3η Εργασία Νευρωνικών Δικτύων

Περιεχόμενα

1. Εισαγωγή

Η εργασία που εκπονήθηκε, πραγματεύεται την υλοποίηση ενός **Radial Basis Function Neural Network** το οποίο εκπαιδεύεται πάνω στην βάση δεδομένων MNIST και αποσκοπεί στην εύρεση κλάσης παραδειγμάτων που θα το τροφοδοτούμε.

Ο κώδικας υλοποιήθηκε σε γλώσσα προγραμματισμού python σε περιβάλλον VScode και εμπεριέχει συναρτήσεις από το API **keras** τις βιβλιοθήκες **numpy**, **sklearn**, **matplotlib**, **sys**.

2.Κώδικας

```
from tensorflow import keras
```

```
from keras.datasets import mnist
from matplotlib import pyplot as plt
import numpy as np
from keras import backend as K
from keras.layers import Layer
from keras import models, layers
from keras.initializers import RandomUniform, Initializer, Constant
import numpy as np
from keras.initializers import Initializer
from sklearn.cluster import KMeans
import pandas as pd
from keras.datasets import cifar10

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])

        if type(self.X) == np.ndarray:
            return self.X[idx, :]
        elif type(self.X) == pd.core.frame.DataFrame:
            return self.X.iloc[idx, :]



    # type checking to access elements of data correctly




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()))



# mnist dataset
# Loading the Data
(x_train,y_train), (x_test,y_test) = mnist.load_data()
# Shaping the Data
x_train = x_train.reshape(x_train.shape[0], x_train.shape[1]**2)
x_test = x_test.reshape(x_test.shape[0], x_test.shape[1]**2)
```

```
x_train, x_test = np.array(x_train, np.float32), np.array(x_test, np.float32)
x_train, x_test = x_train/255., x_test/255.
num_inputs = 28*28
num_outputs = 10

rbflayer = RBFLayer(10,
                    initializer=InitCentersRandom(x_train),
                    betas=0.01,
                    input_shape=(x_test.shape[1],))

model = keras.Sequential()
model.add(rbflayer)
model.add(layers.Dense(10, activation="sigmoid"))


model.compile(optimizer='Adamax',
loss = keras.losses.SparseCategoricalCrossentropy(from_logits=True),
metrics=['accuracy'])

history = model.fit(x_train,y_train,
    validation_data = (x_test, y_test),
    epochs = 50)

plt.figure(1)
plt.plot(history.history['accuracy'], label = 'train')
plt.plot(history.history['val_accuracy'], label = 'test')
plt.legend()
plt.title('Performance on training and validation sets')
plt.show()




sample_idx = int(np.random.random() * len(y_test)) # Choose a random sample index

x_sample = x_test[sample_idx]
x_sample = np.expand_dims(x_sample, axis=0)#Changing he shape to vector
from(784,) to (1,784)
prediction = model.predict(x_sample) #Returns a vector of probabilities for each
class


predicted_number = np.argmax(prediction)#Get the next number by adding 1 to the
predicted number
next_number = (predicted_number + 1) % 10
```

```
print("Predicted",predicted_number)

x_sample = x_sample.reshape(28,28)
plt.imshow(x_sample, cmap='gray_r')
plt.show()
```

3. Ανάλυση Κώδικα

```
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()))
```

Είναι η υλοποίηση του Layer που θα μας βοηθήσει να το εντάξουμε στο νευρωνικό δίκτυο μας.

```
rbflayer = RBFLayer(10,
                    initializer=InitCentersRandom(x_train),
                    betas=0.01,
                    input_shape=(x_test.shape[1],))
```

Εδώ αρχικοποιούμε το layer μας σύμφωνα με το input shape του dataset στο οποίο θα το εκπαιδεύσουμε και το `betas` είναι η υπερπαράμετρος που καθορίζει το πλάτος της

γκαουσιανής συνάρτησης που χρησιμοποιείται στην έξοδο κάθε νευρώνα. Το initializer τον θέτουμε με την συνάρτηση `InitCentersRandom()` η οποία παίρνει τυχαία δείγματα για να αρχικοποιήσει τα βάρη.

```
# Shaping the Data
x_train = x_train.reshape(x_train.shape[0], x_train.shape[1]**2)
x_test = x_test.reshape(x_test.shape[0], x_test.shape[1]**2)

x_train, x_test = np.array(x_train, np.float32), np.array(x_test, np.float32)
x_train, x_test = x_train/255., x_test/255.
```

Σε αυτό το σημείο του κώδικα κατεβάζουμε το dataset **MNIST**, και εν συνεχεία αλλάζουμε τις διαστάσεις των πινάκων των samples έτσι ώστε να είναι πίνακες από vectors (που είναι τα samples) για να μπορούμε να τα προωθήσουμε στις επόμενες συναρτήσεις μας.

