input_shape[0], self.output_dim)

    def get_config(self):
        # have to define get_config to be able to use model_from_json
        config = {
            'output_dim': self.output_dim
        }
        base_config = super(RBFLayer, self).get_config()
        return dict(list(base_config.items()) + list(config.items()))
```

beta = $1/2 * \sigma_1^2$

Where σ_1 the deviation

Basic Training Algorithm

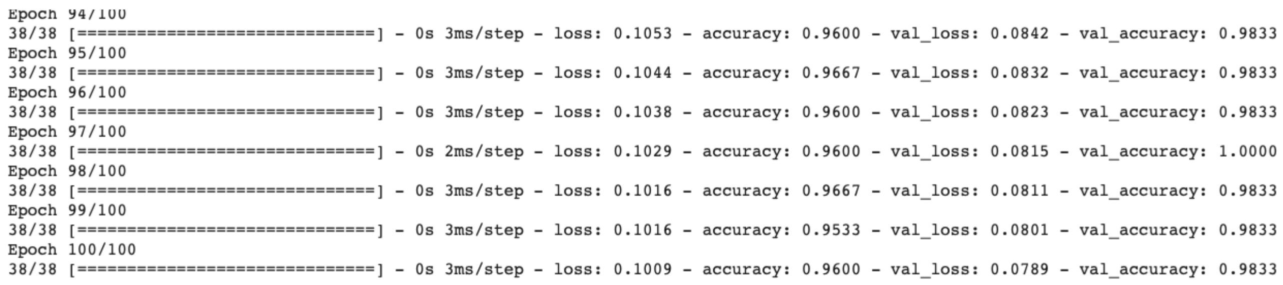
Listing 3: The code used for the implementaion of the basic Training Algorithm.

```python

import sys
sys.path.append('/gdrive/MyDrive')

import tensorflow as tf
from tensorflow import keras
import numpy as np
from keras.models import Sequential
from sklearn.cluster import KMeans
from matplotlib import pyplot as plt
import math
import pandas as pd
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
from rbf import RBFLayer
from keras.layers import Dense
import kmeans
from sklearn.model_selection import train_test_split


classesNames = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
le = LabelEncoder()
scaler = MinMaxScaler()

#Reshaping the Data
dt = pd.read_csv('/gdrive/MyDrive/Iris.csv')
df_iris = dt.iloc[:, 1:].copy()
le = LabelEncoder()
scaled_df_iris = le.fit_transform(df_iris[['Species']])
df_iris[['Species']] = scaled_df_iris.reshape(scaled_df_iris.shape[0], -1)

X = df_iris.iloc[:, :-1]
X['SepalLengthCm'] = scaler.fit_transform(X[['SepalLengthCm']])
X['SepalWidthCm'] = scaler.fit_transform(X[['SepalWidthCm']])
X['PetalLengthCm'] = scaler.fit_transform(X[['PetalLengthCm']])
X['PetalWidthCm'] = scaler.fit_transform(X[['PetalWidthCm']])

y = df_iris.iloc[:,-1]

X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.4)



```

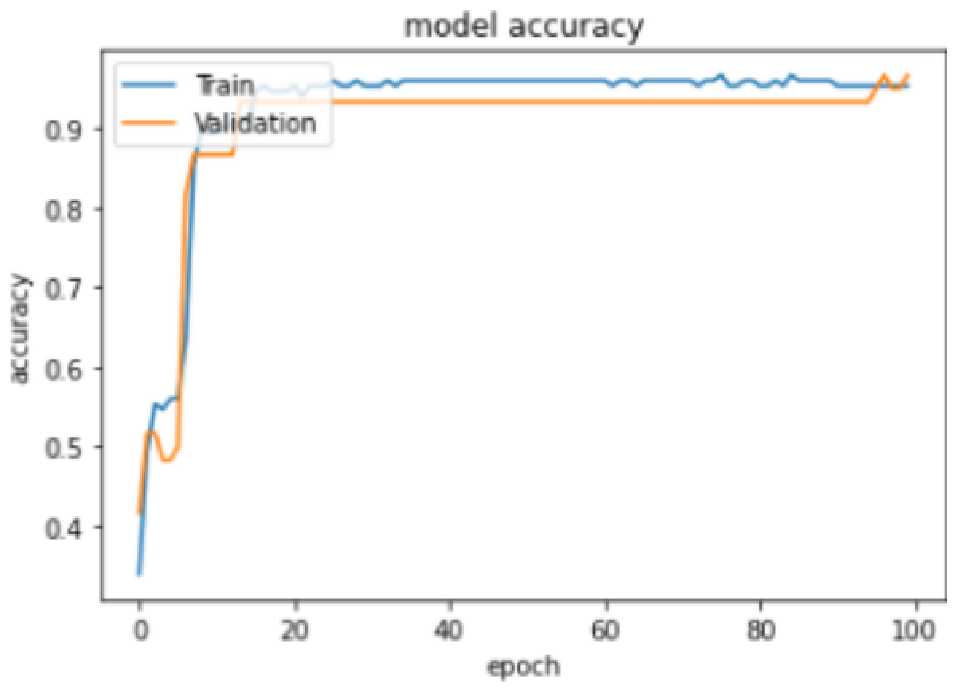
```python


