Code:

(Parts highlighted were edited)

import torch

from torch import nn

from torch.utils.data import DataLoader

from torchvision import datasets

from torchvision.transforms import ToTensor

import matplotlib.pyplot as plt

import numpy as np

import random

#check if gpu is available

use\_cuda = torch.cuda.is\_available()

device = torch.device("cuda" if use\_cuda else "cpu")

print ('<== CUDA availability : ==>', use\_cuda)

#set seed for reproducability

def set\_seeds(seed):

    torch.backends.cudnn.benchmark = False

    torch.backends.cudnn.determinictic = True

    torch.cuda.manual\_seed\_all(seed)

    np.random.seed(seed)

    torch.manual\_seed(seed)

    random.seed(seed)

set\_seeds(42)

training\_data = datasets.MNIST(

    root="data",

    train=True,

    download=True,

    transform=ToTensor(),

)

# Download test data from open datasets.

test\_data = datasets.MNIST(

    root="data",

    train=False,

    download=True,

    transform=ToTensor(),

)

#Visualise some of the input images.

labels\_map = {

    0: "zero",

    1: "one",

    2: "two",

    3: "three",

    4: "four",

    5: "five",

    6: "six",

    7: "seven",

    8: "eight",

    9: "nine",

}

figure = plt.figure(figsize=(8, 8))

cols, rows = 3, 3

for i in range(1, cols \* rows + 1):

    sample\_idx = torch.randint(len(training\_data), size=(1,)).item()  # randomly pick indices from the training data

    img, label = training\_data[sample\_idx]                            # Read the images using their indices.

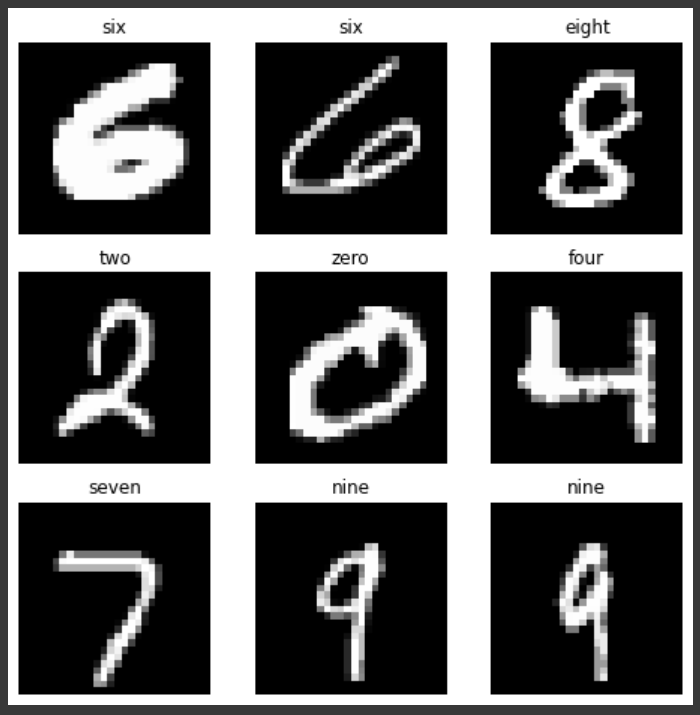
    figure.add\_subplot(rows, cols, i)

    plt.title(labels\_map[label])

    plt.axis("off")

    plt.imshow(img.squeeze(), cmap="gray")

plt.show()



The labels below were used for Fashion MINST instead of the ones in the original code.

####################################

labels\_map = {

    0: "T-shirt/top",

    1: "Trouser",

    2: "Pullover",

    3: "Dress",

    4: "Coat",

    5: "Sandal",

    6: "Shirt",

    7: "Sneaker",

    8: "Bag",

    9: "Ankle boot",

}

A picture containing text, screenshot

Description automatically generated

####################################

#Check the size and shape of training and validation data.

print ('Shape of Training Data', np.shape(training\_data))

print ('Shape of Test Data', np.shape(test\_data))

batch\_size = 64

# Create data loaders.

train\_dataloader = DataLoader(training\_data, batch\_size=batch\_size)

test\_dataloader = DataLoader(test\_data, batch\_size=batch\_size)

for X, y in test\_dataloader:

    print(f"Shape of X [N, C, H, W]: {X.shape}")

    print(f"Shape of y: {y.shape} {y.dtype}")

    break

# Define model

class NeuralNetwork(nn.Module):

    def \_\_init\_\_(self):

        super(NeuralNetwork, self).\_\_init\_\_()

        self.flatten = nn.Flatten()

        self.linear\_relu\_stack = nn.Sequential(

            nn.Linear(28\*28, 256),

            nn.ReLU(),

            nn.Linear(256, 256),

            nn.ReLU(),

            nn.Linear(256, 10)