Είναι σημαντικό να κανονικοποιήσουμε τα δεδομένα μας στο πεδίο [0,1] έτσι ώστε να μην έχουμε λανθασμένους υπολογισμούς στο δίκτυο μας από τυχόντα outliers. Επειδή ξέρουμε ότι οι gray-scale εικόνες που έχουμε παίρνουν τιμές σε κάθε pixel από 0 έως 255 τότε διαιρούμε όλα τα pixel με των αριθμό 255 κάτι που βγαίνει από τον τύπο:

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}}$$

```
model = keras.Sequential()
model.add(rbflayer)
model.add(layers.Dense(10, activation="sigmoid"))
```

```
model.compile(optimizer='Adamax',
loss = keras.losses.SparseCategoricalCrossentropy(from_logits=True),
metrics=['accuracy'])

history = model.fit(x_train,y_train,
    validation_data = (x_test, y_test),
    epochs = 50)
```

Σε αυτό το σημείο του κώδικα καλούμε την συνάρτηση

```
model = keras.Sequential()
```

έπειτα βάζουμε ένα layer από το rbflayer που φτιάξαμε και ένα Dense για να μας δίνει την έξοδο.

έπειτα καλούμε την συνάρτηση compile για να δώσουμε στο μοντέλο μας τις μετρικές και τις μεθόδους με τις οποίες θα εκπαιδεύεται και μετά καλούμε την συνάρτηση fit του μοντέλου για να το εκπαιδεύσουμε πάνω στο MNIST dataset.

```
plt.figure(1)
plt.plot(history.history['accuracy'], label = 'train')
plt.plot(history.history['val_accuracy'], label = 'test')
plt.legend()
plt.title('Performance on training and validation sets')
plt.show()
```

Τέλος δημιουργούμε ένα διάγραμμά που μας δείχνει ανάλογα των εποχών που περνάνε πως αποδίδει το μοντέλο μας.

```
sample_idx = int(np.random.random() * len(y_test)) # Choose a random sample index

x_sample = x_test[sample_idx]
x_sample = np.expand_dims(x_sample, axis=0)#Changing he shape to vector
from(784,) to (1,784)
prediction = model.predict(x_sample) #Returns a vector of probabilities for each
class


predicted_number = np.argmax(prediction)#Get the next number by adding 1 to the
predicted number
next_number = (predicted_number + 1) % 10
print("Predicted",predicted_number)

x_sample = x_sample.reshape(28,28)
plt.imshow(x_sample, cmap='gray_r')
plt.show()
```

Εδώ βρίσκεται ένα προαιρετικό script το οποίο ζωγραφίζει ένα τυχαίο sample το οποίο κάνει predict και βλέπουμε αν ταυτίζονται.

