Aristotle University of Thessaloniki
School of Science
School of Informatics

Report in Convolutional Neural Networks trained with MNIST-DIGIT

Vasileios Asimakopoulos

21 Μαΐου 2023

Περιεχόμενα

Κατάλογος σχημάτων

Code Listings

Κεφάλαιο 1

CNN from Scratch

1.1 Brief code description

Σε αυτην την εργασία επιλέχτηκε να γραφεί ο κώδικα για το CNN μοντέλο με την βοήθεια της βιβλιοθήκης numpy. Συγκεκριμένα γράφηκε κώδικας για τα Dense Layers, Convolutonal Layers, Relu και Softmax Activation, Dropout Layers, Loss function(LossCategoricalCrossEntropy), και τέλος για τους ADAM και SGD optimizers. Οι κώδικες βρίσκονται στο παράρτημα A . Η αρχική ιδέα ήταν να γραφτεί ενα απλό νευρωνικό δίκτυο να δοκιμαστεί σε ένα μικρό dataset και μετά να εμπλουτιστεί αυτο το νευρωνικό με ένα συνελικτικό layer. Υπήρξε μία δυσκολία να δουλέψουν όλα μαζί λόγω προβλημάτων στην μορφη των πινάκων τελικά μπόρεσε και εγίνε το train πάνω στο dataset της MNIST-DIGIT.

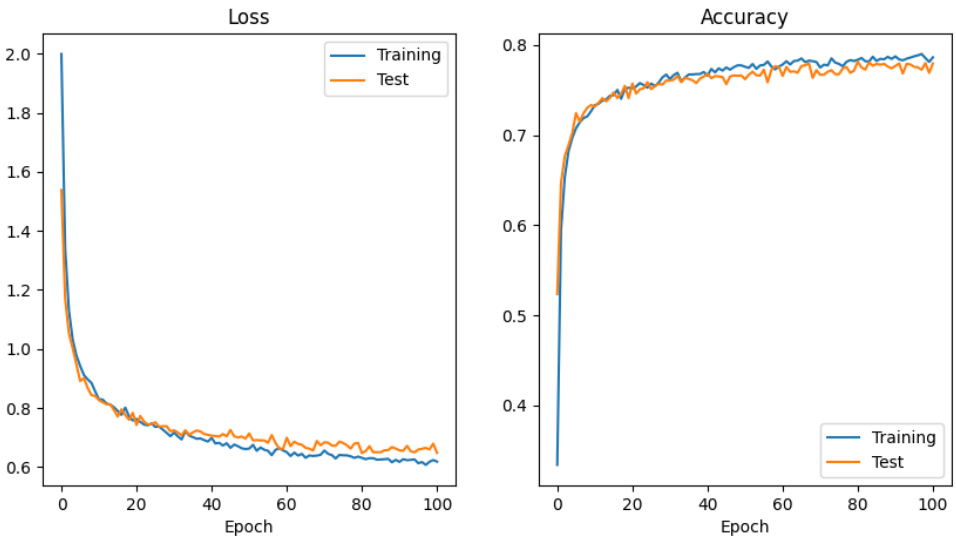
1.2 Training the model with different optimizers

Κατά το training επιλέχθηκε να μπεί ενα όριο στα δεδομένα που θα χρησιμοποιηθούν απο την MNIST-DIGIT λόγω χρόνου και μη βελτιστοποιημενου κώδικα να τρέχει σε παραπάνω απο 1 πυρήνες. Συγκεκριμένα επιλέχθηκε να τρέξουν όλα τα διαφορετικά μοντέλα σε 20000 δεδομένα. Τα μοντέλα είχαν κοινή αρχιτεκτονική (πέρα απο την 3η περίπτωση) αλλά διαφορετικούς optimizer. Στην 1η περίπτωση επιλέχθηκε να γίνεται βελτιστοποίηση μόνο με learning rate = 10e-5. Στην 2η περίπτωση επιλέχθηκε να χρησιμοποιηθεί ο SGD optimizer, με διαφορετικές παραμέτρους. Την 1η φορά έτρεξε το μοντέλο με SGD(learning rate=10e-4, decay=10e-5, momentum=0.5), και την 2η φορά με SGD(learning rate=10e-5, decay=10e-6, momentum=0.9). Ουσιαστικά μειώθηκαν την 2η φορά το learing rate και το decay ενώ το momentum ανέβηκε.Στην 3ή και τελευταί περίπτωση σύγκρισης των optimizers χρησιμοποιήθηκε ο ADAM με learning rate = 10e-5 και decay = 10e-6. Σε όλες τις περιπτώσεις το train πραγματοποιήθηκε για 100 εποχές, ενώ η αρχιτεκτονική του δικτύου ήταν : Convolution Layer((1, 28, 28), 3, 10) - Activation ReLU() - Dropout Layer(0.4) - Reshape((10, 26, 26), (10 * 26 * 26, 1)) - Layer Dense(10 * 26* 26,128) - Activation ReLU() - Dropout Layer(0.5) - Layer Dense(128,10) - Activation ReLU() - Dropout Layer(0.2) - Activation Softmax() - Loss CategoricalCrossentropy(). Στην 3η περίπτωση η αρχιτεκτόνικη άλλαξε, βγάζοντας το Dropout Layer μετά το Convolution Layer, και μειώνοντας το rate του επόμενου σε 0.2

1.2.1 Results of the models with different optimizers

Σέ όλες τις περιπτώσεις μετρήθηκε το training loss/accuracy και το validation loss/accuracy.

Στο διάγραμμα 1.1 παρατηρούνται τα αποτελέσματα της 1ής περίπτωσης. Φαίνεται οτι μόνο με learning rate το μοντέλο μαθαίνει αρκετά γρήγορα στις πρωτες 20 εποχές, ενω για τις υπόλοιπες 80 βλέπουμε να μην μπορεί να ξεπεράσει το 80%. Αντίστοιχα το loss του μοντέλου πέφτει απο το 2.0 στο 0.8 τις πρώτες 20 εποχές, και μετά σχεδόν σταθεροποιείται με την τιμη του loss να παραμενει μεταξυ 0.8 και 0.6.

