Πρώτη Εργασία Νευρωνικά δίκτυα

Γρηγόρης Παντζαρτζής AEM:3785

Στην εργασία χρησιμοποιήθηκε το Dataset της Mnist

```
import torch
import torchvision
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as transforms
import numpy as np
import torch.nn as nn
import math



# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# parameters
batch_size = 100
input_size = 784
hidden_size = 500
num_classes = 10
num_epochs = 5
batch_size = 100
learning_rate = 0.001
```

Αρχικά κάνουμε import της κατάλληλες βιβλιοθήκες που θα μας βοηθήσουν στην υλοποίηση της άσκησης

Φτιάχνουμε μια συσκευή με την βοήθεια της torch και την δίνουμε την δυνατότητα να τρέξει τα δεδομένα στην gpu αν είναι διαθέσιμος αλλιώς στον cpu

Έπειτα δίνουμε παραμέτρους για να την λύση της ασκήσεις

Input_size=784 γιατί οι εικόνες έχουν το size 28x28

Num_classes=10 γιατί έχουμε 10 διαφορετικές κλασεις από το 0 εώς το 9

Τα υπόλοιπα μπορούμε να τα αλλάξουμε τιμές ώστε να κάνουμε διάφορα τεστ

Σαν Batch_Size ορίζουμε τον αριθμό τον samples που γίνονται training και σε ένα forward και backward pass

Σαν Epoch ορίζουμε 1 forward και backward pass για όλα τα training samples

Σαν hidden size ορίζουμε το hidden size που έχει το hidden layer του νευρωνικου δικτύου

```
train_dataset = torchvision.datasets.MNIST(root='../../data', train=True,download=True, transform=transforms.ToTensor())


test_dataset = torchvision.datasets.MNIST(root='../../data', train=False, transform=transforms.ToTensor())

train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)
x,y = train_dataset[0]
```

Κάνουμε import το Dataset της mnist με την βοήθεια της βιβλιοθήκης torch train =true γιατί αυτό είναι το dataset που θα γίνει train download=true για να κατέβει αν δεν είναι ήδη διαθέσιμη και μετα κάνουμε convert το dataset σε tesnor

Στην συνέχεια φτιάχνουμε τους dataloaders που έχουν σαν ορίσματα το dataset που θέλει το καθένα το batch size και στην περίπτωση του train_loader βάζουμε shuffle =True που ¨ανακατεύει¨ τα δεδομένα και αυτό μας βοηθάει στο training.Στο test_loader δεν έχει κάποια σημασία να το βάλουμε true οπότε το βάζουμε στο false.

```
#Neural Network -- one hidden layer
class NeuralNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out

model = NeuralNetwork(input_size, hidden_size, num_classes).to(device)
```

Στην συνέχεια φτιάχνουμε ένα νευρωνικο δίκτυο με ένα hidden layer.