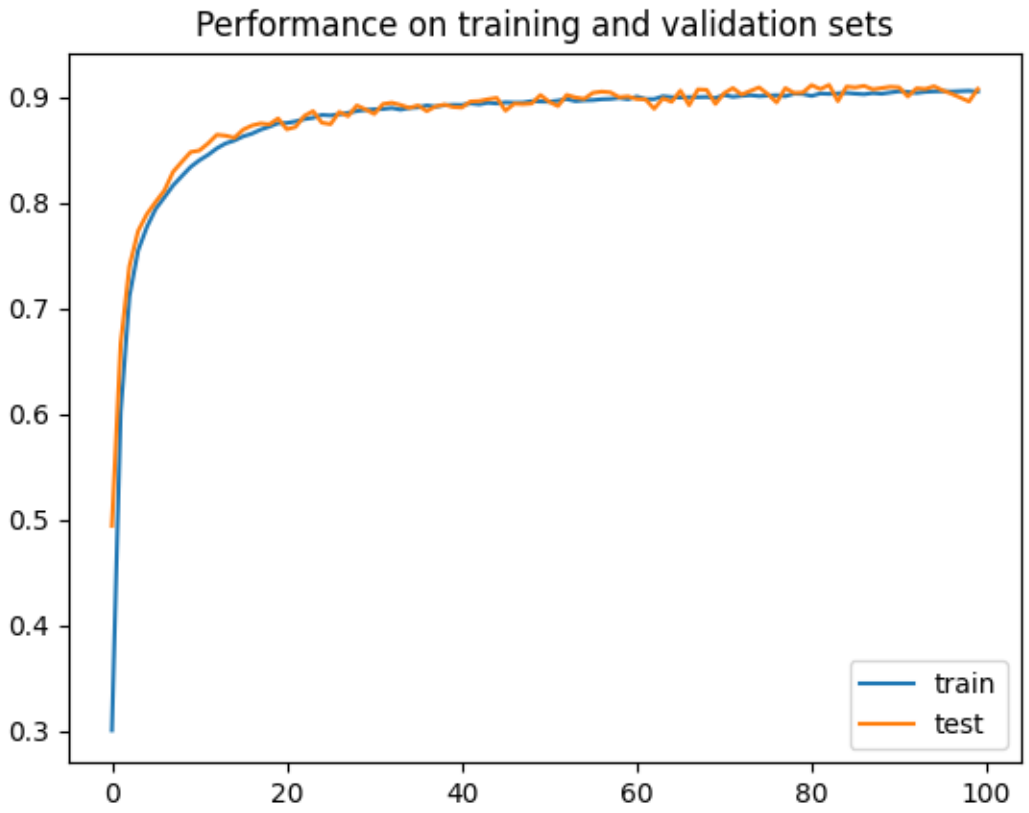
4. Δοκιμές

4.1 Initiliazer=InitCentersRandom, Neurons=10

4.1.1 Εποχές 50, betas=0.01

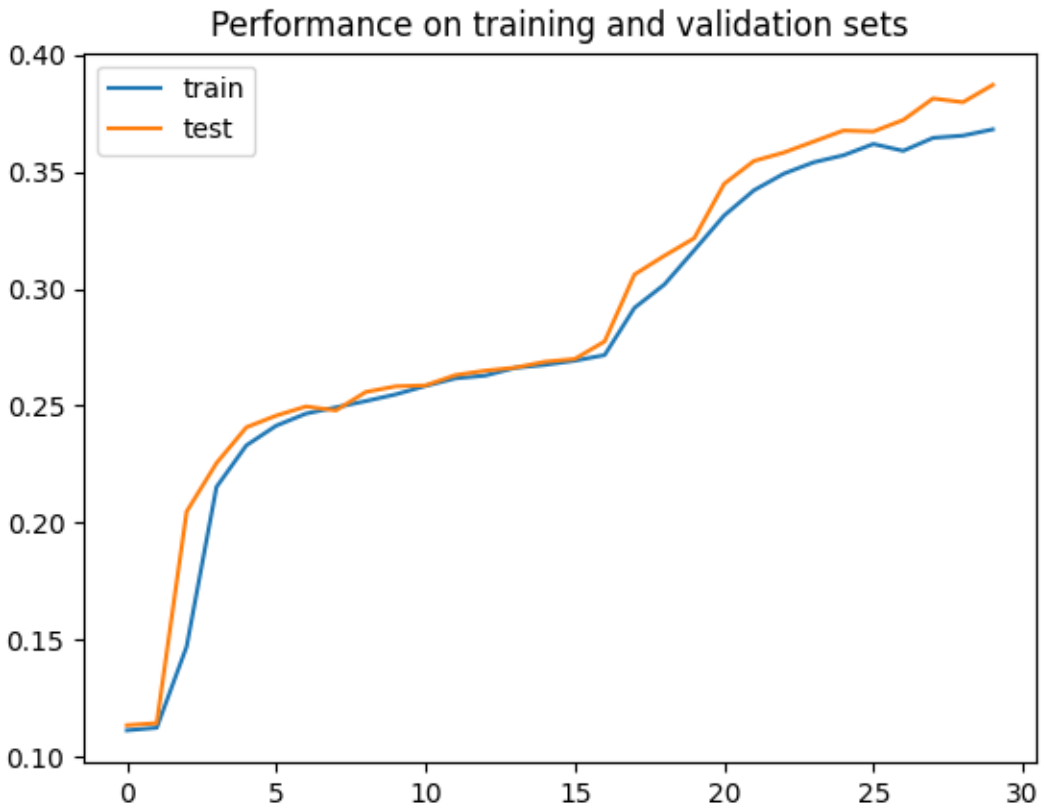
Χρόνος Εκτέλεσης = 304s