        )

    def forward(self, x):

        x = self.flatten(x)

        logits = self.linear\_relu\_stack(x)

        return logits

model = NeuralNetwork().to(device)

print(model)

#define the loss function and set up optimiser.

loss\_fn = nn.CrossEntropyLoss()

optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

#Define train function.

def train(dataloader, model, loss\_fn, optimizer):

    size = len(dataloader.dataset)

    model.train()

    running\_loss = 0                            # accumuate loss of each input sample

    for batch, (X, y) in enumerate(dataloader):

        X, y = X.to(device), y.to(device)       # assign input samples to the available device (CPU or GPU) for computation.

        # Compute prediction error

        pred = model(X)

        loss = loss\_fn(pred, y)

        # Backpropagation

        optimizer.zero\_grad()                   # set the gradients to zero to avoid gradient accumuation. Gradient Accumulation is useful in some cases, like in training RNN.

        loss.backward()

        optimizer.step()

        running\_loss += loss.item()

        if batch % 100 == 0:

            loss, current = loss.item(), batch \* len(X)

            #print(f"training loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

    return running\_loss/len(dataloader)

#Define evaluation function.

def test(dataloader, model, loss\_fn):

    size = len(dataloader.dataset)

    num\_batches = len(dataloader)

    model.eval()

    test\_loss, correct = 0, 0

    with torch.no\_grad():         # No gradients need to be calculated for evaluation. Just the forward pass.

        for X, y in dataloader:

            X, y = X.to(device), y.to(device)

            pred = model(X)

            test\_loss += loss\_fn(pred, y).item()

            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test\_loss /= num\_batches

    correct /= size               # Normalise correctly classified count.

    print(f"test loss: \nAccuracy: {(100\*correct):>0.1f}%, avg loss: {test\_loss:>8f} \n")

    return test\_loss

#Initiate Training Process

epochs = 5 # (Try 5, 10 and 50 epoch and record values for average validation loss and accuracy)

train\_losses = []

test\_losses = []

for t in range(epochs):

    print(f"Epoch {t+1}\n-------------------------------")

    train\_loss = train(train\_dataloader, model, loss\_fn, optimizer)

    train\_losses.append(train\_loss)

    test\_loss = test(test\_dataloader, model, loss\_fn)

    test\_losses.append(test\_loss)

print("Done!")

# Plot training and validation losses.

plt.figure(figsize=(10,5))

plt.title("Training and Validation Loss")

plt.plot(test\_losses,label="validation")

plt.plot(train\_losses,label="training")

plt.xlabel("Epochs")

plt.ylabel("Loss")

plt.legend()

plt.show()

model = model.to('cpu')

torch.save(model.state\_dict(), "model.pth")

print("Saved PyTorch Model State to model.pth")

#Perform Inferrence

classes = [

    "zero",

    "one",

    "two",

    "three",

    "four",

    "five",

    "six",

    "seven",

    "eight",

    "nine",

]

#classes = np.array(test\_data.class\_to\_idx.values())

model.eval()

x, y = test\_data[0][0], test\_data[0][1]

with torch.no\_grad():

    pred = model(x)

    predicted, actual = classes[pred[0].argmax(0)], classes[y]

    print(f'Predicted: "{predicted}", Actual: "{actual}"')

The classes below were used for Fashion MINST instead of the ones in the original code.

####################################

classes = [

    "T-shirt/top",

    "Trouser",

    "Pullover",

    "Dress",

    "Coat",

    "Sandal",

    "Shirt",

    "Sneaker",

    "Bag",

    "Ankle boot",

]

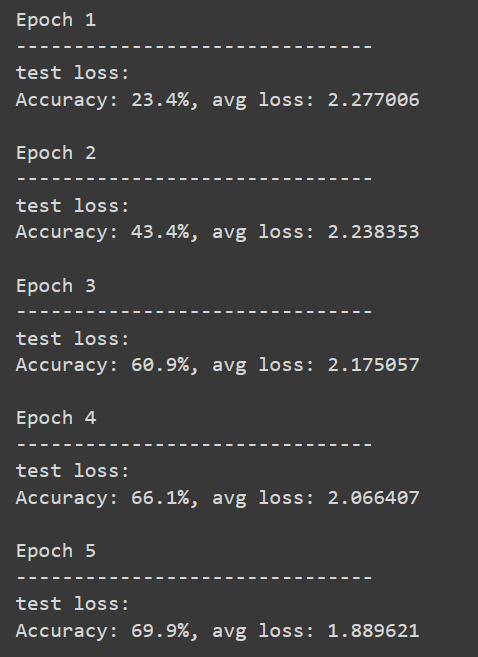
####################################

Outputs:

*MNIST Dataset*

## Question 1

### 128 Width, SGD without Momentum

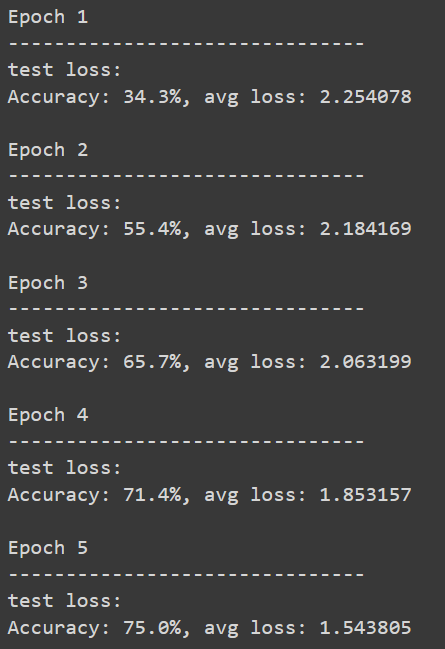
 Chart, line chart

Description automatically generated

### 256 Width, SGD without Momentum

### Chart, line chart Description automatically generated

### 512 Width, SGD without Momentum

 Chart, line chart

Description automatically generated

### Table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Epoch** | **128 Width, SGD without Momentum** | | **256 Width, SGD without Momentum** | | **512 Width, SGD without Momentum** | |
|  | **Accuracy %** | **Avg. Loss** | **Accuracy %** | **Avg. Loss** | **Accuracy %** | **Avg. Loss** |
| 1 | 23.4 | 2.277006 | 27.4 | 2.266749 | 34.3 | 2.254078 |
| 2 | 43.4 | 2.238353 | 47.1 | 2.218246 | 55.4 | 2.184169 |
| 3 | 60.9 | 2.175057 | 64.5 | 2.136616 | 65.7 | 2.063199 |
| 4 | 66.1 | 2.066407 | 69.1 | 1.992802 | 71.4 | 1.853157 |
| 5 | 69.9 | 1.889621 | 72.5 | 1.755129 | 75.0 | 1.543805 |

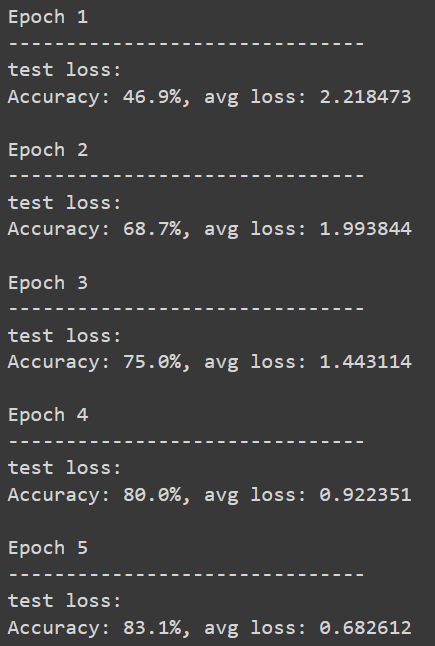
From the values in the table above we can conclude that when the number of neurons in the hidden layers are increased, the accuracy increases too as more combinations of ‘sub-properties’ or characteristics are tried out by the layer.

## Question 2

### 256 Width, SGD without Momentum

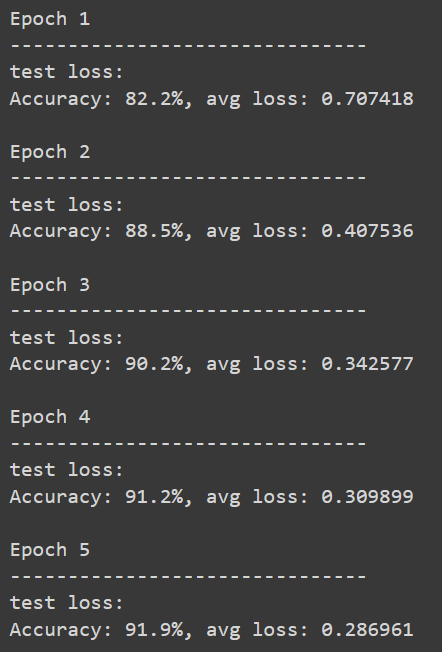
### Chart, line chart Description automatically generated