Προσδιορίζουμε τα layers μας. Έχουμε ένα input linear layer(οπου αποτελείται από 784 νευρώνες αφού οι εικόνες μας εχουν size 28x28 και έτσι κάθε ένα από τα pixel θα γίνει inputed σε έναν νευρώνα).Ενδιάμεσα έχουμε μια συνάρτηση ενεργοποίησης στο συγκεκριμένο νευρωνικό την RELU (οι συναρτήσεις ενεργοποιήσεις εφαρμόζουν ένα non-linear transformation και έτσι αποφασίζουν έναν ένας νευρώνας πρέπει να ενεργοποιηθεί η όχι. Χρειαζόμαστε non-linear transformation έτσι ώστε το δίκτυο μας να μαθαίνει καλύτερά και να μπορεί να διαχειριστεί πιο περίπλοκα προβλήματα) και τέλος ακόμη ένα output linear layer

```
#Set Loss
criterion=nn.CrossEntropyLoss()

#Optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

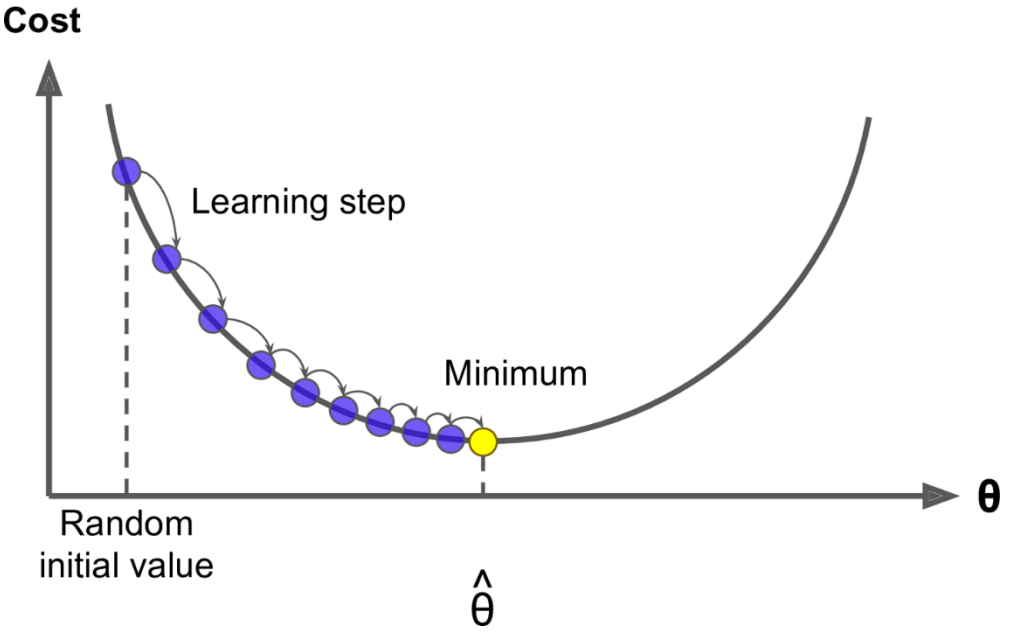
Έπειτα κάνουμε set την συνάρτηση του loss που θα μας βοηθήσει να υπολογίσουμε το loss.Στην συγκεκριμένη περίπτωση η συνάρτηση του cross-entropy

$$D(\hat{Y}, Y) = -\frac{1}{N} \cdot \sum Y_i \cdot \log(\hat{Y}_i)$$

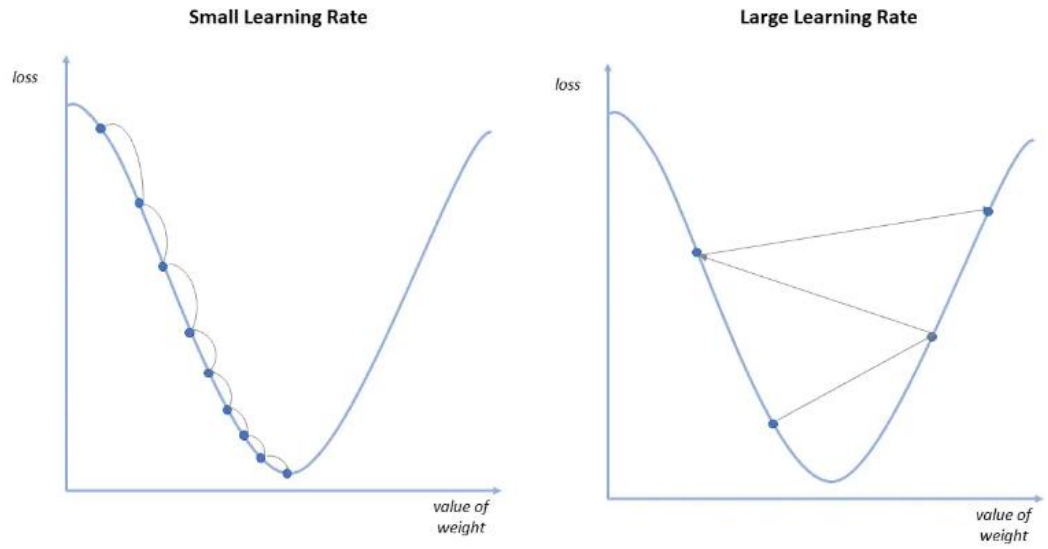
Όσο καλύτερο prediction τόσο χαμηλότερο loss θα έχουμε.

Η συνάρτηση nn.CrossEntropyLoss εφαρμόζει LogSoftmax και Negative log likehood loss.

Ο optimizer μας δείχνει πως να update τα βαροι



Ο gradient descent υπολογίζει το slope έτσι ώστε να ξέρουμε προς τα που να κατευθυνθούμε μέχρι να βρούμε την μικρότερη τιμή. Αυτό γίνεται για όλες τις παραμέτρους ταυτόχρονα. Ποσά steps θα κάνουμε εξαρτώνται από το learning rate που δίνουμε. Δεν θέλουμε να βάλουμε ούτε πολύ μεγάλο learning rate ούτε πολύ μικρό.



Με ένα μικρό learning rate μπορεί να μας πάρει πολύ ώρα ώστε να φτιάσουμε στο min value.Αν βάλουμε μεγάλο learning rate μπορεί να κάνουμε overshooting το minimum value και να μην φτάσουμε σε αυτό.

```
total_step = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):

        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)


        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, num_epochs, i+1, total_step, loss.item()))
```

Μετα κάνουμε train το μοντέλο μας. Καθορίζουμε στα συνολικά βήματα που θα χρειαστούν. Κάνουμε loop over the epochs.Κάνουμε μια λούπα με όλα τα batches.Η enumerate θα μας δώσει το πραγματικό index και

τα δεδομένα. Μετα κάνουμε reshape τις εικόνες μας και μετα τα βάζουμε στην gpu αν είναι available αλλιώς στην cpu.Βάζουμε τα labels στο device

Μετα κάνουμε ένα forward pass.Βάζουμε στο output τα images και μετα υπολογίζουμε το loss

Μετα κάνουμε ένα backend pass.καλούμε το optimizer.zero_grad() ώστε να αδειάσει τις τιμές στην gradient attribute.Με το loss.backward() κάνουμε back-propagation.Καλούμε το optimizer.step ώστε να κάνει update τις παραμέτρους μας.Μετα με την print δείχνουμε το loss ανάλογα με τις εποχές και τα βήματα που έχουν γίνει.