4.1.2 Εποχές 100, betas = 0.05



Χρόνος Εκτέλεσης =603 s

4.1.3 Εποχές 30, betas = 0.5



Χρόνος Εκτέλεσης = 183s

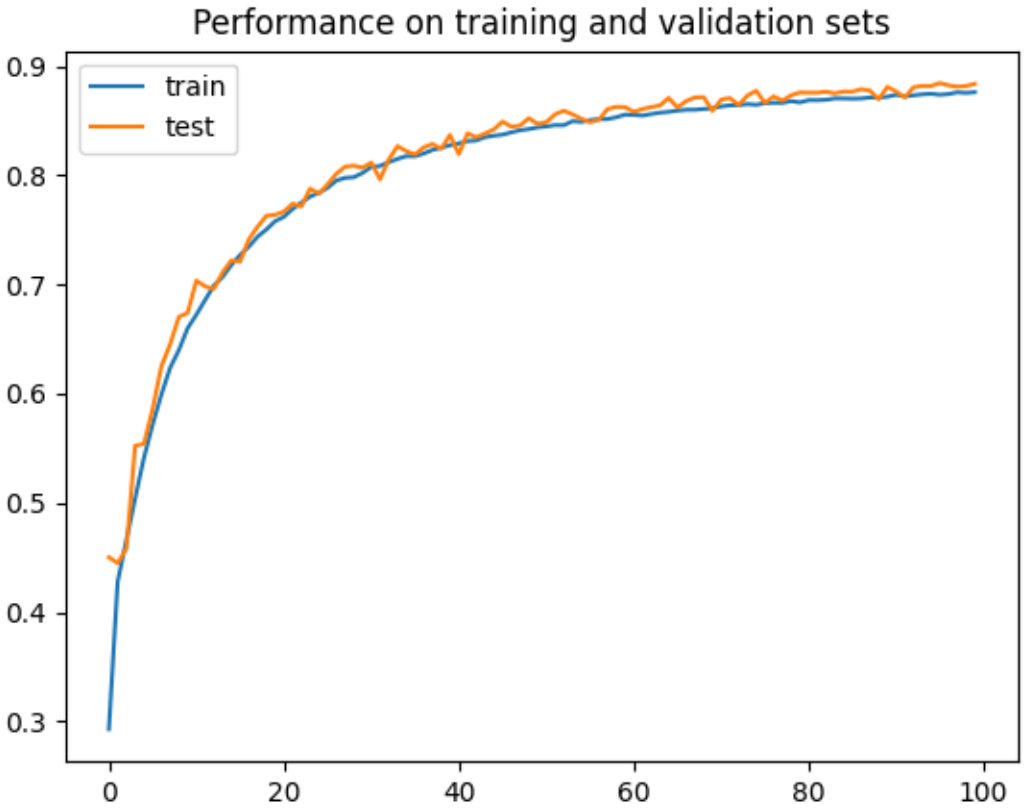
4.2 Initiliazer=InitCentersKmeans, Neurons=10