### 256 Width, SGD with 0.5 Momentum

 Chart, line chart

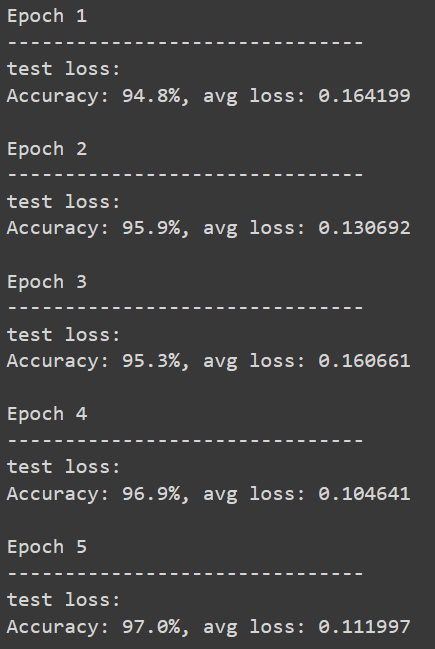
Description automatically generated

### 256 Width, SGD with 0.9 Momentum

 Chart, line chart

Description automatically generated

### 256 Width, Adam Optimizer without Momentum

 Chart, line chart

Description automatically generated

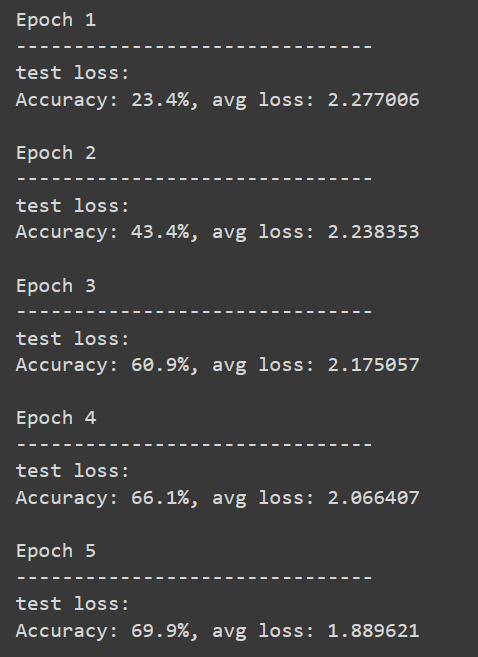
### Table (256 Width)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Epoch** | **SGD without Momentum** | | **SGD with 0.5 Momentum** | | **SGD with 0.9 Momentum** | | **Adam Optimizer without Momentum** | |
|  | **Accuracy %** | **Avg. Loss** | **Accuracy %** | **Avg. Loss** | **Accuracy %** | **Avg. Loss** | **Accuracy %** | **Avg. Loss** |
| 1 | 27.4 | 2.266749 | 46.9 | 2.218473 | 82.2 | 0.707418 | 94.8 | 0.164199 |
| 2 | 47.1 | 2.218246 | 68.7 | 1.993944 | 88.5 | 0.407536 | 95.9 | 0.130692 |
| 3 | 64.5 | 2.136616 | 75.0 | 1.443114 | 90.2 | 0.342577 | 95.3 | 0.160661 |
| 4 | 69.1 | 1.992802 | 80.0 | 0.922351 | 91.2 | 0.309899 | 96.9 | 0.104641 |
| 5 | 72.5 | 1.755129 | 83.1 | 0.682612 | 91.9 | 0.286961 | 97.0 | 0.111997 |

From the values in the table above, it can be concluded that when we add momentum to SGD the accuracy increases. Moreover, when the momentum increases, the accuracy increases along with it. These values also show that the Adam optimizer works better than SGD, but the behavior of Adam optimizer is more random than SGD.

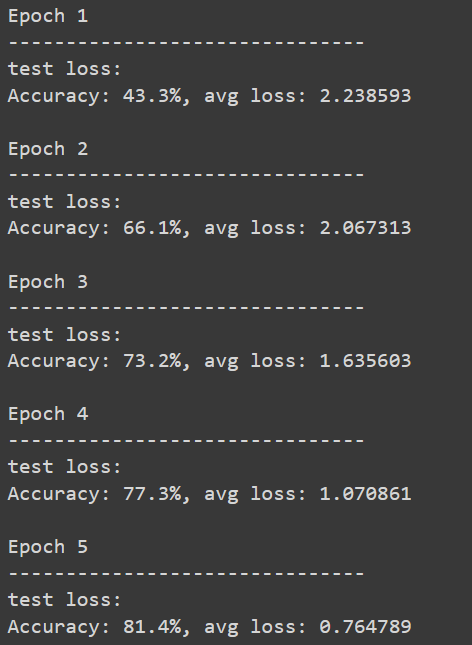
These tests were carried out for a network with 256 neurons in the hidden layer. The same observations can be made for a hidden layer with 128 or 512 neurons as shown below.