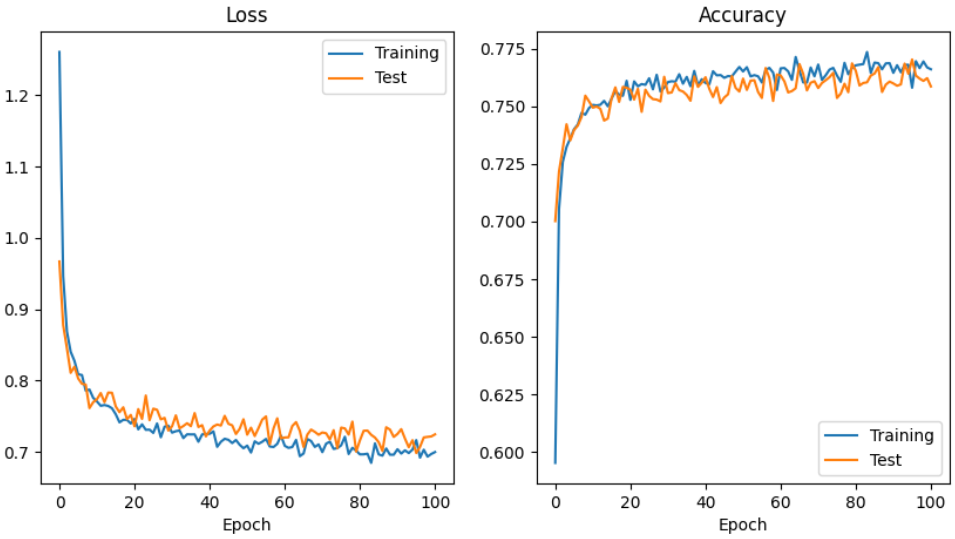


Σχήμα 1.1: Διάγραμμα loss, accuracy για την 1ή περίπτωση εκπαίδευσης του μοντέλου

```
epoch: 0, training loss: 1.999, training accuracy: 0.334, test loss: 1.538, test accuracy: 0.524, optimizer: 0.0001
epoch: 10, training loss: 0.829, training accuracy: 0.733, test loss: 0.827, test accuracy: 0.732, optimizer: 0.0001
epoch: 20, training loss: 0.763, training accuracy: 0.752, test loss: 0.743, test accuracy: 0.757, optimizer: 0.0001
epoch: 30, training loss: 0.716, training accuracy: 0.762, test loss: 0.724, test accuracy: 0.761, optimizer: 0.0001
epoch: 40, training loss: 0.699, training accuracy: 0.766, test loss: 0.706, test accuracy: 0.768, optimizer: 0.0001
epoch: 50, training loss: 0.662, training accuracy: 0.776, test loss: 0.713, test accuracy: 0.762, optimizer: 0.0001
epoch: 60, training loss: 0.650, training accuracy: 0.778, test loss: 0.699, test accuracy: 0.766, optimizer: 0.0001
epoch: 70, training loss: 0.656, training accuracy: 0.775, test loss: 0.684, test accuracy: 0.767, optimizer: 0.0001
epoch: 80, training loss: 0.631, training accuracy: 0.784, test loss: 0.647, test accuracy: 0.782, optimizer: 0.0001
epoch: 90, training loss: 0.617, training accuracy: 0.787, test loss: 0.657, test accuracy: 0.777, optimizer: 0.0001
epoch: 100, training loss: 0.618, training accuracy: 0.786, test loss: 0.648, test accuracy: 0.780, optimizer: 0.0001
```

Σχήμα 1.2: Πίνακας εποχών

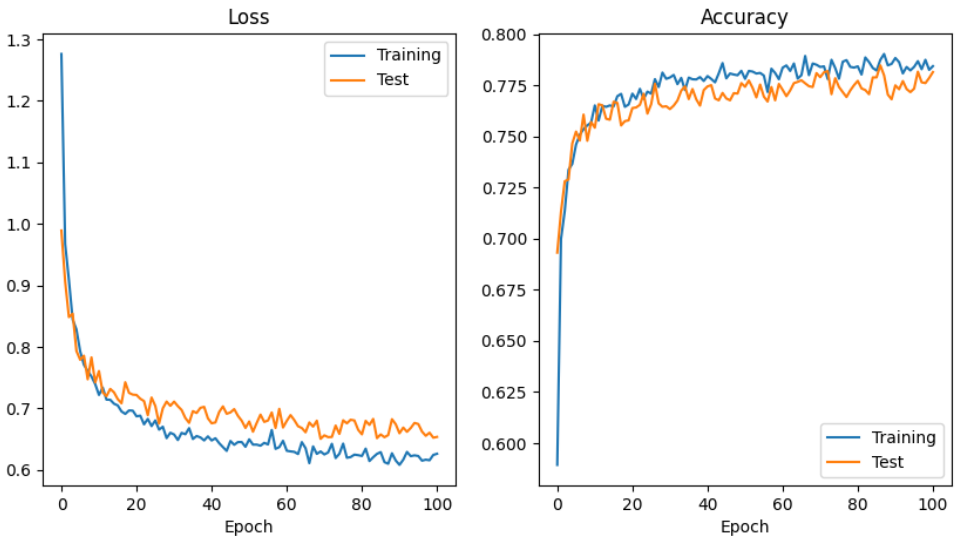
Στο διάγραμμα 1.3 παρατηρούνται τα αποτελέσματα της 2ης περίπτωσης θέτοντας των SGD με τις εξής παραμέτρους : learning rate=10e-4, decay=10e-5, momentum=0.5, ενώ στο δίαγραμμα 1.5 παρατηρούνται τα αποτελέσματα με SGD(learning rate=10e-5, decay=10e-6, momentum=0.9). Τόσο την 1η φορά οσο και την 2ή παρατηρείται το ίδιο μοτίβό με την προηγούμενη περίπτωση. Τις πρώτες 20 εποχές το loss πέφτει παρα πολύ φτάνοντας σχεδον στο 0.6, ενώ το accuracy αυξανεται παρα πολύ φτάνοντας σχεδόν στο 80% και μετά παραμένει σχεδον σταθερό. Αυτο φαινεται και στους πίνακες εποχών και των δύο training 1.4, 1.6.



Σχήμα 1.3: Διάγραμμα loss, accuracy για την 2ή περίπτωση εκπαίδευσης του μοντέλου με παραμέτρους : learning rate=10e-4, decay=10e-5, momentum=0.5

```
epoch: 0, training loss: 1.261, training accuracy: 0.595, test loss: 0.967, test accuracy: 0.700, optimizer: 0.0003333444448148271 3
epoch: 10, training loss: 0.771, training accuracy: 0.751, test loss: 0.774, test accuracy: 0.749, optimizer: 4.3478449906303936e-05
epoch: 20, training loss: 0.746, training accuracy: 0.753, test loss: 0.736, test accuracy: 0.757, optimizer: 2.325586803690241e-05
epoch: 30, training loss: 0.727, training accuracy: 0.760, test loss: 0.736, test accuracy: 0.756, optimizer: 1.5873041068319158e-05
epoch: 40, training loss: 0.725, training accuracy: 0.760, test loss: 0.730, test accuracy: 0.763, optimizer: 1.2048207286996731e-05
epoch: 50, training loss: 0.709, training accuracy: 0.765, test loss: 0.724, test accuracy: 0.762, optimizer: 9.708747290045912e-06
epoch: 60, training loss: 0.709, training accuracy: 0.766, test loss: 0.720, test accuracy: 0.763, optimizer: 8.130087910640576e-06
epoch: 70, training loss: 0.700, training accuracy: 0.768, test loss: 0.727, test accuracy: 0.758, optimizer: 6.993011883225094e-06
epoch: 80, training loss: 0.697, training accuracy: 0.768, test loss: 0.716, test accuracy: 0.765, optimizer: 6.134973088940546e-06
epoch: 90, training loss: 0.704, training accuracy: 0.764, test loss: 0.725, test accuracy: 0.760, optimizer: 5.464438603736946e-06
epoch: 100, training loss: 0.700, training accuracy: 0.766, test loss: 0.725, test accuracy: 0.759, optimizer: 4.9261108010398045e-06
```