rbf_layer = RBFLayer(10, initializer = kmeans.InitCentersKMeans(X_train), ↩
    betas=2.0, input_shape=([4]))

model = Sequential()
model.add(rbf_layer)
model.add(Dense(3, activation = 'softmax'))

model.compile(loss='sparse_categorical_crossentropy', optimizer='rmsprop',↩
    metrics=['accuracy'])



history = model.fit(X, y, epochs=100, batch_size=4, validation_data=(↩
    X_test, y_test))

y_pred = model.predict(X_test)


# #Some predictions

y_test = y_test.to_list()

y_pred_no_prob = []
testy = []

for i in range (0,20):
  #take indice of max value in y_pred
  r = np.argmax(y_pred[i], axis=0)
  #take suitable name from classesNames and pass it to y_pred_no_prob list
  a = classesNames[r]
  y_pred_no_prob.append(a)
  b = y_test[i]
  c = classesNames[b]
  testy.append(c)



for i in range (1,20):
  if y_pred_no_prob[i]!=testy[i]:
    print('Wrong prediction. Expected specie: '+str(testy[i])+' and got ↩
        specie: '+str(y_pred_no_prob[i]))
  elif y_pred_no_prob[i]==testy[i]:
    print('Right prediction. Expected specie: '+str(testy[i])+' and got ↩
        specie: '+str(y_pred_no_prob[i]))

```

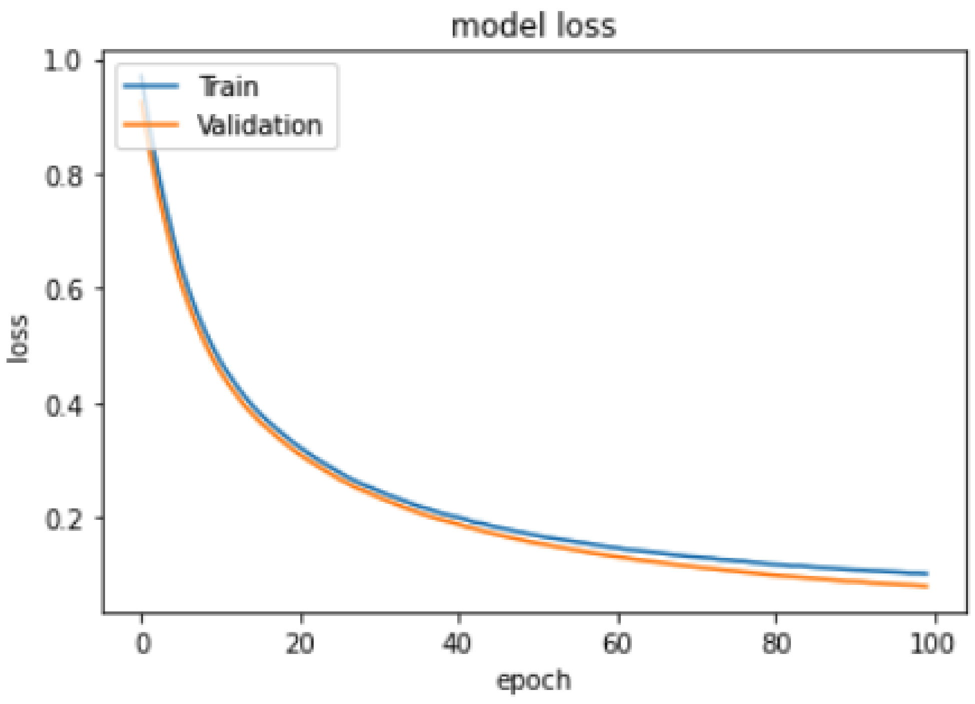
```python

# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

We started with pre-processing the data by normalising the input data with MaxScaler to [0,1] and the the output data to [0,2].
Next we designed the basic architecture of our Radial Basis Function Neural Network (RBFNN), as well as some accurate and inaccurate classification examples, accuracy diagrams and cost/loss diagrams.
The results from training our neural network are given below, mostly examples from the last epochs for convenience.