4.2.1 Εποχές 50, betas=0.01

Χρόνος Εκτέλεσης = 304s

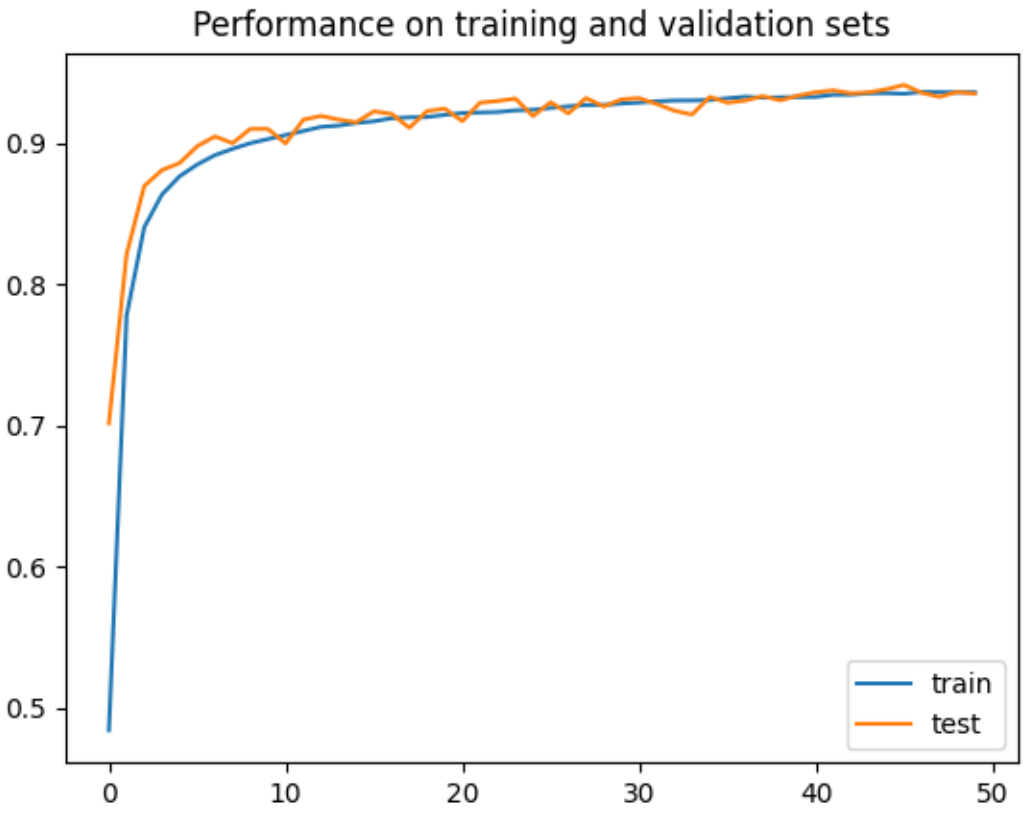
4.2.2 Εποχές 100, betas=0.005

Χρόνος Εκτέλεσης = 654s


Performance on training and validation sets
train
test
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
20
40
60
80
100

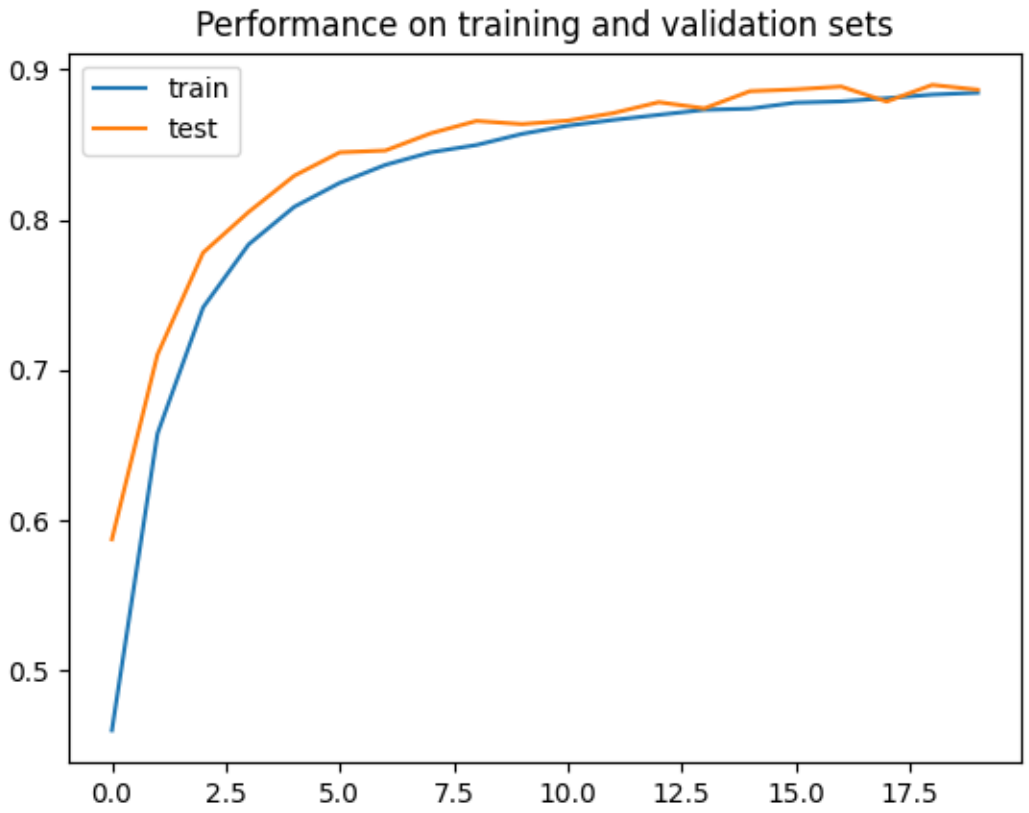
4.3 Initiliazer=InitCentersRandom, Neurons=50