Σχήμα 1.4: Πίνακας εποχών

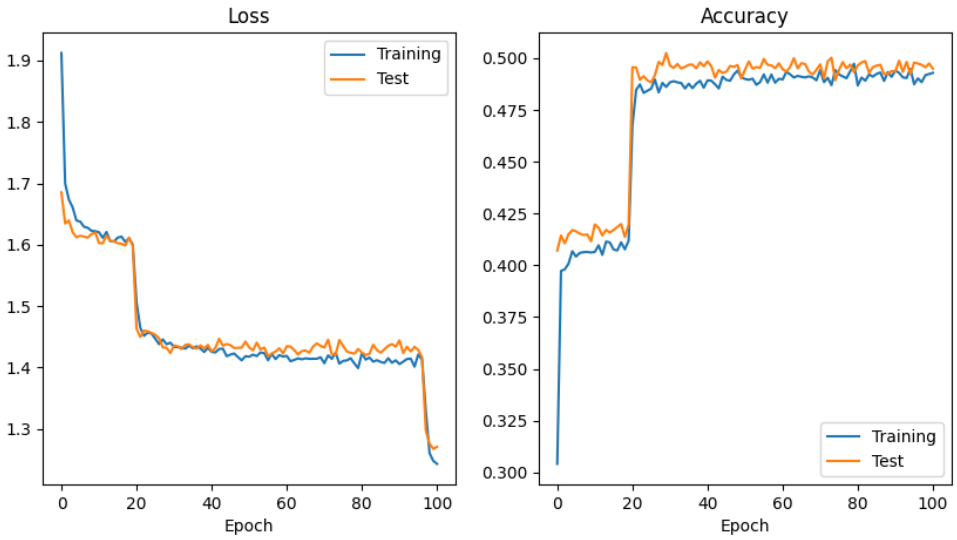


Σχήμα 1.5: Διάγραμμα loss, accuracy για την 2ή περίπτωση εκπαίδευσης του μοντέλου με παραμέτρους : learning rate=10e-5, decay=10e-6, momentum=0.8

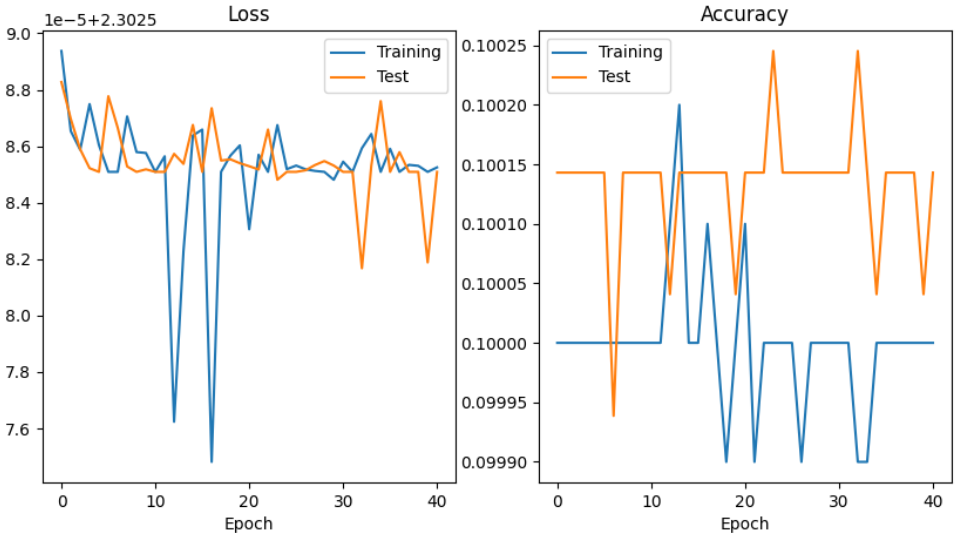
```
epoch: 0, training loss: 1.276, training accuracy: 0.589, test loss: 0.989, test accuracy: 0.693, optimizer: 8.333402778356486e-05
epoch: 10, training loss: 0.722, training accuracy: 0.765, test loss: 0.761, test accuracy: 0.754, optimizer: 3.125009765655518e-05
epoch: 20, training loss: 0.687, training accuracy: 0.771, test loss: 0.722, test accuracy: 0.764, optimizer: 1.923080621308887e-05
epoch: 30, training loss: 0.658, training accuracy: 0.779, test loss: 0.711, test accuracy: 0.763, optimizer: 1.3888908179039137e-05
epoch: 40, training loss: 0.648, training accuracy: 0.780, test loss: 0.676, test accuracy: 0.774, optimizer: 1.0869577032148946e-05
epoch: 50, training loss: 0.650, training accuracy: 0.778, test loss: 0.679, test accuracy: 0.774, optimizer: 8.928579400517322e-06
epoch: 60, training loss: 0.631, training accuracy: 0.784, test loss: 0.680, test accuracy: 0.773, optimizer: 7.5757633149722075e-06
epoch: 70, training loss: 0.625, training accuracy: 0.784, test loss: 0.656, test accuracy: 0.779, optimizer: 6.578951696678748e-06
epoch: 80, training loss: 0.623, training accuracy: 0.784, test loss: 0.658, test accuracy: 0.777, optimizer: 5.813956868579575e-06
epoch: 90, training loss: 0.608, training accuracy: 0.788, test loss: 0.660, test accuracy: 0.775, optimizer: 5.208336046008356e-06
epoch: 100, training loss: 0.626, training accuracy: 0.784, test loss: 0.654, test accuracy: 0.781, optimizer: 4.716983357067621e-06
```

Σχήμα 1.6: Πίνακας εποχών

Τέλος στο διάγραμμα 1.7 παρατηρούνται τα αποτελέσματα της 3ης περίπτωσης, δηλαδή την χρήσης ADAM. Στην συγκεκριμένη περίπτωση παρατηρήθηκε κάτι πολυ περίεργο. Όσες φορες και να ετρεξε το μοντέλο, με τον adam για optimizer, πάντα κολλούσε το training είτε στην αρχη με 10% είτε όπως βλέπουμε και στο διάγραμμα 1.7 , όπου ξεκινάει με ένα καλό accuracy ανεβαίνει γρηγορα μετα σταθεροποιείται για λίγο ξανα αναβαινει και παραμένει σχεδόν σταθερό.Η διαφορά με τα άλλα δυο μοντέλα είναι ότι αυτό εξαρτάται πάρα πολύ απο την αρχικοποίηση των βαρών των Layers, κατι που μπορεί να το κολλήσει και να βγάλει ένα διάγραμμα σαν αυτό 1.8. Παρόλο που στο συγκεκριμένο διάγραμμα το μοντέλο εκπαιδεύτηκε σε 10000 δεδομένα, παρόμοια αποτελέσματα παρατηρήθηκαν και με παραπάνω δεδομένα. Ποσοστό επιτυχίας άνω του 50 % δεν παρατηρήθηκε ποτέ ακόμα και με την αρχιτεκτονική των προηγούμεων περιπτώσεων.