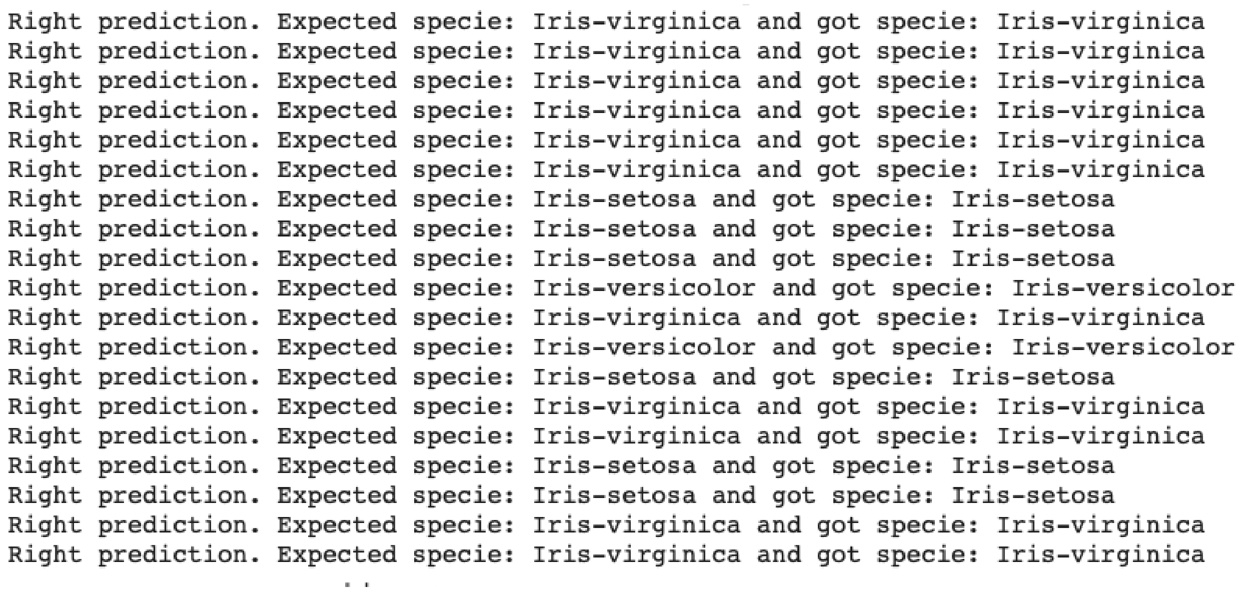


```
Epoch 94/100
38/38 [==============================] - 0s 3ms/step - loss: 0.1053 - accuracy: 0.9600 - val_loss: 0.0842 - val_accuracy: 0.9833
Epoch 95/100
38/38 [==============================] - 0s 3ms/step - loss: 0.1044 - accuracy: 0.9667 - val_loss: 0.0832 - val_accuracy: 0.9833
Epoch 96/100
38/38 [==============================] - 0s 3ms/step - loss: 0.1038 - accuracy: 0.9600 - val_loss: 0.0823 - val_accuracy: 0.9833
Epoch 97/100
38/38 [==============================] - 0s 2ms/step - loss: 0.1029 - accuracy: 0.9600 - val_loss: 0.0815 - val_accuracy: 1.0000
Epoch 98/100
38/38 [==============================] - 0s 3ms/step - loss: 0.1016 - accuracy: 0.9667 - val_loss: 0.0811 - val_accuracy: 0.9833
Epoch 99/100
38/38 [==============================] - 0s 3ms/step - loss: 0.1016 - accuracy: 0.9533 - val_loss: 0.0801 - val_accuracy: 0.9833
Epoch 100/100
38/38 [==============================] - 0s 3ms/step - loss: 0.1009 - accuracy: 0.9600 - val_loss: 0.0789 - val_accuracy: 0.9833
```





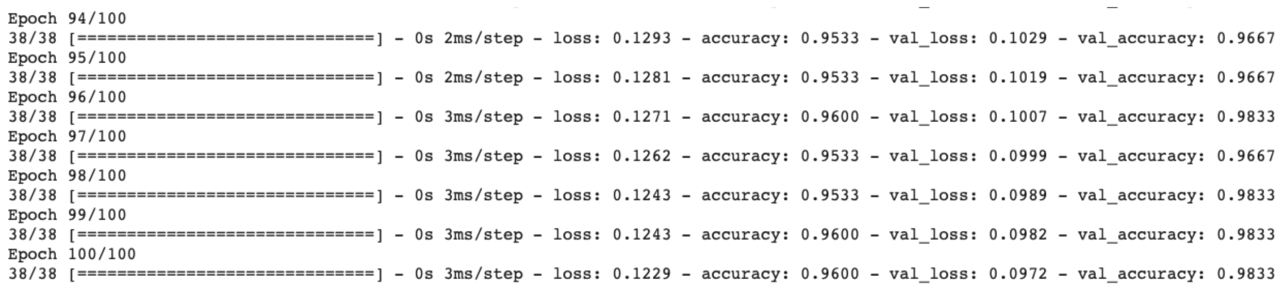
The results that we get are nearly optimal because after the end of the 100 epochs the models seems to be more than 95% accurate both in training and in test data. It is also conveniently fast accomplishing this level of accuracy at only the first 20 epochs while the loss function stays above 0.1. The completion time of the algorithm is calculated to be only at 13 seconds.
Below are presented some examples of accurate and inaccurate classification