### 128 Width, SGD without Momentum

 Chart, line chart

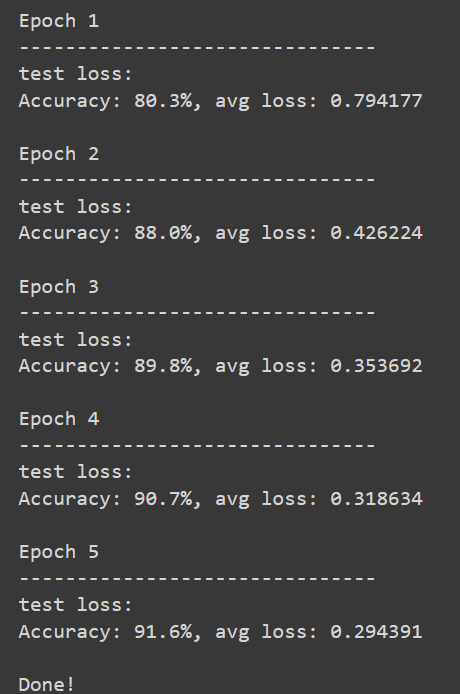
Description automatically generated

### 128 Width, SGD with 0.5 Momentum

 Chart, line chart

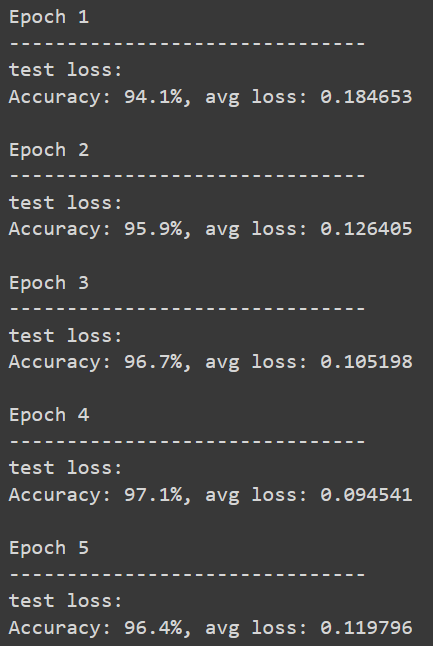
Description automatically generated

### 128 Width, SGD with 0.9 Momentum

 Chart, line chart

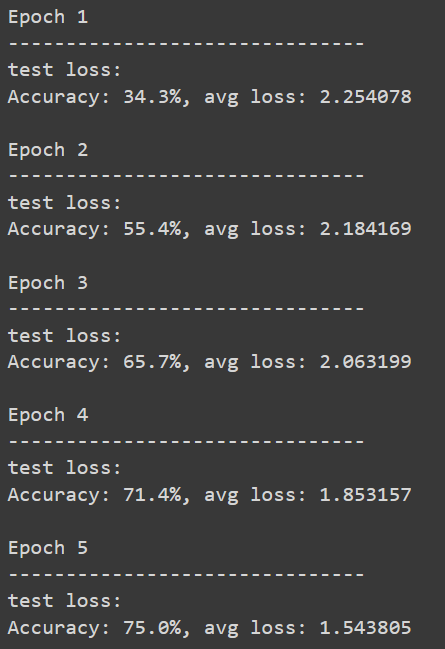
Description automatically generated