Σχήμα 1.7: Διάγραμμα loss, accuracy για την 3ή περίπτωση εκπαίδευσης του μοντέλου για 20k δεδομενα



Σχήμα 1.8: Διάγραμμα loss, accuracy για την 3ή περίπτωση εκπαίδευσης του μοντέλου για 10k δεδομένα

1.3 Conclusion

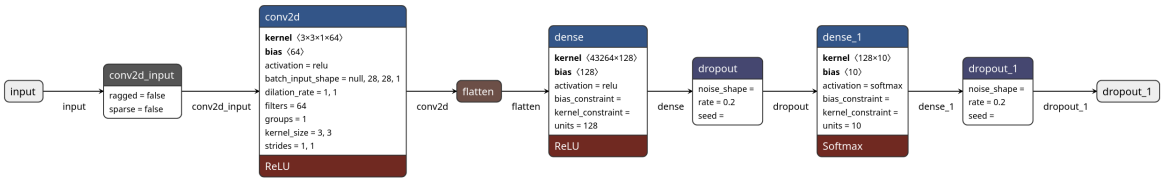
Στην συνεχεια, επιλέχθηκε ο στοχαστικός (SGD) με παραμέτρους learning rate=10e-5, decay=10e-6, momentum=0.8, με την αρχιτεκτονική του δικτύου που χρησιμοποιήθηκε στον ADAM με την μόνη αλλαγη να βρισκεται στο Convolution Layer, όπου οι kernels αυξηθηκαν απο 10 σε 64 : Convolution Layer((1, 28, 28), 3, 64). Αυτό το μοντέλο επιλέχθηκε να εκπαιδευτει σε 40000 δεδομένα στην MNIST-DIGIT και να συγκριθεί με το αντιστοιχο δίκτυο γραμμένο στην tensorflow.

Κεφάλαιο 2

Results of CNN from scratch vs CNN using tensorflow

2.1 Tensorflow CNN

Δημιουργήθηκαν δύο δίκτυα με την βοήθεια της tensorflow. Και τα δύο έχουν κοινή αρχιτεκτονική , αλλα διαφορετικούς optimizer. Η αρχιτεκτονική τους απεικονίζεται στην εικόνα 2.1. Στο 1ο δίκτυο χρησιμοποιήθηκε ο στοχαστικός optimizer με ίδιες παραμέτρους με του τελικού from scratch δικτύου (tensorflow CNN-SGD), ενώ στο 2ο χρησιμοποιήθηκε μονο learning rate με τίμη 10e-5 (tensorflow CNN-lr).



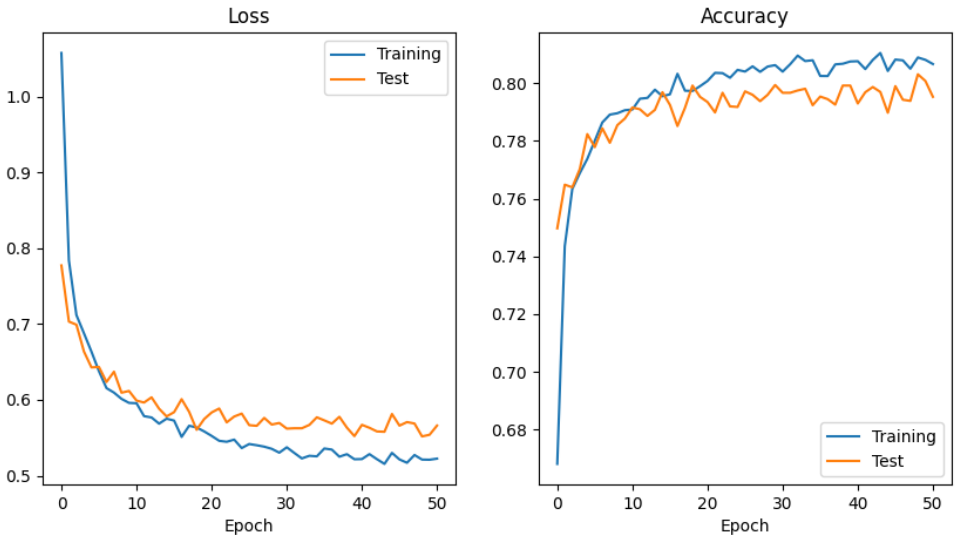
Σχήμα 2.1: Απεικόνιση αρχιτεκτονική CNN δικτύου

2.2 Numpy CNN vs Tensorflow CNN

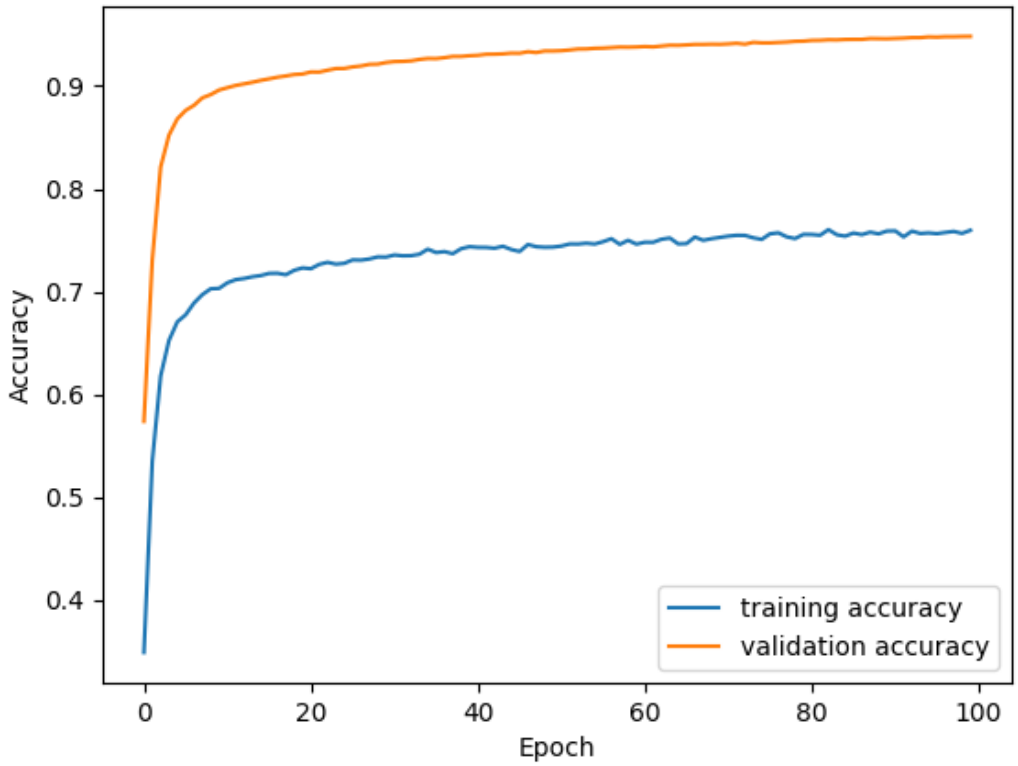
Στο τελευταίο μέρος της εργασίας συγκρίθηκαν τα αποτελέσματα επιτυχίας τόσο στο training οσο και στο testing(validation) μεταξύ του τελικού μοντέλου CNN γραμμένο σε numpy και των δύο μοντέλων CNN γραμμένων σε tensorflow. Το dataset που επιλέχθηκε είναι το ίδιο με πριν, απλά απο 20000 δεδομένα αυξήθηκε σε 40000 δεδομένα στο training και το validation παρέμεινε στο μεγιστο των 10000 δεδομένων.