```
Right prediction. Expected specie: Iris-virginica and got specie: Iris-virginica
Right prediction. Expected specie: Iris-virginica and got specie: Iris-virginica
Right prediction. Expected specie: Iris-virginica and got specie: Iris-virginica
Right prediction. Expected specie: Iris-virginica and got specie: Iris-virginica
Right prediction. Expected specie: Iris-virginica and got specie: Iris-virginica
Right prediction. Expected specie: Iris-virginica and got specie: Iris-virginica
Right prediction. Expected specie: Iris-setosa and got specie: Iris-setosa
Right prediction. Expected specie: Iris-setosa and got specie: Iris-setosa
Right prediction. Expected specie: Iris-setosa and got specie: Iris-setosa
Right prediction. Expected specie: Iris-versicolor and got specie: Iris-versicolor
Right prediction. Expected specie: Iris-virginica and got specie: Iris-virginica
Right prediction. Expected specie: Iris-versicolor and got specie: Iris-versicolor
Right prediction. Expected specie: Iris-setosa and got specie: Iris-setosa
Right prediction. Expected specie: Iris-virginica and got specie: Iris-virginica
Right prediction. Expected specie: Iris-virginica and got specie: Iris-virginica
Right prediction. Expected specie: Iris-setosa and got specie: Iris-setosa
Right prediction. Expected specie: Iris-setosa and got specie: Iris-setosa
Right prediction. Expected specie: Iris-virginica and got specie: Iris-virginica
Right prediction. Expected specie: Iris-virginica and got specie: Iris-virginica
```

