Neural networks coursework

Firstly, we start off by initialising our hyperparameter batch_size which chooses how much data is used per training cycle.

```
import matplotlib.pyplot as plt
#We will set the batch_size here to be 64
batch_size = 64
#This is where we load the data from the mnist dataset
train_dataset, test_dataset =mu.load_data_fashion_mnist(batch_size)
#This is to see what one of the images look like from the dataset
```

We can also import the training data and test data from the MNIST dataset which will be used later.

The stem:

For the stem all we must do is split an image into patches which can then be used in the neural network.

```
x = nn.functional.unfold(x,(14,14),stride =14)
```

The backbone:

```
def forward(self, x):
    x = nn.functional.unfold(x,(14,14),stride =14)

    #This is the first mlp where we do O1 =g(xTw1)W2
    #First we transpose x so that it is in the correct shape for the lin
    x = torch.transpose(x , 1 , 2)
    #Then we do the linear layer
    x = self.Linear_1(x)
    #Then we do the relu
    x = self.relu(x)
    #Then we do the linear layer
    x = self.Linear_2_hidden(x)
    # dropout layer
    #torch.nn.Dropout(p = 0.1 ,  inplace = True)
    #step 2
    #This is the second mlp where we do O2 =g(01w3)W4
    x = torch.transpose(x ,1 ,2)

    #second mlp
    #Then we do the linear layer
    x = self.Linear_3_hidden(x)
    #Then we do the relu
    x = self.relu(x)
    #Then we do the linear layer
    x = self.Linear_4_hidden(x)
    #Then we do the softmax for output
    output = x
    return output.mean(axis=1)
```

Here we implement the two MLPs shown in the slides

The first MLP is: $O_1 = g(X^T W_1)W_2$, w

- Next step is $O_1 \leftarrow O_1^T$.

The second MLP is: $O_2 = g(O_1 W_3)W_4$

And we initialise the layers with

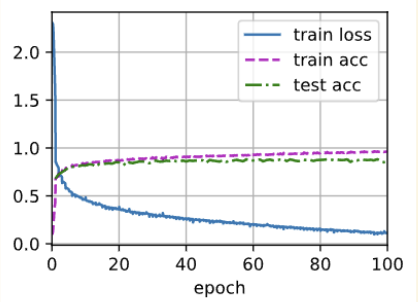
```
    super(Net, self).__init__()
    self.num_inputs = num_inputs
    self.num_outputs = num_outputs
    self.relu = nn.ReLU(inplace =True)
    self.Linear_1 = nn.Linear( 196 ,784 )
    self.Linear_2_hidden = nn.Linear(784, 200)
    self.Linear_3_hidden = nn.Linear(4 , 64)
    self.Linear_4_hidden = nn.Linear(64, 10)
```

Also, we can change the model to run on CH6 which uses the GPU instead of the CPU which is much quicker

```
    # Train the network
    # This is the number of epochs we will train the network for
    num_epochs = 100
    initialtime = time.perf_counter() #This is to keep track of the time
    mu.train_ch6(netWcuda, train_dataset, test_dataset,num_epochs , 0.1)
    posttime = time.perf_counter()
    print('Time taken: {}'.format(posttime - initialtime)) #This is to pr
    mu.evaluate_accuracy_gpu(netWcuda, test_dataset) # This is to evaluat

✓ 12m 1.1s
```

Best accuracy was 0.8603

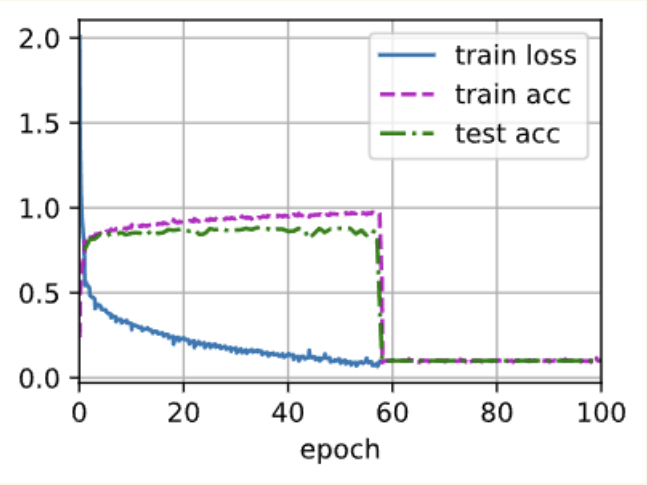
```
loss 0.113, train acc 0.960, test acc 0.860
21217.8 examples/sec on cuda:0
Time taken: 719.6274362

0.8603
```



This was done using SGD and a learning rate of 0.1 with epoch at 100. Epoch at 100 was only possible since we used the GPU and doesn't take that long.