2.2.1 Numpy CNN vs Tensorflow CNN-SGD

Αρχικά θα συγκρίνουμε το τελικό μοντέλο της numpy και το μοντέλο της tensorflow που έχουν ακριβως την ίδια αρχιτεκτονική (με ίδεις παραμέτρους στον optimizer SGD). Στο διάγραμμα 2.2 παρατηρούμε τα ίδια αποτελέσματα με τις περιπτώσεις των optimizer στο κεφάλαιο 1,δηλαδή αυξάνεται το accuracy αρκετά γρήγορα και μετά το 80% παραμένει σχεδόν σταθερό. Απο την άλλη στο διάγραμμα 2.3 , το validation accuracy ανεβαίνε σχεδόν στο 91%, αλλά το training accuracy δεν μπορεί να ξεπεράσει το 70%. Το κενό μεταξύ των δυο ποσοστών επιτυχίας, υποδηλώνει ότι έχουμε μικρό evaluation bias δηλαδή το dataset που κανουμε validate είναι πολυ μικρό Ίσως αν είχαμε μεγαλύτερο dataset για validation, το μοντέλο να μην έφτανε σε underfitting.



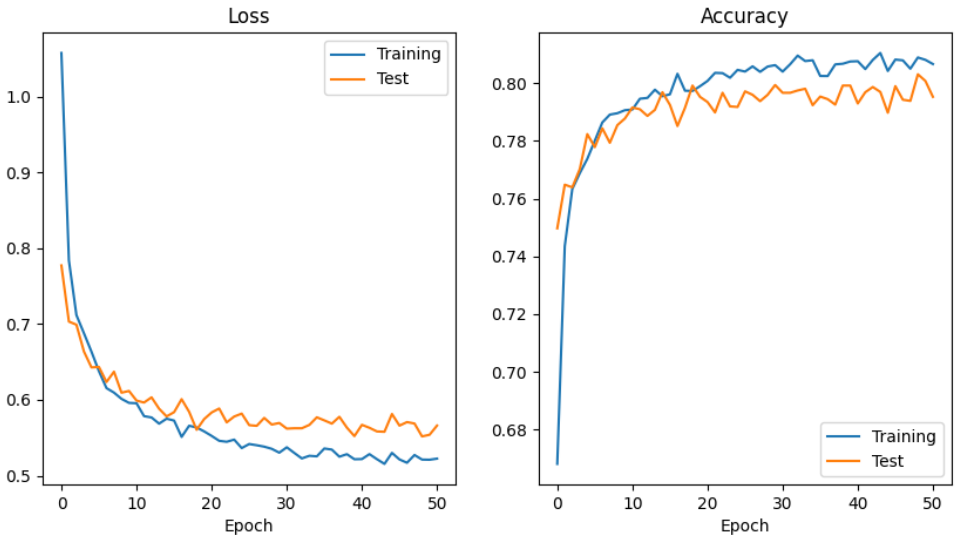
Σχήμα 2.2: Διάγραμμα loss, accuracy numpy CNN δικτύου



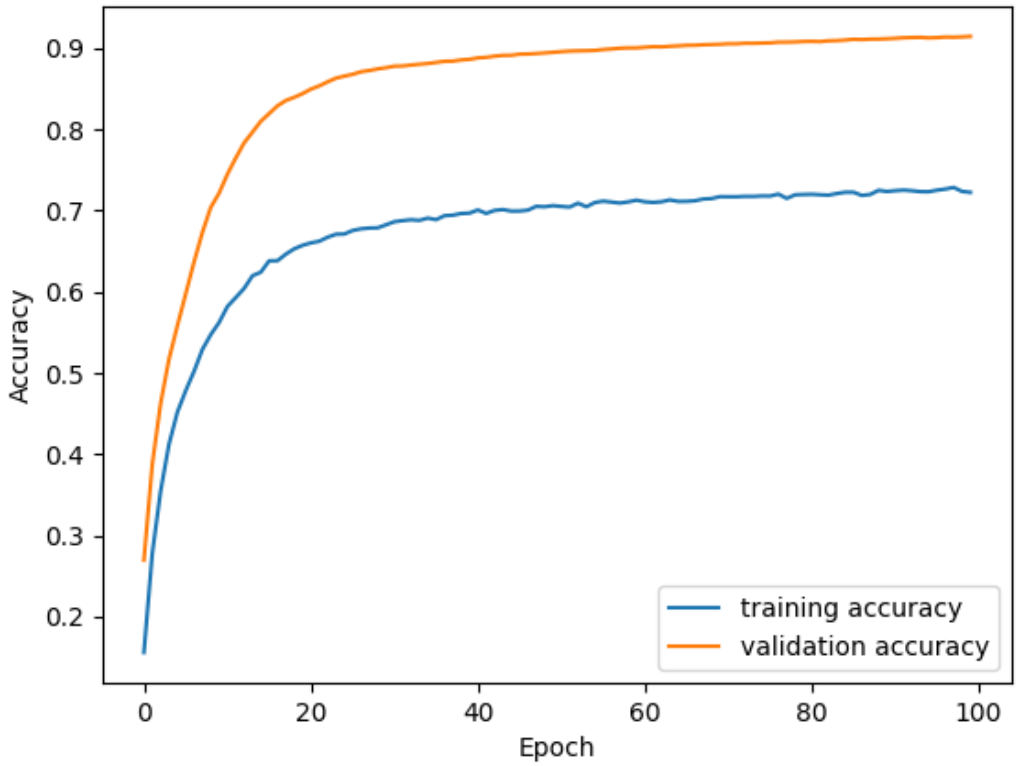
Σχήμα 2.3: Διάγραμμα accuracy tensorflow CNN-SGD δικτύου

2.2.2 Numpy CNN vs Tensorflow CNN-lr

Στην συνέχεια συγκρίνουμε το τελικό μοντέλο της που δημιουργήθηκε με την βοήθεια της numpy και το μοντέλο της tensorflow παρόμοια αρχιτεκτονική με το 1ο απλά η μόνη παράμετρος που χρησιμοποιείται είναι το learning rate. Στο διάγραμμα 2.5 παρατηρούμε μία πιο ομαλή αυξηση του ποσοστού επιτυχίας απο το προηγούμενο μοντέλο με tensorflow, παρόλα αυτα τα αποτελέσματα παραμένουν τα ίδια, δηλαδή και έδω βλέπουμε underfitting, που ίσως να οφείλεται σε αυτο που αναλύθηκε παραπάνω.



Σχήμα 2.4: Διάγραμμα loss, accuracy numpy CNN δικτύου



Σχήμα 2.5: Διάγραμμα accuracy tensorflow CNN-lr δικτύου

2.3 Conclusion

Παρόλο που πέτυχαν ποσοστό επιτυχίας άνω του 90% έστω και στο validation, τα δύο μοντέλα της tensorflow, το μοντέλο που πραγματοποιήθηκε απο την αρχή μόνο με numpy είχε πιο σταθερά αποτελέσματα, άν και χαμηλά για το συγκεκριμένο dataset. Επισης αξιοσημείωτο είναι το γεγονός ότι ο χρόνος που πήρε τα δύο μοντέλα να επαιδευτουν ήταν περίπου 1 ώρα ένω το from-scratch μοντέλο έκανε περίπου 60 ώρες στον Αριστοτέλη(την συστοιχία του ΑΠΘ).