Lowering Beta equal to 1

Below are presented the result by lowering the beta value to 1.

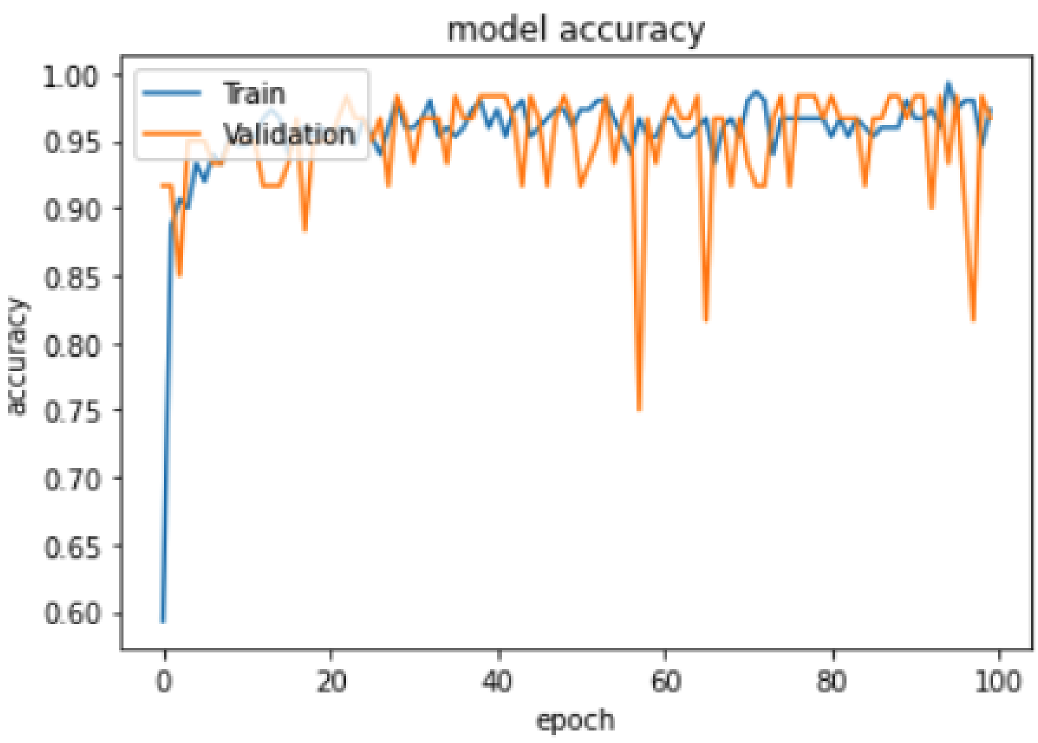


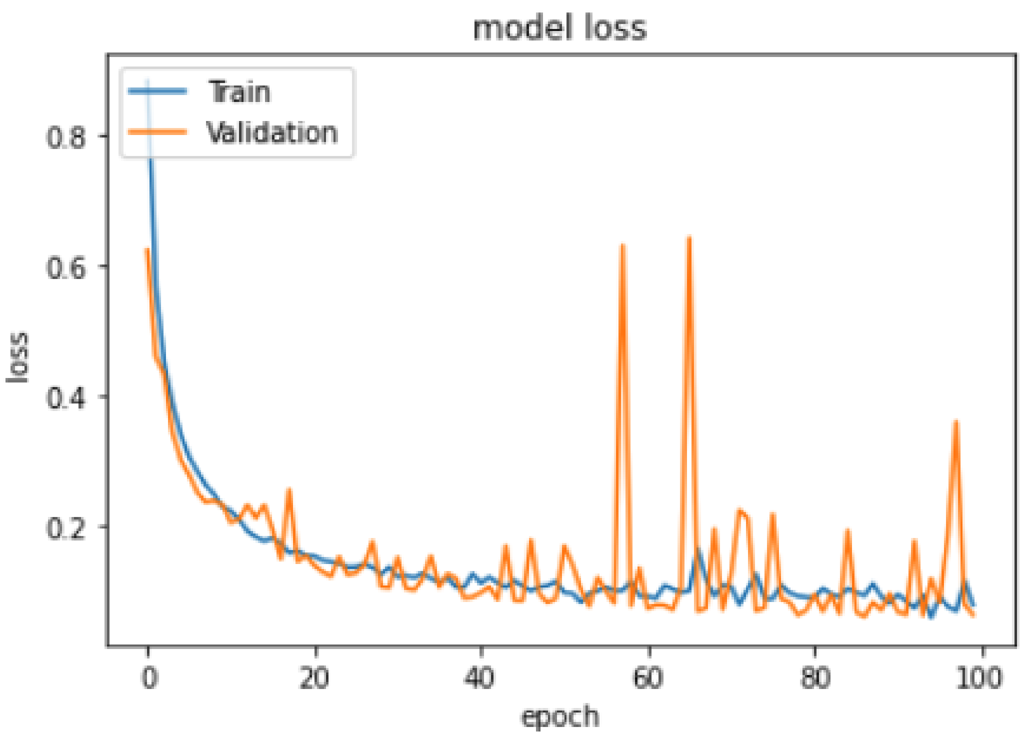
```
Epoch 94/100
38/38 [==============================] - 0s 2ms/step - loss: 0.1293 - accuracy: 0.9533 - val_loss: 0.1029 - val_accuracy: 0.9667
Epoch 95/100
38/38 [==============================] - 0s 2ms/step - loss: 0.1281 - accuracy: 0.9533 - val_loss: 0.1019 - val_accuracy: 0.9667
Epoch 96/100
38/38 [==============================] - 0s 3ms/step - loss: 0.1271 - accuracy: 0.9600 - val_loss: 0.1007 - val_accuracy: 0.9833
Epoch 97/100
38/38 [==============================] - 0s 3ms/step - loss: 0.1262 - accuracy: 0.9533 - val_loss: 0.0999 - val_accuracy: 0.9667
Epoch 98/100
38/38 [==============================] - 0s 3ms/step - loss: 0.1243 - accuracy: 0.9533 - val_loss: 0.0989 - val_accuracy: 0.9833
Epoch 99/100
38/38 [==============================] - 0s 3ms/step - loss: 0.1243 - accuracy: 0.9600 - val_loss: 0.0982 - val_accuracy: 0.9833
Epoch 100/100
38/38 [==============================] - 0s 3ms/step - loss: 0.1229 - accuracy: 0.9600 - val_loss: 0.0972 - val_accuracy: 0.9833
```

There is no substantial difference while lowering the beta value to 1 and the same applies for increasing beta above 2.
Similar results occur when we change the optimiser from adam to sigmoid(sgd) and learning_rate = 0.01 (default rate).

Increasing learning rate to 0.1

There are some interesting results though when we change the learning_rate value to 0.1 using the sgd optimiser presented bellow.