### 128 Width, Adam Optimizer without Momentum

 Chart, line chart

Description automatically generated

### 512 Width, SGD without Momentum

 Chart, line chart

Description automatically generated

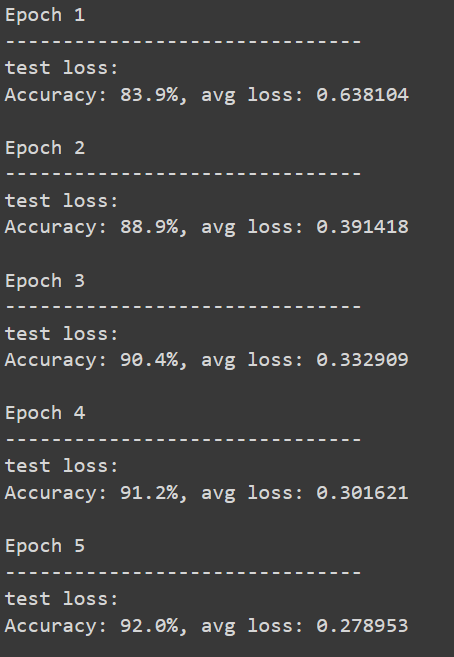
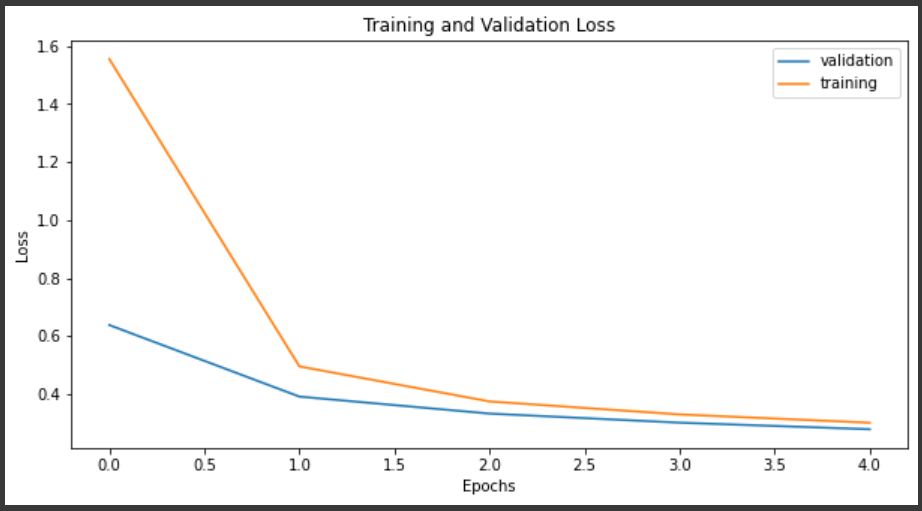
### 512 Width, SGD with 0.5 Momentum

Text

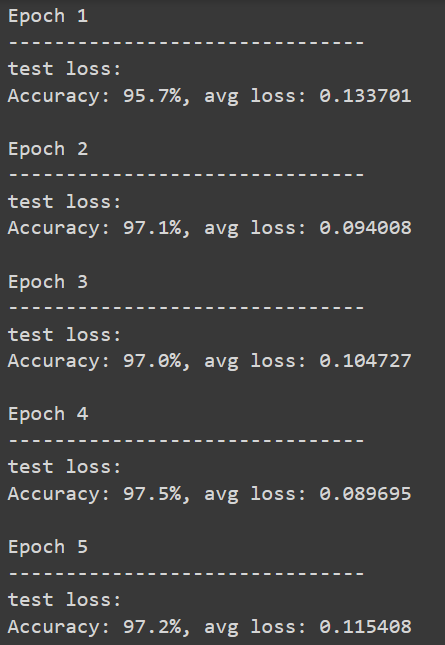
Description automatically generated Chart, line chart