Παράρτημα Α

Code

Α.1 Code CNN-numpy

Code Listing Α.1: Convolutional Layer

```
import numpy as np
from scipy import signal

class Convolution_Layer(Layer):
def __init__(self, input_shape, kernel_size, depth):
input_depth, input_height, input_width = input_shape
self.depth = depth
self.input_shape = input_shape
self.input_depth = input_depth
self.output_shape = (depth, input_height - kernel_size + 1,
    input_width - kernel_size + 1)
self.kernels_shape = (depth, input_depth, kernel_size,
   kernel_size) # Το depth μαςδίνειτοναριθμοτων kernels,
   Το input depth τιςδιαστασειςτου input, καιτο kernel size
    μαςδινειτομεγεθοςτουπινακαστους kernel
self.kernels = np.random.randn(*self.kernels_shape)
                    # Γιαπαραδειγμαανεχωενατρισδιαστατο
   input(3 καναλια), καιεχωβαλει depth 5, και kernel sizei
   3*3, θαεχω (5,3,3,3)
self.biases = np.random.randn(*self.output_shape)

def forward(self, input):
self.input = input
self.output = np.copy(self.biases)
for i in range(self.depth):
for j in range(self.input_depth):
self.output[i] += signal.correlate2d((self.input[j]), (self
   .kernels[i, j]), "valid")
return self.output

def backwards(self, d_output):
self.d_kernels = np.zeros(self.kernels_shape)
self.d_inputs= np.zeros(self.input_shape)

for i in range(self.depth):
```

```
for j in range(self.input_depth):
self.d_kernels[i, j] = signal.correlate2d(self.input[j],
   d_output[i], "valid")
self.d_inputs[j] += signal.convolve2d(d_output[i], self.
   kernels[i, j], "full")
self.d_biases = np.sum(d_output,axis=0,keepdims=True)

class Layer:
def __init__(self):
self.input = None
self.output = None

def forward(self, input):
# TODO: return output
pass

def backward(self, output_gradient):
# TODO: update parameters and return input gradient
pass
```

Code Listing A.2: Dense Layer

```
class Layer_Dense:
#Initialization of Layers
def __init__(self,n_inputs,n_neurons):
self.weights = 0.01* np.random.randn(n_inputs, n_neurons)
self.biases = np.zeros((1,n_neurons))

#Forward pass
def forward(self,inputs):
self.inputs = inputs
self.output = np.dot(inputs,self.weights) + self.biases

def backwards(self, d_output):
self.d_weights = np.dot(self.inputs.T,d_output)

self.d_biases = np.sum(d_output,axis=0,keepdims=True)

self.d_inputs = np.dot(d_output,self.weights.T)
```

Code Listing A.3: ReLU Layer

```
class Activation_ReLU:
def forward(self,inputs):
self.inputs = inputs
self.output = np.maximum(0,inputs)

def backwards(self, d_output):
self.d_inputs = d_output.copy()

self.d_inputs[self.inputs<=0] = 0
```

Code Listing A.4: Reshape Layer

```
class Reshape(Layer):
def __init__(self, input_shape, output_shape):
self.input_shape = input_shape
self.output_shape = output_shape

def forward(self, input):
self.output = np.reshape(input, self.output_shape)
self.output= self.output.T

def backwards(self, output_gradient):
self.d_inputs = np.reshape(output_gradient, self.
   input_shape)

        class Layer:
def __init__(self):
self.input = None
self.output = None

def forward(self, input):
# TODO: return output
pass

def backward(self, output_gradient):
# TODO: update parameters and return input gradient
pass
```

Code Listing A.5: Dropout Layer

```
import numpy as np

class Dropout_Layer:

def __init__(self, rate):
self.rate = 1 -rate

def forward(self, inputs):

self.inputs = inputs


self.binary_mask = np.random.binomial(1,self.rate,size=
   inputs.shape) / self.rate

self.output = inputs * self.binary_mask

def backwards(self, d_output):

self.d_inputs = d_output * self.binary_mask
```

Code Listing A.6: Softmax Activation

```
import numpy as np
```

```
class Activation_Softmax:

def __init__(self):
self.predictions = []
def forward(self,inputs):
self.inputs = inputs

exp_values = np.exp(inputs-np.max(inputs, axis=1, keepdims=
    True))
probabilities = exp_values / np.sum(exp_values, axis=1,
    keepdims=True)
self.output = probabilities
self.predictions.append(probabilities)
# print(predictions)


def backwards(self,d_output):
self.d_inputs = np.empty_like(d_output)

for index, (single_output,single_d_output) in enumerate(zip
    (self.output,d_output)):
single_output = single_output.reshape(-1,1)

jacobian_matrix = np.diagflat(single_output) - np.dot(
    single_output,single_output.T)

self.d_inputs[index] = np.dot(jacobian_matrix,
    single_d_output)


def predictions(self,output):
# print(np.argmax(output,axis=1))
return np.argmax(output,axis=1)
```

Code Listing Α.7: Loss function

```
import numpy as np

class Loss:
def remember_trainable_layers(self,trainable_layers):
self.trainable_layers = trainable_layers

def calculate(self,output,y):
sample_losses= self.forward(output,y)

data_loss = np.mean(sample_losses)


return data_loss

class Loss_CategoricalCrossentropy(Loss):
```

```
        def forward(self,y_pred,y_true):
        samples = len(y_pred)
        y_pred_clipped = np.clip(y_pred, 10e-3, 1.0 - 10e-3)

)
        if len(y_true.shape) == 1:
        correct_confidences = y_pred_clipped [range(samples),
           y_true.T]

        elif len(y_true.shape) == 2:
        correct_confidences = np.sum(y_pred_clipped*y_true,axis=1)
        negative_log_likelihood = -np.log(correct_confidences)
        return negative_log_likelihood

        def backwards(self,d_output,y_true):
        samples =len(d_output)
        labels = len(d_output[0])
        if len(y_true.shape) == 1:
        print('yehaw')
        y_true = np.eye(labels)[y_true]

        self.d_inputs = -y_true.T/d_output
        self.d_inputs = self.d_inputs /samples
```