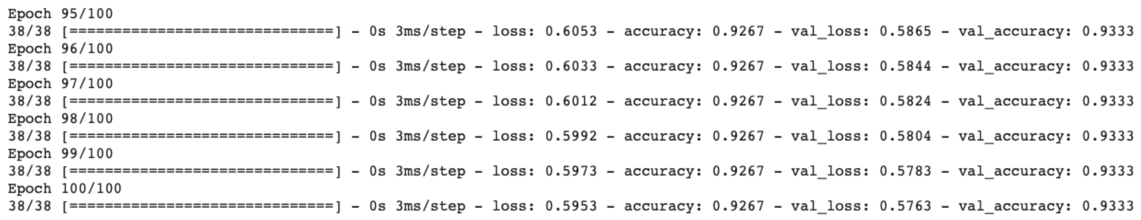




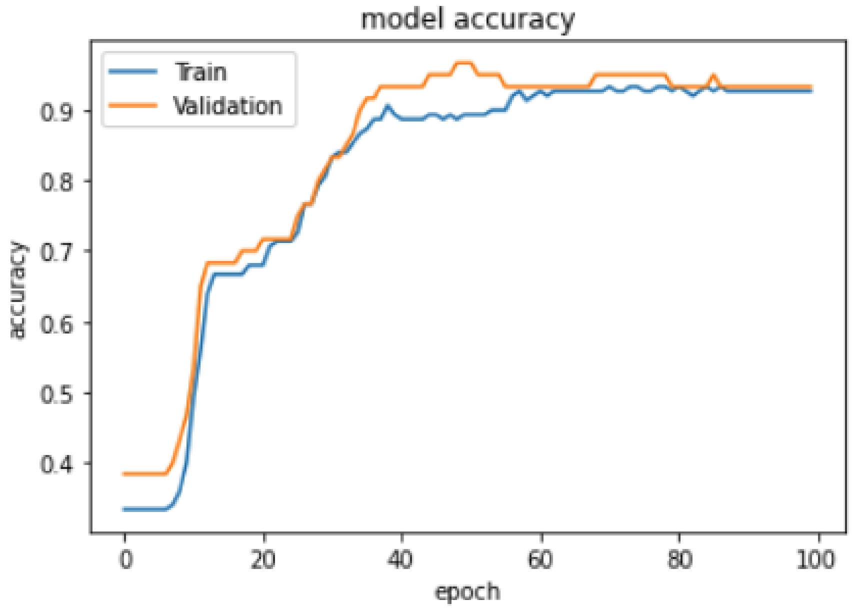
The abnormal spikes in the diagrams indicate that the learning rate we chose is considered large causing instability during the accuracy and loss change and therefore is not an optimal choice.

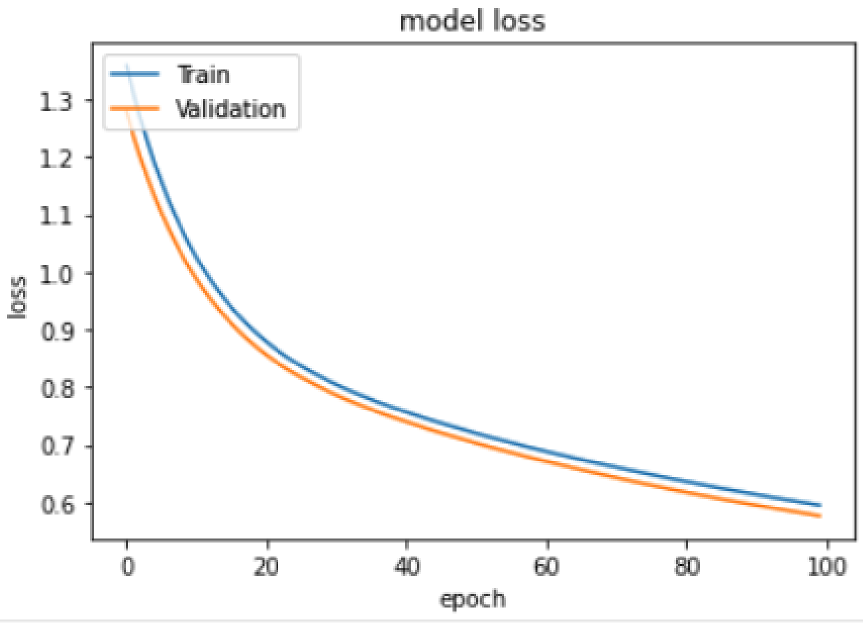
Decreasing learning rate to 0.001

We can see that by decreasing the learning_rate to 0.001 the final loss value increases and the accuracy decreases a little, which makes the 0.1 the best choice.
The completion time stayed the same throughout the different experiments.



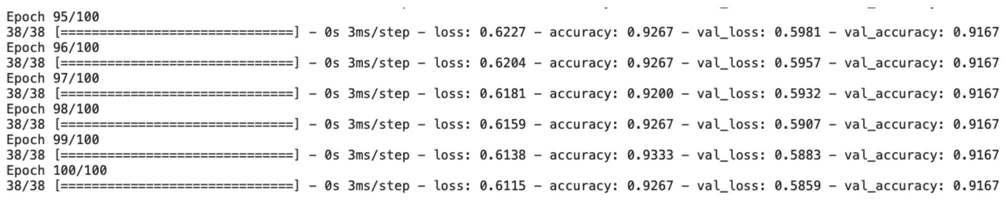
```
Epoch 95/100
38/38 [==============================] - 0s 3ms/step - loss: 0.6053 - accuracy: 0.9267 - val_loss: 0.5865 - val_accuracy: 0.9333
Epoch 96/100
38/38 [==============================] - 0s 3ms/step - loss: 0.6033 - accuracy: 0.9267 - val_loss: 0.5844 - val_accuracy: 0.9333
Epoch 97/100
38/38 [==============================] - 0s 3ms/step - loss: 0.6012 - accuracy: 0.9267 - val_loss: 0.5824 - val_accuracy: 0.9333
Epoch 98/100
38/38 [==============================] - 0s 3ms/step - loss: 0.5992 - accuracy: 0.9267 - val_loss: 0.5804 - val_accuracy: 0.9333
Epoch 99/100
38/38 [==============================] - 0s 3ms/step - loss: 0.5973 - accuracy: 0.9267 - val_loss: 0.5783 - val_accuracy: 0.9333
Epoch 100/100
38/38 [==============================] - 0s 3ms/step - loss: 0.5953 - accuracy: 0.9267 - val_loss: 0.5763 - val_accuracy: 0.9333
```