```
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Accuracy of the network on the 10000 test images: {} %'.format(100 * correct / total))
```

Μετα φτιάχνουμε το testing.Δεν θέλουμε να υπολογίσουμε το gradient για όλα τα βήματα που κάνουμε οπότε χρησιμοποιούμε torch.no_grad().Στην αρχή αρχικοποιηουμε τις σωστές προβλέψεις με 0 όπως και τον αριθμο τον samples.Φτιάχνουμε μια λούπα οπού κάνουμε πάλι reshape τα images και τα βάζουμε στην συσκευή όπως και τα labels.Μετα υπολογίζουμε τα predictions.Στην συνέχεια περνούμε τα πραγματικά predictions.Μετα περνούμε τον αριθμό τον κάθε samples στο current batch.Και τέλος υπολογίζουμε τον αριθμό τον σωστόν predictions,και με μια print εμφανίζουμε το accuracy.

Testing με το learning rate

Με testing rate 0.001:

```
Epoch [1/5], Step [100/600], Loss: 0.4193
Epoch [1/5], Step [200/600], Loss: 0.3791
Epoch [1/5], Step [300/600], Loss: 0.1391
Epoch [1/5], Step [400/600], Loss: 0.2251
Epoch [1/5], Step [500/600], Loss: 0.1202
Epoch [1/5], Step [600/600], Loss: 0.1764
Epoch [2/5], Step [100/600], Loss: 0.1077
Epoch [2/5], Step [200/600], Loss: 0.0881
Epoch [2/5], Step [300/600], Loss: 0.0815
Epoch [2/5], Step [400/600], Loss: 0.1080
Epoch [2/5], Step [500/600], Loss: 0.2030
Epoch [2/5], Step [600/600], Loss: 0.0521
Epoch [3/5], Step [100/600], Loss: 0.0756
Epoch [3/5], Step [200/600], Loss: 0.1708
Epoch [3/5], Step [300/600], Loss: 0.0974
Epoch [3/5], Step [400/600], Loss: 0.0463
Epoch [3/5], Step [500/600], Loss: 0.1377
Epoch [3/5], Step [600/600], Loss: 0.0133
Epoch [4/5], Step [100/600], Loss: 0.0277
Epoch [4/5], Step [200/600], Loss: 0.0281
Epoch [4/5], Step [300/600], Loss: 0.0703
Epoch [4/5], Step [400/600], Loss: 0.0661
Epoch [4/5], Step [500/600], Loss: 0.0361
Epoch [4/5], Step [600/600], Loss: 0.0454
Epoch [5/5], Step [100/600], Loss: 0.0157
Epoch [5/5], Step [200/600], Loss: 0.0229
Epoch [5/5], Step [300/600], Loss: 0.0456
Epoch [5/5], Step [400/600], Loss: 0.0345
Epoch [5/5], Step [500/600], Loss: 0.0534
Epoch [5/5], Step [600/600], Loss: 0.0142
Time Passed: 38.859sec
Accuracy of the network on the 10000 test images: 98.09 %
```

Με testing rate 0.1:

```
Epoch [1/5], Step [100/600], Loss: 0.8065
Epoch [1/5], Step [200/600], Loss: 0.5622
Epoch [1/5], Step [300/600], Loss: 0.5751
Epoch [1/5], Step [400/600], Loss: 0.3154
Epoch [1/5], Step [500/600], Loss: 0.6503
Epoch [1/5], Step [600/600], Loss: 0.8487
Epoch [2/5], Step [100/600], Loss: 0.3842
Epoch [2/5], Step [200/600], Loss: 0.7941
Epoch [2/5], Step [300/600], Loss: 0.6183
Epoch [2/5], Step [400/600], Loss: 0.4554
Epoch [2/5], Step [500/600], Loss: 0.4823
Epoch [2/5], Step [600/600], Loss: 0.4330
Epoch [3/5], Step [100/600], Loss: 0.6821
Epoch [3/5], Step [200/600], Loss: 0.4452
Epoch [3/5], Step [300/600], Loss: 0.3973
Epoch [3/5], Step [400/600], Loss: 0.4948
Epoch [3/5], Step [500/600], Loss: 1.1476
Epoch [3/5], Step [600/600], Loss: 0.4788
Epoch [4/5], Step [100/600], Loss: 0.3040
Epoch [4/5], Step [200/600], Loss: 0.4941
Epoch [4/5], Step [300/600], Loss: 0.5429
Epoch [4/5], Step [400/600], Loss: 0.9941
Epoch [4/5], Step [500/600], Loss: 0.4897
Epoch [4/5], Step [600/600], Loss: 0.4161
Epoch [5/5], Step [100/600], Loss: 1.1298
Epoch [5/5], Step [200/600], Loss: 0.9119
Epoch [5/5], Step [300/600], Loss: 0.6397
Epoch [5/5], Step [400/600], Loss: 0.5189
Epoch [5/5], Step [500/600], Loss: 0.4926
Epoch [5/5], Step [600/600], Loss: 1.2673
Time Passed: 42.119sec
Accuracy of the network on the 10000 test images: 86.94 %
```

Mε testing rate 1:

```
Epoch [1/5], Step [100/600], Loss: 2.0057
Epoch [1/5], Step [200/600], Loss: 1.7124
Epoch [1/5], Step [300/600], Loss: 1.9587
Epoch [1/5], Step [400/600], Loss: 2.0207
Epoch [1/5], Step [500/600], Loss: 1.9949
Epoch [1/5], Step [600/600], Loss: 2.1028
Epoch [2/5], Step [100/600], Loss: 2.0361
Epoch [2/5], Step [200/600], Loss: 2.2268
Epoch [2/5], Step [300/600], Loss: 1.9992
Epoch [2/5], Step [400/600], Loss: 2.3441
Epoch [2/5], Step [500/600], Loss: 2.3519
Epoch [2/5], Step [600/600], Loss: 2.3260
Epoch [3/5], Step [100/600], Loss: 2.4409
Epoch [3/5], Step [200/600], Loss: 2.4316
Epoch [3/5], Step [300/600], Loss: 2.2840
Epoch [3/5], Step [400/600], Loss: 2.2794
Epoch [3/5], Step [500/600], Loss: 2.3166
Epoch [3/5], Step [600/600], Loss: 2.4113
Epoch [4/5], Step [100/600], Loss: 2.3455
Epoch [4/5], Step [200/600], Loss: 2.3886
Epoch [4/5], Step [300/600], Loss: 2.3204
Epoch [4/5], Step [400/600], Loss: 2.3644
Epoch [4/5], Step [500/600], Loss: 2.2931
Epoch [4/5], Step [600/600], Loss: 2.3221
Epoch [5/5], Step [100/600], Loss: 2.3204
Epoch [5/5], Step [200/600], Loss: 2.4128
Epoch [5/5], Step [300/600], Loss: 2.3816
Epoch [5/5], Step [400/600], Loss: 2.4243
Epoch [5/5], Step [500/600], Loss: 2.3984
Epoch [5/5], Step [600/600], Loss: 2.3323
Time Passed: 43.923sec
Accuracy of the network on the 10000 test images: 11.35 %
```

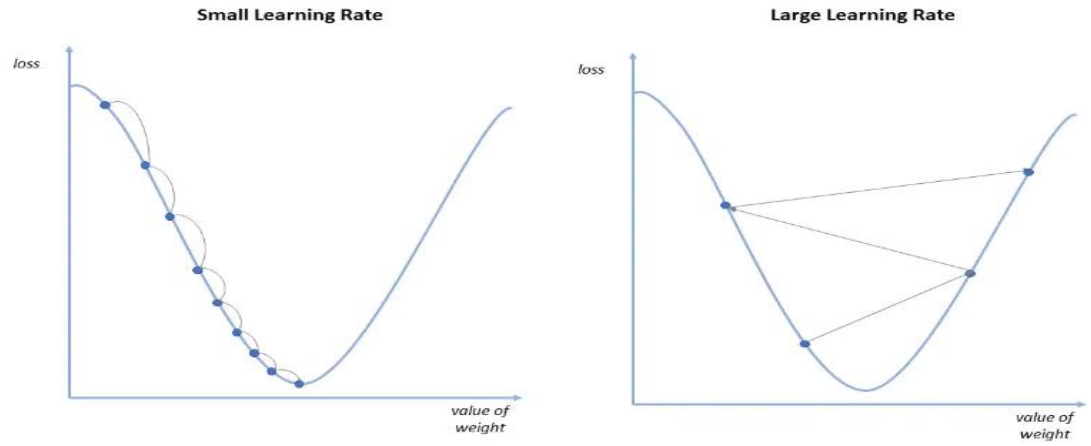
Με testing rate 0.0001:

```
Epoch [1/5], Step [100/600], Loss: 1.2938
Epoch [1/5], Step [200/600], Loss: 0.6879
Epoch [1/5], Step [300/600], Loss: 0.4899
Epoch [1/5], Step [400/600], Loss: 0.3753
Epoch [1/5], Step [500/600], Loss: 0.3586
Epoch [1/5], Step [600/600], Loss: 0.2981
Epoch [2/5], Step [100/600], Loss: 0.2696
Epoch [2/5], Step [200/600], Loss: 0.3000
Epoch [2/5], Step [300/600], Loss: 0.3104
Epoch [2/5], Step [400/600], Loss: 0.2502
Epoch [2/5], Step [500/600], Loss: 0.2692
Epoch [2/5], Step [600/600], Loss: 0.1815
Epoch [3/5], Step [100/600], Loss: 0.2647
Epoch [3/5], Step [200/600], Loss: 0.1906
Epoch [3/5], Step [300/600], Loss: 0.2537
Epoch [3/5], Step [400/600], Loss: 0.2423
Epoch [3/5], Step [500/600], Loss: 0.3129
Epoch [3/5], Step [600/600], Loss: 0.1884
Epoch [4/5], Step [100/600], Loss: 0.2618
Epoch [4/5], Step [200/600], Loss: 0.2324
Epoch [4/5], Step [300/600], Loss: 0.1082
Epoch [4/5], Step [400/600], Loss: 0.1588
Epoch [4/5], Step [500/600], Loss: 0.2448
Epoch [4/5], Step [600/600], Loss: 0.2738
Epoch [5/5], Step [100/600], Loss: 0.1940
Epoch [5/5], Step [200/600], Loss: 0.1907
Epoch [5/5], Step [300/600], Loss: 0.1810
Epoch [5/5], Step [400/600], Loss: 0.2491
Epoch [5/5], Step [500/600], Loss: 0.2316
Epoch [5/5], Step [600/600], Loss: 0.1516
Time Passed: 36.977sec
Accuracy of the network on the 10000 test images: 94.59 %
```

Με testing rate 0.00001:

```
Epoch [1/5], Step [100/600], Loss: 2.2927
Epoch [1/5], Step [200/600], Loss: 2.2685
Epoch [1/5], Step [300/600], Loss: 2.2825
Epoch [1/5], Step [400/600], Loss: 2.2639
Epoch [1/5], Step [500/600], Loss: 2.2489
Epoch [1/5], Step [600/600], Loss: 2.2504
Epoch [2/5], Step [100/600], Loss: 2.2183
Epoch [2/5], Step [200/600], Loss: 2.2396
Epoch [2/5], Step [300/600], Loss: 2.2192
Epoch [2/5], Step [400/600], Loss: 2.1887
Epoch [2/5], Step [500/600], Loss: 2.1938
Epoch [2/5], Step [600/600], Loss: 2.1666
Epoch [3/5], Step [100/600], Loss: 2.1812
Epoch [3/5], Step [200/600], Loss: 2.1556
Epoch [3/5], Step [300/600], Loss: 2.1709
Epoch [3/5], Step [400/600], Loss: 2.1285
Epoch [3/5], Step [500/600], Loss: 2.1072
Epoch [3/5], Step [600/600], Loss: 2.1356
Epoch [4/5], Step [100/600], Loss: 2.0928
Epoch [4/5], Step [200/600], Loss: 2.0826
Epoch [4/5], Step [300/600], Loss: 2.0948
Epoch [4/5], Step [400/600], Loss: 2.0493
Epoch [4/5], Step [500/600], Loss: 2.0485
Epoch [4/5], Step [600/600], Loss: 2.0171
Epoch [5/5], Step [100/600], Loss: 2.0327
Epoch [5/5], Step [200/600], Loss: 2.0231
Epoch [5/5], Step [300/600], Loss: 2.0235
Epoch [5/5], Step [400/600], Loss: 2.0338
Epoch [5/5], Step [500/600], Loss: 1.9899
Epoch [5/5], Step [600/600], Loss: 1.9556
Time Passed: 40.397sec
Accuracy of the network on the 10000 test images: 72.02 %
>>>
```

Συμπέρασμα: Δεν μας συμφέρει να δίνουμε ούτε πολύ μεγάλο learning rate ούτε πολύ μικρό. Αν έχουμε πολύ μεγάλο learning rate μειώνεται το accuracy και αυξάνεται το loss.Αν έχουμε πολύ μικρό learning rate το loss μειώνεται πολύ αργά και το μειώνεται το accuracy.



Testing με το Batch Size

Batch size 1:

```
Epoch [5/5], Step [60000/60000], Loss: 0.0000
Time Passed: 1421.360sec
Accuracy of the network on the 10000 test images: 97.46 %
```

Batch size 10:

```
Epoch [5/5], Step [6000/6000], Loss: 0.0001
Time Passed: 138.987sec
Accuracy of the network on the 10000 test images: 97.59 %
```

Batch size 50:

```
Epoch [5/5], Step [1200/1200], Loss: 0.0069
Time Passed: 50.026sec
Accuracy of the network on the 10000 test images: 97.84 %
```

Batch size 100:

```
Epoch [5/5], Step [600/600], Loss: 0.0860
Time Passed: 39.045sec
Accuracy of the network on the 10000 test images: 97.94 %
```

Batch size 1000:

```
Time Passed: 29.131sec
Accuracy of the network on the 10000 test images: 96.19 %
```

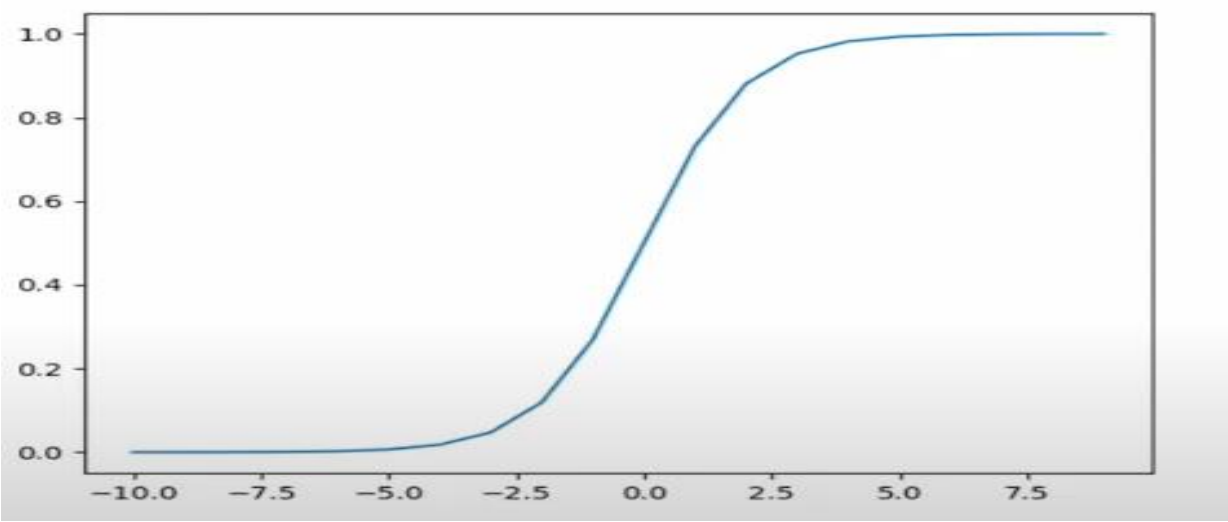
Batch size 10000:

```
Time Passed: 27.732sec
Accuracy of the network on the 10000 test images: 89.06 %
```

Συμπέρασμα: Το καλύτερο batch size είναι κάπου ανάμερα στο 100.Τα μικρά batch size περνούν πολύ ώρα να εκτελεστούν Και τα μεγάλα batch size Παρόλου που εκτελούνται γρηγορά το accuracy πέφτει όσο το batch μεγαλώνει.

Testing με το Activation Function

Sigmoid activation function:



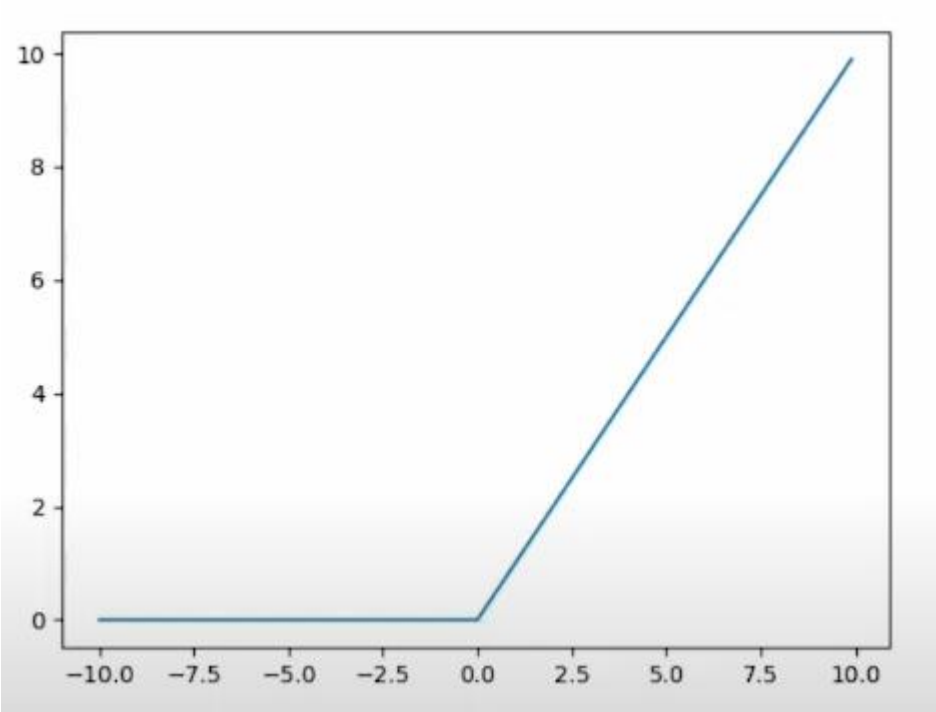
$$f(x) = \frac{1}{1 + e^{-x}}$$

H sigmoid μας δίνει ένα probability ανάμεσα στο 0 και 1

Εφαρμόζοντας την Sigmoid() στον νευρωσικό έχω τα εξής αποτελέσματα:

```
Time Passed: 35.528sec
Accuracy of the network on the 10000 test images: 96.71 %
>>>
```

Relu Activation function:

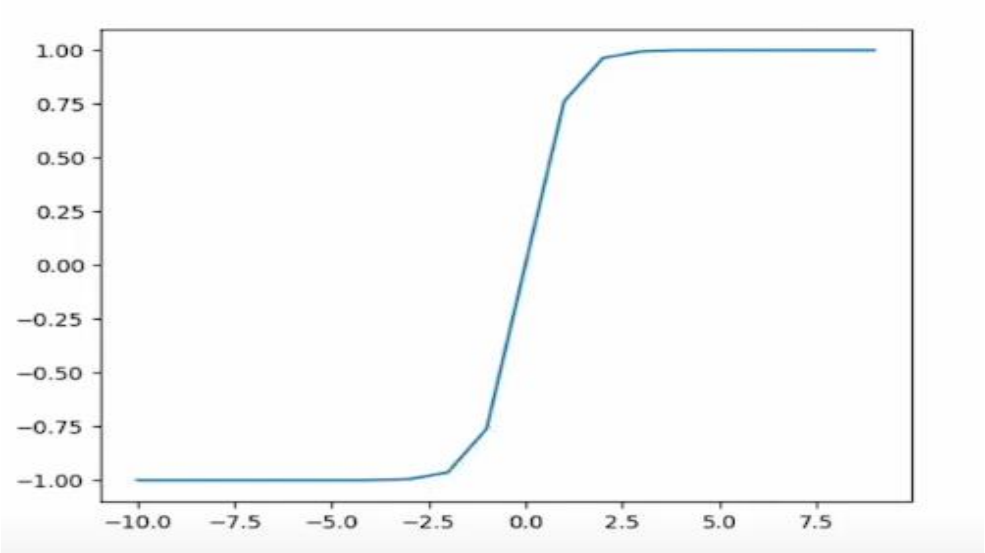


Δίνει output 0 για αρνητικές τιμές. Και για θετικές τιμές δίνει output linear function.

Εφαρμόζοντας Relu() περνούμε τα εξής αποτελέσματα:

```
Time Passed: 38.579sec
Accuracy of the network on the 10000 test images: 97.93 %
```

Tahn Activation function:



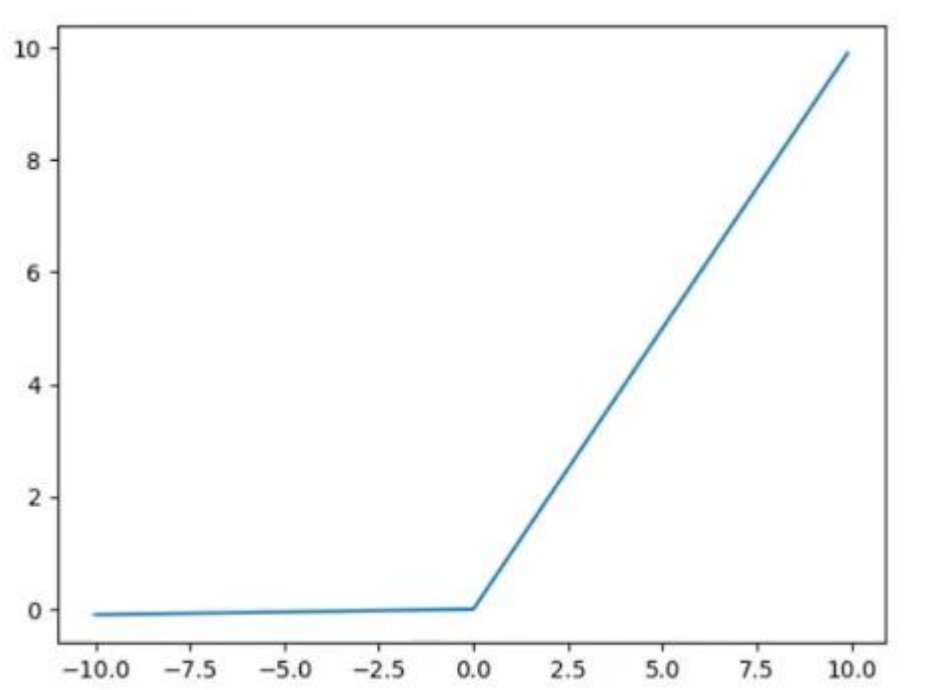
$$f(x) = \frac{2}{1 + e^{-2x}} - 1$$

Είναι scaled sigmoid και λίγο shifted .θα δώσει τιμές από -1 εως 1

Εφαρμόζοντας Tahn() περνούμε τα εξής αποτελέσματα:

```
Time Passed: 34.973sec
Accuracy of the network on the 10000 test images: 97.57 %
```

LeakyRelu Activation function:



Είναι ελαφρώς βελτιωμένη από την relu.Βοηθάει για να λυθεί το vanish gradient problem.

Εφαρμόζοντας LeakyRelu () περνούμε τα εξής αποτελέσματα:

```
Time Passed: 38.272sec
Accuracy of the network on the 10000 test images: 97.87 %
```

Testing με περισσότερα από ένα hidden layers

```
class NeuralNetwork2(nn.Module):
    def __init__(self, input_size, num_classes):
        super(NeuralNetwork2, self).__init__()
        self.fC1 = nn.Linear(input_size, 600)
        self.relu1 = nn.ReLU()
        self.fC2 = nn.Linear(600, 300)
        self.relu2 = nn.ReLU()
        self.fC3 = nn.Linear(300, num_classes)

    def forward(self, x):
        out = self.fC1(x)
        out = self.relu1(out)
        out = self.fC2(out)
        out = self.relu2(out)
        out = self.fC3(out)
        return out


model2 = NeuralNetwork2(input_size, num_classes).to(device)
```

Παρατηρούμε ότι:

```
Time Passed: 43.308sec
Accuracy of the network on the 10000 test images: 97.72 %
```

Έχει ίδιο σχεδόν accuracy με το ένα hidden layer χρησιμοποιώντας relu απλά εκτελούνταν για περισσότερο χρόνο

```
class NeuralNetwork3(nn.Module):
    def __init__(self, input_size, num_classes):
        super(NeuralNetwork3, self).__init__()
        self.fC1 = nn.Linear(input_size, 4096)
        self.relu1 = nn.ReLU()
        self.fC2 = nn.Linear(4096, 4096)
        self.relu2 = nn.ReLU()
        self.fC3 = nn.Linear(4096, 512)
        self.relu3 = nn.ReLU()
        self.fC4 = nn.Linear(512, 256)
        self.relu4 = nn.ReLU()
        self.fC5 = nn.Linear(256, num_classes)

    def forward(self, x):
        out = self.fC1(x)
        out = self.relu1(out)
        out = self.fC2(out)
        out = self.relu2(out)
        out = self.fC3(out)
        out = self.relu3(out)
        out = self.fC4(out)
        out = self.relu4(out)
        out = self.fC5(out)
        return out

model3 = NeuralNetwork3(input_size, num_classes).to(device)
```

Παρατηρούμε ότι:

```
Time Passed: 614.126sec
Accuracy of the network on the 10000 test images: 97.38 %
```

Το νευρωσικό μας κάνει παραπάνω ώρα και το accuracy έχει ελαφρώς πέσει.

Testing στους optimizers

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Παρατηρούμε ότι:

```
Time Passed: 33.461sec
Accuracy of the network on the 10000 test images: 77.71 %
```

Το accuracy έχει πέσει σε σχέση με το Adam

Με Adam:

```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Παρατηρούμε ότι:

```
Time Passed: 38.884sec
Accuracy of the network on the 10000 test images: 98.03 %
```

Το accuracy έχει αυξηθεί

Με Adagrad:

```
optimizer = torch.optim.Adagrad(model.parameters(), lr=learning_rate)
```

Παρατηρούμε ότι:

```
Time Passed: 34.457sec
Accuracy of the network on the 10000 test images: 92.21 %
```

Το accuracy έχει πέσει σχετικά με το adam αλλά είναι πιο πάνω από το sgd.

```
from sklearn.neighbors import NearestNeighbors
import numpy as np
import math
from sklearn import neighbors
from sklearn.neighbors import NearestCentroid

TrainMnist = np.loadtxt('mnist_train.csv',delimiter=",", dtype=np.float32, skiprows=1)
TestMnist = np.loadtxt('mnist_test.csv',delimiter=",", dtype=np.float32, skiprows=1)

labels = TrainMnist[:,0:1]
TrainMnist = TrainMnist[:,1:]

labelst = TestMnist[:,0:1]
TestMnist = TestMnist[:,1:]

y_train=np.ravel(labels)
x_train=np.array(TrainMnist)

y_test=np.ravel(labelst)
x_test=np.array(TestMnist)


kmn = neighbors.KNeighborsClassifier(1).fit(x_train,y_train)

print("Accuracy of 1-Neighbors:")
print(kmn.score(x_test,y_test))

kmn = neighbors.KNeighborsClassifier(3).fit(x_train,y_train)

print("Accuracy of 3-Neighbors:")
print(kmn.score(x_test,y_test))

Cen = NearestCentroid()
Cen.fit(x_train, y_train)


print("Accuracy of Centroid:")
print(Cen.score(x_test, y_test))
```

Παρατηρούμε ότι:

```
Accuracy of 1-Neighbors:
0.9691
Accuracy of 3-Neighbors:
0.9705
Accuracy of Centroid:
0.8203
```

Συμπέρασμα: Βλέπουμε ότι στο νευρωνικο δίκτυο μας η ακρίβεια είναι μεγαλύτερη από τους αλγορίθμους του κοντινότερου γείτονα και centroid