Code Listing A.8: Optimizer function

```
        import numpy as np
        from Conv_layer import *
        from Dense_layer import *

        class Optimizer_SGD:

        def __init__(self, learning_rate=0.01, decay=0, momentum=0)
           :
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.momentum = momentum

        def pre_update_parameters(self):
        if self.decay:
        self.current_learning_rate = self.learning_rate * (1. / (1.
            + self.decay * self.iterations))

        def update_parameters(self, layer):
        if isinstance(layer, Convolution_Layer):
        if self.momentum:

        if not hasattr(layer, 'kernel_momentums'):

        layer.kernel_momentums = np.zeros_like(layer.kernels)
```

```
layer.bias_momentums = np.zeros_like(layer.biases)

kernel_updates = self.momentum * layer.kernel_momentums -
    self.current_learning_rate * layer.d_kernels
layer.kernel_momentums = kernel_updates

bias_updates = self.momentum * layer.bias_momentums - self.
    current_learning_rate * layer.d_biases
layer.bias_momentums = bias_updates

else :
kernel_updates= -self.current_learning_rate * layer.
    d_kernels
bias_updates = -self.current_learning_rate * layer.d_biases

layer.kernels += kernel_updates
layer.biases += bias_updates

elif isinstance(layer, Layer_Dense):
if self.momentum:

if not hasattr(layer, 'weight_momentums'):
layer.weight_momentums = np.zeros_like(layer.weights)
layer.bias_momentums = np.zeros_like(layer.biases)

weight_updates = self.momentum * layer.weight_momentums -
    self.current_learning_rate * layer.d_weights
layer.weight_momentums = weight_updates

bias_updates = self.momentum * layer.bias_momentums - self.
    current_learning_rate * layer.d_biases
layer.bias_momentums = bias_updates

else :
weight_updates= -self.current_learning_rate * layer.
    d_weights
bias_updates = -self.current_learning_rate * layer.d_biases
layer.weights += weight_updates
layer.biases += bias_updates

def post_update_parameters(self):
self.iterations += 1




class Optimizer_Adam:

def __init__(self, learning_rate=10e-4, decay = 0, epsilon
```

```
    = 1e-7, beta1=0.9, beta2=0.999):
self.learning_rate = learning_rate
self.current_learning_rate = learning_rate
self.decay = decay
self.iterations = 0
self.epsilon = epsilon
self.beta1 = beta1
self.beta2 = beta2

def pre_update_parameters(self):
if self.decay:
self.current_learning_rate = self.learning_rate * (1. /(1.
    + self.decay * self.iterations))


def update_parameters(self, layer):
if isinstance(layer, Convolution_Layer):
if not hasattr(layer, 'kernels_cache'):
layer.bias_cache = np.zeros_like(layer.biases)
layer.bias_momentums = np.zeros_like(layer.biases)
layer.kernels_cache = np.zeros_like(layer.kernels)
layer.kernels_momentums = np.zeros_like(layer.kernels)

layer.bias_momentums = self.beta1 * layer.bias_momentums +
    (1 - self.beta1) * layer.d_biases
layer.kernels_momentums = self.beta1 * layer.
    kernels_momentums + (1 - self.beta1) * layer.d_kernels

bias_momentums_corrected = layer.bias_momentums / (1 - self
    .beta1 ** (self.iterations +1))
kernels_momentums_corrected = layer.kernels_momentums / (1
    - self.beta1 ** (self.iterations +1))

layer.bias_cache = self.beta2 * layer.bias_cache + (1 -
    self.beta2) * layer.d_biases**2
layer.kernels_cache = self.beta2 * layer.kernels_cache + (1
     - self.beta2) * layer.d_kernels**2

bias_cache_corrected = layer.bias_cache / (1 - self.beta2
    ** (self.iterations+1))
kernels_cache_corrected = layer.kernels_cache / (1 - self.
    beta2 ** (self.iterations+1))

layer.biases += -self.current_learning_rate *
    bias_momentums_corrected / (np.sqrt(bias_cache_corrected
    ) + self.epsilon)
layer.kernels += -self.current_learning_rate *
    kernels_momentums_corrected / (np.sqrt(
    kernels_cache_corrected) + self.epsilon)
```

```
elif isinstance(layer, Layer_Dense):
if not hasattr(layer, 'weight_cache'):
layer.weight_cache = np.zeros_like(layer.weights)
layer.weight_momentums = np.zeros_like(layer.weights)
layer.bias_cache = np.zeros_like(layer.biases)
layer.bias_momentums = np.zeros_like(layer.biases)

layer.weight_momentums = self.beta1 * layer.
    weight_momentums + (1 - self.beta1) * layer.d_weights
layer.bias_momentums = self.beta1 * layer.bias_momentums +
    (1 - self.beta1) * layer.d_biases
# print(layer.weight_momentums)
weight_momentums_corrected = layer.weight_momentums / (1 -
    self.beta1 ** (self.iterations +1))
bias_momentums_corrected = layer.bias_momentums / (1 - self
    .beta1 ** (self.iterations +1))

layer.weight_cache = self.beta2 * layer.weight_cache + (1 -
     self.beta2) * layer.d_weights**2
layer.bias_cache = self.beta2 * layer.bias_cache + (1 -
    self.beta2) * layer.d_biases**2


weight_cache_corrected = layer.weight_cache / (1 - self.
    beta2 ** (self.iterations+1))
bias_cache_corrected = layer.bias_cache / (1 - self.beta2
    ** (self.iterations+1))

layer.weights += -self.current_learning_rate *
    weight_momentums_corrected / (np.sqrt(
    weight_cache_corrected) + self.epsilon)
layer.biases += -self.current_learning_rate *
    bias_momentums_corrected / (np.sqrt(bias_cache_corrected
    ) + self.epsilon)
# print(layer.weights)
def post_update_parameters(self):
self.iterations +=1
```