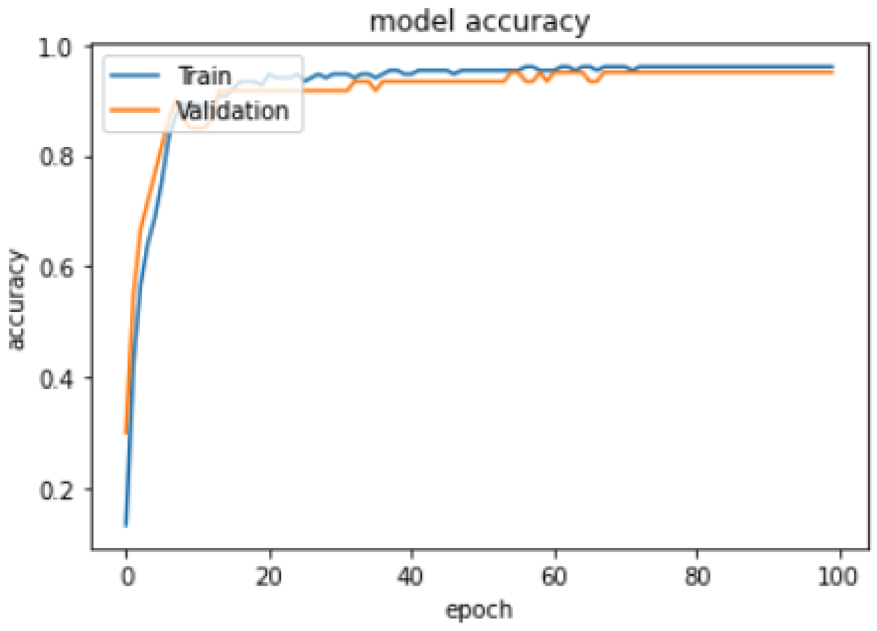


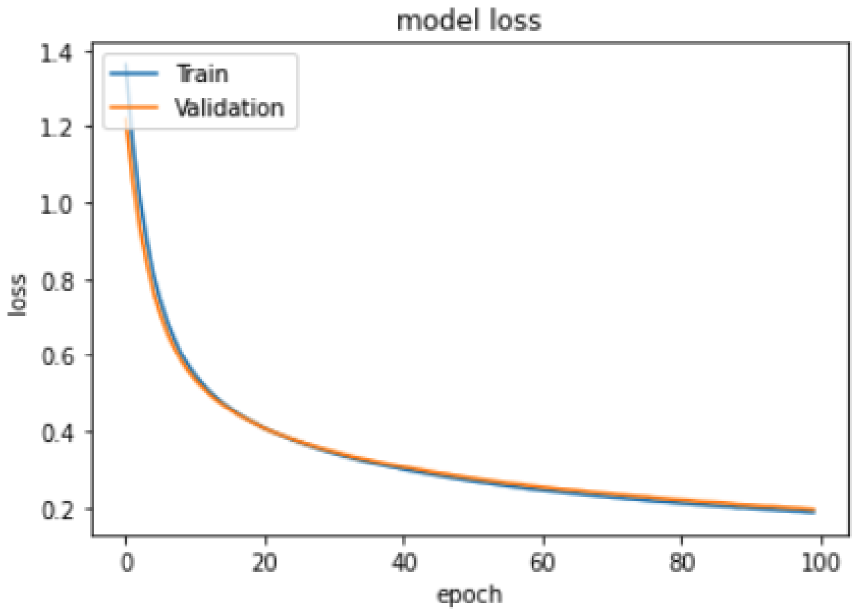
Random Selection of k-means centroids

Next we change the centroid selection method to random and bellow are presented the results There is no obvious change except the slightly abnormal change of accuracy during training.



```
Epoch 95/100
38/38 [==============================] - 0s 3ms/step - loss: 0.6227 - accuracy: 0.9267 - val_loss: 0.5981 - val_accuracy: 0.9167
Epoch 96/100
38/38 [==============================] - 0s 3ms/step - loss: 0.6204 - accuracy: 0.9267 - val_loss: 0.5957 - val_accuracy: 0.9167
Epoch 97/100
38/38 [==============================] - 0s 3ms/step - loss: 0.6181 - accuracy: 0.9200 - val_loss: 0.5932 - val_accuracy: 0.9167
Epoch 98/100
38/38 [==============================] - 0s 3ms/step - loss: 0.6159 - accuracy: 0.9267 - val_loss: 0.5907 - val_accuracy: 0.9167
Epoch 99/100
38/38 [==============================] - 0s 3ms/step - loss: 0.6138 - accuracy: 0.9333 - val_loss: 0.5883 - val_accuracy: 0.9167
Epoch 100/100
38/38 [==============================] - 0s 3ms/step - loss: 0.6115 - accuracy: 0.9267 - val_loss: 0.5859 - val_accuracy: 0.9167
```