Description automatically generated

### 512 Width, SGD with 0.9 Momentum

### 512 Width, Adam Optimizer without Momentum

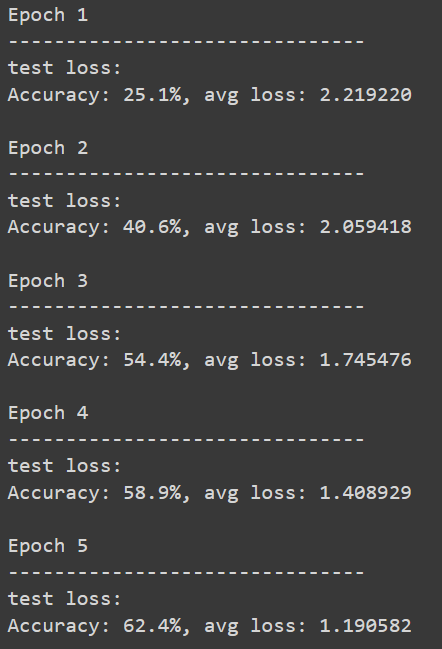
 Chart, line chart

Description automatically generated

*Fashion MNIST Dataset*

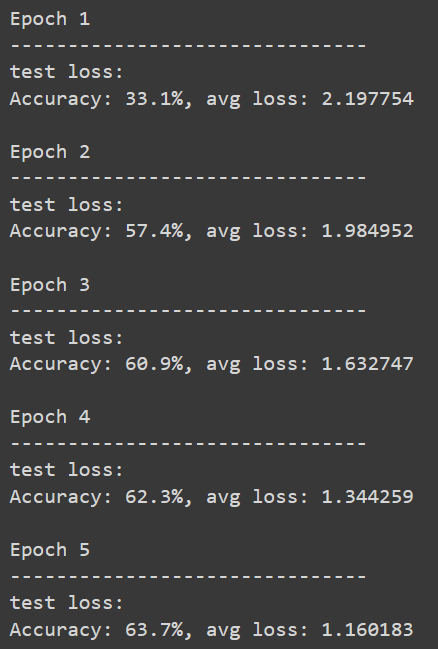
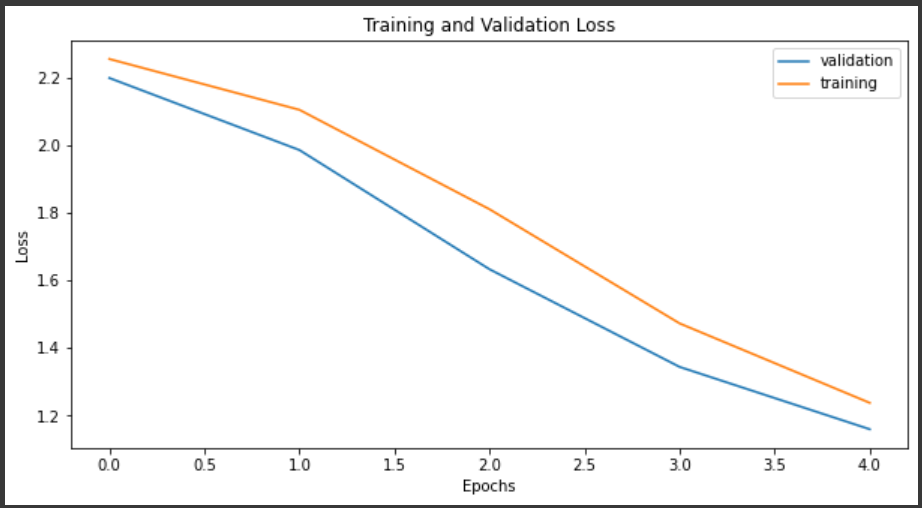
## Question 3

### 128 Width, SGD without Momentum

 Chart, line chart

Description automatically generated

### 256 Width, SGD without Momentum

### 512 Width, SGD without Momentum

Text

Description automatically generated Chart, line chart

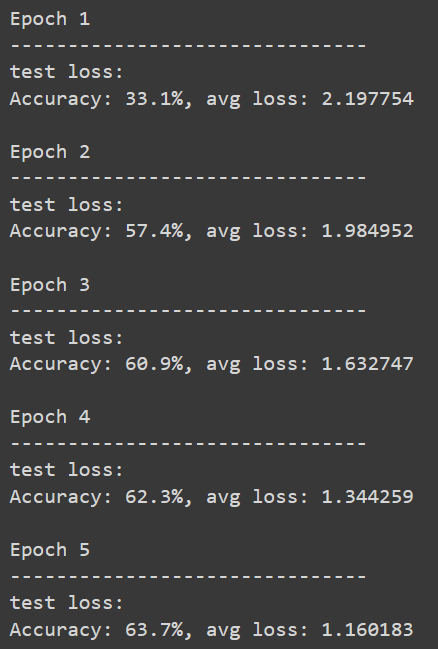
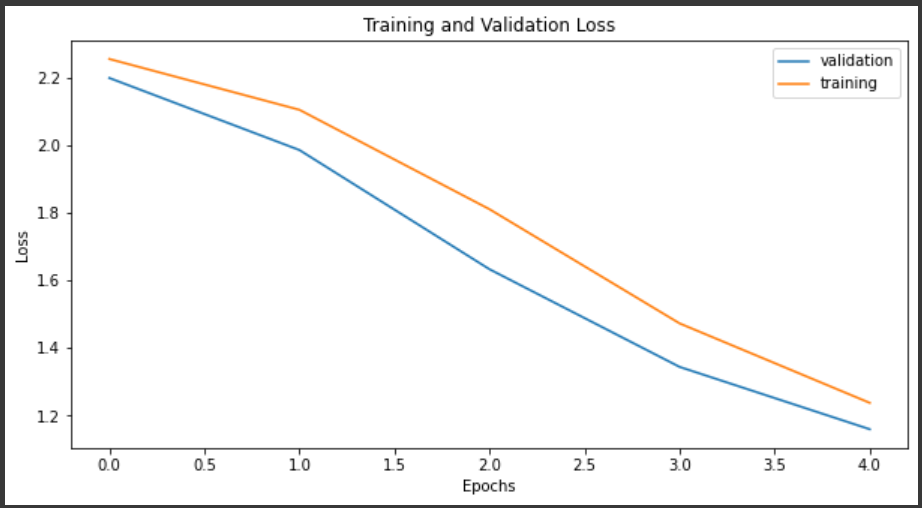
Description automatically generated