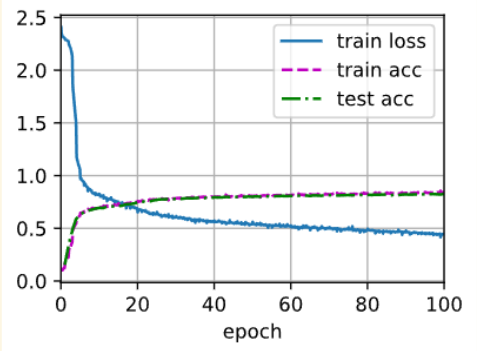
Increasing the learning rate would skewer the results. When the learning rate was increased to 0.9 we get this :

```
loss nan, train acc 0.100, test acc 0.100
20473.1 examples/sec on cuda:0
Time taken: 728.1154689

0.1
```



When lowering the Learning rate we would see :

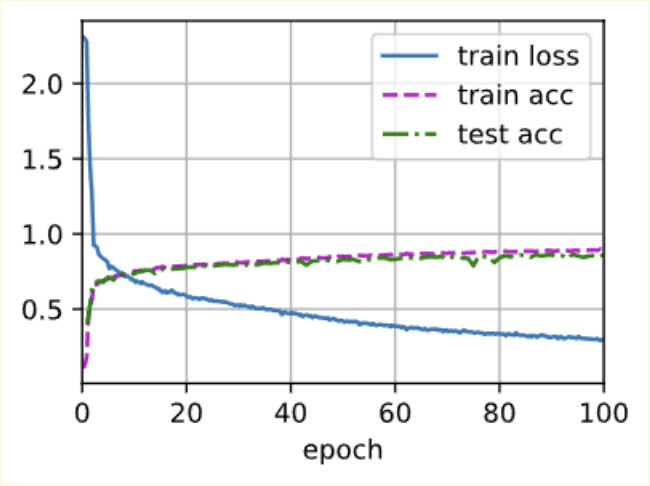
```
loss 0.446, train acc 0.840, test acc 0.826
18291.8 examples/sec on cuda:0
Time taken: 786.5817491999996

0.8256
```



Which shows us that training loss increases when learning rate is decreased.

Test and training accuracy is high until around 58 epochs where it falls drastically. The batch size was kept at 64. When increasing the batch size we would see:

```
loss 0.300, train acc 0.893, test acc 0.862
33791.3 examples/sec on cuda:0
Time taken: 480.7625185999998

0.862
```



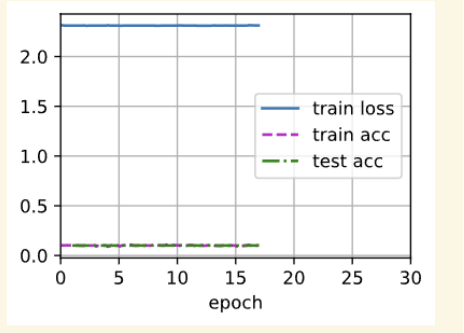
Where the batch size was 256, the loss was higher from 0.113 to 0.3 however the training accuracy and test accuracy would decrease

When try weight decay and momentum with SGD we get the following results

learning rate	weight decay	epochs	momentum	accuracy
0.1	NA	100	0.9	0.85
0.15	na	100	0.95	0.8566
0.2	n/a	100	0.99	0.8451
0.2	0.01	100	0.99	0.8595
0.2	0.05	100	0.99	0.8595
0.2	0.09	100	0.99	0.8551
0.5	0.09	100	0.99	0.8601
0.9	0.09	100	0.99	0.861

However, removing momentum and weight decay would yield a higher accuracy.

When trying to train with Adam on the GPU , there were errors such as :



Overall my method of getting the highest accuracy was to use SGD with a learning rate of 0.1 and epoch of 100 without momentum or weight decay.