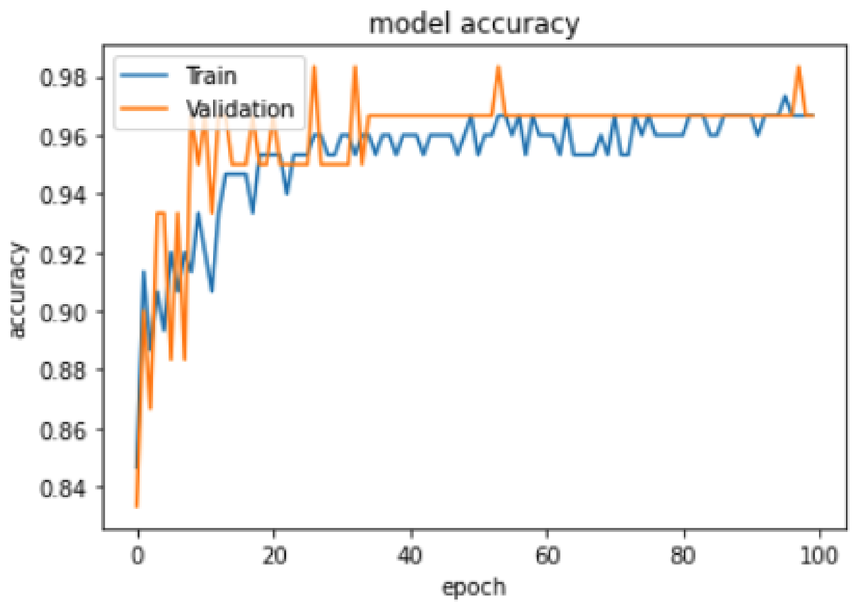


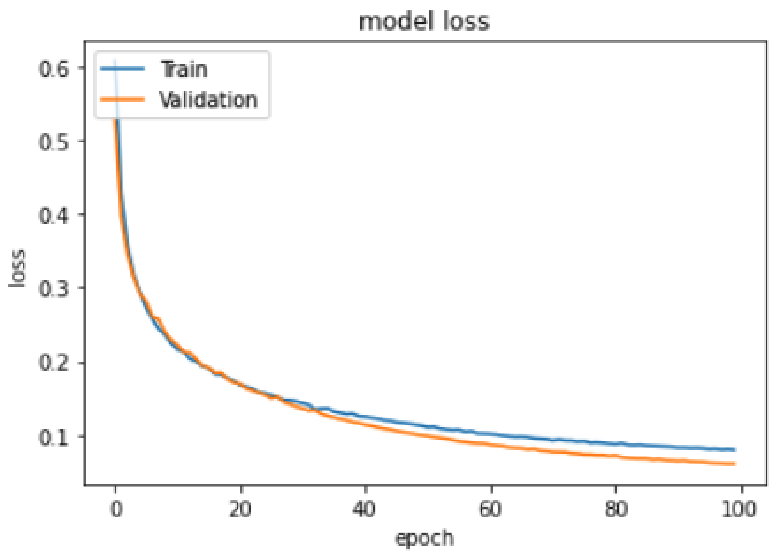
There is although a significant completion time difference at 13 seconds.

Random Selection of k-means centroids and Neurons Count = 100

We increase the neuron count at the RBF layer from to 10 to 100.

```
Epoch 95/100
38/38 [==============================] - 0s 3ms/step - loss: 0.0816 - accuracy: 0.9667 - val_loss: 0.0630 - val_accuracy: 0.9667
Epoch 96/100
38/38 [==============================] - 0s 3ms/step - loss: 0.0807 - accuracy: 0.9733 - val_loss: 0.0620 - val_accuracy: 0.9667
Epoch 97/100
38/38 [==============================] - 0s 3ms/step - loss: 0.0811 - accuracy: 0.9667 - val_loss: 0.0615 - val_accuracy: 0.9667
Epoch 98/100
38/38 [==============================] - 0s 3ms/step - loss: 0.0799 - accuracy: 0.9667 - val_loss: 0.0613 - val_accuracy: 0.9833
Epoch 99/100
38/38 [==============================] - 0s 3ms/step - loss: 0.0806 - accuracy: 0.9667 - val_loss: 0.0608 - val_accuracy: 0.9667
Epoch 100/100
38/38 [==============================] - 0s 3ms/step - loss: 0.0795 - accuracy: 0.9667 - val_loss: 0.0609 - val_accuracy: 0.9667
```





Similar values in the accuracy results as well as a small decrease in the loss. Completion time increase to 23 seconds.

Comparison

Comparing the results from an RBFNN we can see that the accuracy is almost the same as KNN when k=1 k=3(KNN accuracy = 0.9566) and slightly increased in comparison with the k nearest centroid (KNC accuracy = 0.941 in train set and accuracy = 0.9 in test set), which makes RBFNN one of the best choices for this type of data.