4.3.1 Εποχές 50, betas=0.01



Χρόνος Εκτέλεσης = 1500s

4.3.2 Εποχές 20, betas=0.02



Χρόνος Εκτέλεσης = 1500s

Κώδικας ενδιάμεσης εργασίας:

```
from keras.datasets import mnist
from sklearn.neighbors import KNeighborsClassifier, NearestCentroid
from sklearn.metrics import accuracy_score
import time


clf1 = KNeighborsClassifier(1)
clf2 = KNeighborsClassifier(3)
CentroidClassifier = NearestCentroid()
```

```
(x_train,y_train), (x_test,y_test) = mnist.load_data()
x_train = x_train.reshape(x_train.shape[0], x_train.shape[1]**2)
x_test = x_test.reshape(x_test.shape[0], x_test.shape[1]**2)

start_time1 = time.time()
clf1.fit(x_train,y_train)
pred1 = clf1.predict(x_test)
print("Execution Time for kNN-1 Classifier --- %s seconds ---" % (time.time() -
start_time1))
print("Accuracy for kNN1 --- %s ---" % accuracy_score(y_test, pred1))

start_time2 = time.time()
clf2.fit(x_train,y_train)
pred2 = clf2.predict(x_test)
print("Execution for kNN-3 Classifier --- %s seconds ---" % (time.time() -
start_time2))
print("Accuracy for kNN1 --- %s ---" % accuracy_score(y_test, pred2))


start_time3 = time.time()
CentroidClassifier.fit(x_train,y_train)
pred3 = CentroidClassifier.predict(x_test)
print("Execution Time for Centroid Classifier --- %s seconds ---" % (time.time()
- start_time3))
print("Accuracy for kNN1 --- %s ---" % accuracy_score(y_test, pred3))

#Χρησιμοποιήσαμε την μετρική Accuracy για να εκτιμήσουμε την απόδοση των
Classifier μας στο συγκεκριμένο dataset.
#Με την βοήθεια της βιβλιοθήκης time καταφέραμε να μετρήσουμε τον Χρόνο εκτέλεσης
του κάθε ένα από τους Classifiers.
#Αφού δεν μπορούμε να εισάγουμε στο fit 3D data πρέπει να τα ανασχηματίσουμε σε
διαστάσεις που μας βολεύουν.
```

Με output στην κονσόλα:

Execution Time for kNN-1 Classifier --- 21.646045684814453 seconds ---

Accuracy for kNN1 --- 0.9691 ---

Execution for kNN-3 Classifier --- 18.950039625167847 seconds ---

Accuracy for kNN1 --- 0.9705 ---

Execution Time for Centroid Classifier --- 0.12199664115905762 seconds ---

Accuracy for kNN1 --- 0.8203 ---

5. Σχόλια

Όταν έχουμε πολλούς νευρώνες βλέπουμε ότι το μοντέλο μας δεν χρειάζεται τόσες εποχές όσες του δίνουμε για να φτάσει σε μια ικανοποιητική ακρίβεια, όμως η κάθε εποχή έχει πολύ μεγαλύτερο χρόνο περάτωσης.

Στο μοντέλο μας για μεγάλο betas δεν έχουμε επιθυμητή απόδοση και όσο το ανεβάζουμε η ακρίβεια θα πέφτει λόγω ταλαντώσεων, η καλύτερη ακρίβεια που πετύχαμε ήταν στο στη 4.1.2 και σύγκλινε σχετικά γρήγορα από την $20^{η}$ κιόλας εποχή αλλά και στις δοκιμές με τους 50 νευρώνες ειδικά το 4.3.1. Παρατηρούμε ότι ο Centroid Classifier δεν τα καταφέρνει καλά στο να ανεβάσει την μετρική σε επιθυμητά σημεία αλλά ο kNN καταφέρνει σε πολύ μικρό χρόνο να σχηματίσει ένα μοντέλο που έχει αρκετά μεγάλη ακρίβεια στο MNIST dataset, καλύτερη αυτής που δίνουν τα περισσότερα νευρωνικά μοντέλα που δοκιμάσαμε προηγουμένως. Την καλή απόδοση του kNN την αποδίδουμε στην χαμηλή πολυπλοκότητα του MNIST Dataset σε άλλα datasets περιμένουμε μεγαλύτερη απόδοση από το νευρωνικό μοντέλο.

Κούκας Γεώργιος 9486