Code Listing Α.9: Model

```
import numpy as np
from Conv_layer import *
from Softmax_function import *
from Loss_function import *
from Dropout_layer import *
from Dense_layer import *
from Optimizer_function import *
import tensorflow as tf
from keras.datasets import mnist
from keras.utils import np_utils
import matplotlib.pyplot as plt
from tqdm import tqdm
```

```
import time as time


plot_filename = "plot.png"
model_output = "model.txt"

# Preprocess Function
data_limit = 100
def preprocess_data(x, y):
x_new = []
y_new = []
for i in range(10):
indices = np.where(y == i)[0][:data_limit]
np.random.shuffle(indices)  # Shuffle the indices before
    selecting data
x_new.append(x[indices])
y_new.append(np.full(len(indices), i))
x_new = np.concatenate(x_new)
y_new = np.concatenate(y_new)
x_new = x_new.reshape(len(x_new), 1, 28, 28)
x_new = x_new.astype("float32") / 255
y_new = np_utils.to_categorical(y_new)
y_new = y_new.reshape(len(y_new), 10, 1)
indices = np.random.permutation(len(x_new))  # Shuffle the
    data
return x_new[indices], y_new[indices]


# Load mnist and preprocess data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
X_train, y_train = preprocess_data(x_train, y_train)
X_test, y_test = preprocess_data(x_test, y_test)

# Initalize layers and functions
dropout_dense_1 = Dropout_Layer(0.5)
dropout_dense_2 = Dropout_Layer(0.2)
conv_1 = Convolution_Layer((1, 28, 28), 3, 10)
flatten=Reshape((10, 26, 26), (10 * 26 * 26, 1))
dense_1 = Layer_Dense(10 * 26* 26,128)
dense_2=Layer_Dense(128,10)
activation_conv_1 = Activation_ReLU()
activation_dense_1 = Activation_ReLU()
activation_dense_2 = Activation_ReLU()
softmax = Activation_Softmax()

loss_function = Loss_CategoricalCrossentropy()
# optimizer = Optimizer_Adam(learning_rate=10e-5,decay=10e
    -6)
optimizer = Optimizer_SGD(learning_rate=10e-5,decay=0)

train_losses = []
```

```
train_accuracies = []
test_losses = []
test_accuracies = []



with open(model_output, "w") as f:
for epoch in tqdm(range(41)):
softmax.predictions =[]
start_time = time.time()
for x, y in tqdm(zip(X_train, y_train), desc=f'Epoch {epoch
    }'):

# Forward pass
conv_1.forward(x)
activation_conv_1.forward(conv_1.output)
flatten.forward(activation_conv_1.output)
dense_1.forward(flatten.output)
activation_dense_1.forward(dense_1.output)
dropout_dense_1.forward(activation_dense_1.output)
dense_2.forward(dropout_dense_1.output)
activation_dense_2.forward(dense_2.output)
dropout_dense_2.forward(activation_dense_2.output)
softmax.forward(dropout_dense_2.output)
loss_function.forward(softmax.output, y)

# Backward pass
loss_function.backwards(softmax.output, y)
softmax.backwards(loss_function.d_inputs)
dropout_dense_2.backwards(softmax.d_inputs)
activation_dense_2.backwards(dropout_dense_2.d_inputs)
dense_2.backwards(activation_dense_2.d_inputs)
dropout_dense_1.backwards(dense_2.d_inputs)
activation_dense_1.backwards(dropout_dense_1.d_inputs)
dense_1.backwards(activation_dense_1.d_inputs)
flatten.backwards(dense_1.d_inputs)
activation_conv_1.backwards(flatten.d_inputs)
conv_1.backwards(activation_conv_1.d_inputs)

# Optimize gradients
optimizer.pre_update_parameters()
optimizer.update_parameters(conv_1)
optimizer.update_parameters(dense_1)
optimizer.update_parameters(dense_2)
optimizer.post_update_parameters()

# Training accuracy calculation
y_reshaped = y_train.reshape(data_limit*10,10)
predictions_array =np.asarray(softmax.predictions)
predictions_reshaped = predictions_array.reshape(y_reshaped
```

```
    .shape)
predictions_reshaped_true= np.argmax(predictions_reshaped,
    axis=1)
if len(y_reshaped.shape) == 2:
train_labels=np.argmax(y_reshaped,axis=1)
train_predictions = predictions_reshaped_true.reshape(
    train_labels.shape)
training_accuracy = np.mean(train_predictions==train_labels
    )
train_accuracies.append(training_accuracy)
train_accuracies.append(training_accuracy)

#Training loss calculation
loss = loss_function.calculate(predictions_reshaped,
    y_reshaped)
train_losses.append(loss)


# Test predictions
softmax.predictions = []
test_losses_epoch = []
test_predictions = []
for x, y in zip(X_test, y_test):

# Forward pass
conv_1.forward(x)
activation_conv_1.forward(conv_1.output)
flatten.forward(activation_conv_1.output)
dense_1.forward(flatten.output)
activation_dense_1.forward(dense_1.output)
dropout_dense_1.forward(activation_dense_1.output)
dense_2.forward(dropout_dense_1.output)
activation_dense_2.forward(dense_2.output)
dropout_dense_2.forward(activation_dense_2.output)
softmax.forward(dropout_dense_2.output)
loss_function.forward(softmax.output, y)


# Calculate test accuracy and loss
test_predictions = np.asarray(softmax.predictions)
test_predictions = test_predictions.reshape(len(
    test_predictions), -1)
test_labels = y_test.reshape(len(y_test), -1)
test_loss = loss_function.calculate(test_predictions,
    test_labels)
test_accuracy = np.mean(np.argmax(test_predictions, axis=1)
     == np.argmax(test_labels, axis=1))
test_losses_epoch.append(test_loss)
test_losses.append(np.mean(test_losses_epoch))
test_accuracies.append(test_accuracy)
```

```
    if not epoch % 10:
    end_time = time.time()
    elapsed_time = end_time - start_time
    output = (f"Epoch {epoch} took {elapsed_time:.2f} seconds, "
        "
    f"training loss: {loss:.3f}, "
    f"training accuracy: {training_accuracy:.3f}, "
    f"test loss: {test_loss:.3f}, "
    f"test accuracy: {test_accuracy:.3f}, "
    f"optimizer: {optimizer.current_learning_rate}\n")
    print(output, end="")

    f.write(output)



    # Plot training and test loss and accuracy
    def save_plot(filename, train_losses, test_losses,
        train_accuracies, test_accuracies):
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label='Training')
    plt.plot(test_losses, label='Test')
    plt.title('Loss')
    plt.xlabel('Epoch')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(train_accuracies, label='Training')
    plt.plot(test_accuracies, label='Test')
    plt.title('Accuracy');
    plt.xlabel('Epoch')
    plt.legend()
    plt.savefig(filename)
    plt.show()

    save_plot(plot_filename, train_losses, test_losses,
        train_accuracies, test_accuracies)
```

Α.2 Tensorflow-CNN

Code Listing Α.10: CNN

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D,
    Flatten, Dense, Dropout
from tensorflow.keras.optimizers import SGD
import numpy as np
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import mnist
```

```
import matplotlib.pyplot as plt
# Set data limits and filenames
import os

# Define the path to the directory where the model will be
    saved
model_dir = 'saved_models'
os.makedirs(model_dir, exist_ok=True)

# Save the model inside the directory

data_limit = 4000
plot_filename = "tensorflow_lr_moment_decay.png"
# model_output = "model.txt"

# Define a function to preprocess the data
def preprocess_data(x, y):
x_new = []
y_new = []
for i in range(10):
indices = np.where(y == i)[0][:data_limit]
np.random.shuffle(indices)  # Shuffle the indices before
    selecting data
x_new.append(x[indices])
y_new.append(np.full(len(indices), i))
x_new = np.concatenate(x_new)
y_new = np.concatenate(y_new)
x_new = x_new.reshape(len(x_new), 28, 28, 1)
x_new = x_new.astype("float32") / 255
y_new = to_categorical(y_new, num_classes=10)
indices = np.random.permutation(len(x_new))  # Shuffle the
    data
return x_new[indices], y_new[indices]

# Load the MNIST dataset and preprocess it
(x_train, y_train), (x_test, y_test) = mnist.load_data()
X_train, y_train = preprocess_data(x_train, y_train)
X_test, y_test = preprocess_data(x_test, y_test)

# Define the neural network architecture
model = tf.keras.models.Sequential([
tf.keras.layers.Conv2D(64, (3, 3), activation='relu',
    input_shape=(28, 28, 1)),
tf.keras.layers.Flatten(),
tf.keras.layers.Dense(128, activation='relu'),
tf.keras.layers.Dropout(0.2),
# tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dense(10, activation='softmax'),
tf.keras.layers.Dropout(0.2)
])
```

```
opt = tf.keras.optimizers.legacy.SGD(learning_rate=10e-5)

# opt = SGD(learning_rate=10e-5)
model.compile(loss='categorical_crossentropy', optimizer=
    opt, metrics=['accuracy'])
# Train the model
history = model.fit(X_train, y_train, validation_data=(
    X_test, y_test), epochs=100,batch_size=64)

# Save the model and plot the training history
model.save(os.path.join(model_dir, 'tensor.h5'))
plt.plot(history.history['accuracy'], label='training␣
    accuracy')
plt.plot(history.history['val_accuracy'], label='validation
    ␣accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.savefig(plot_filename)
```