### Table

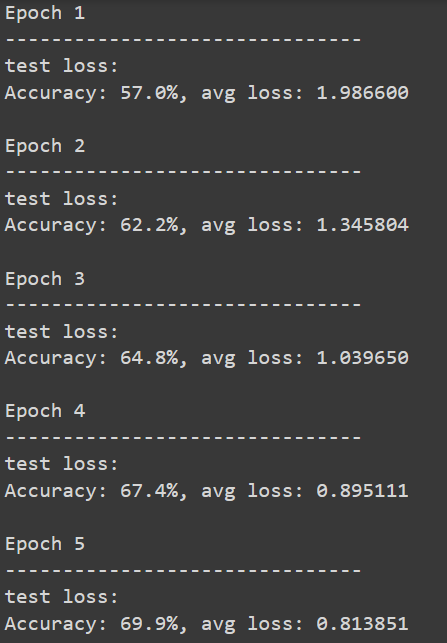
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Epoch** | **128 Width, SGD without Momentum** | | **256 Width, SGD without Momentum** | | **512 Width, SGD without Momentum** | |
|  | **Accuracy %** | **Avg. Loss** | **Accuracy %** | **Avg. Loss** | **Accuracy %** | **Avg. Loss** |
| 1 | 25.1 | 2.219220 | 33.1 | 2.197754 | 57.8 | 2.159178 |
| 2 | 40.6 | 2.059418 | 57.4 | 1.984952 | 61.3 | 1.882623 |
| 3 | 54.4 | 1.745476 | 60.9 | 1.632747 | 62.5 | 1.502814 |
| 4 | 58.9 | 1.408929 | 62.3 | 1.344259 | 64.1 | 1.229386 |
| 5 | 62.4 | 1.190582 | 63.7 | 1.160183 | 65.1 | 1.063732 |

From the values in the table above we can conclude that when the number of neurons in the hidden layers are increased, the accuracy increases too as more combinations of ‘sub-properties’ or characteristics are tried out by the layer.

### 256 Width, SGD without Momentum

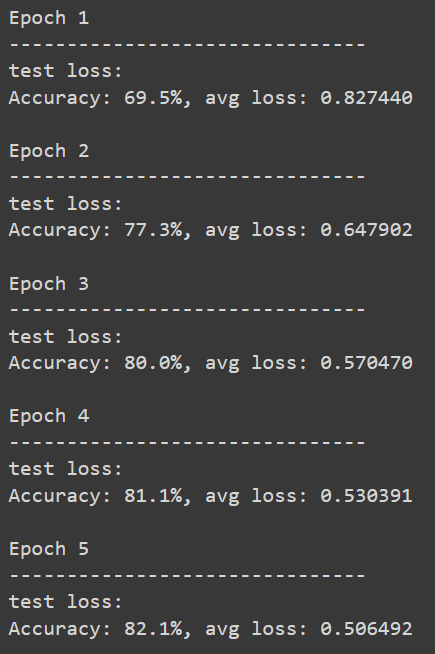
 

### 256 Width, SGD with 0.5 Momentum

 Chart, line chart

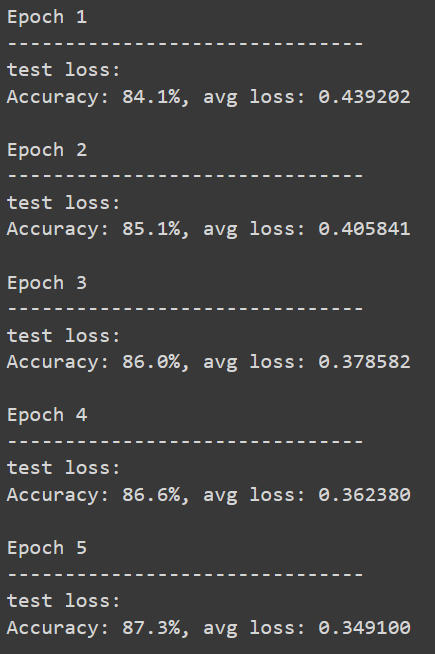
Description automatically generated

### 256 Width, SGD with 0.9 Momentum

 Chart, line chart

Description automatically generated

### 256 Width, Adam Optimizer without Momentum

 Chart, line chart

Description automatically generated

### Table (256 Width)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Epoch** | **SGD without Momentum** | | **SGD with 0.5 Momentum** | | **SGD with 0.9 Momentum** | | **Adam Optimizer without Momentum** | |
|  | **Accuracy %** | **Avg. Loss** | **Accuracy %** | **Avg. Loss** | **Accuracy %** | **Avg. Loss** | **Accuracy %** | **Avg. Loss** |
| 1 | 33.1 | 2.197754 | 57.0 | 1.986600 | 69.5 | 0.827440 | 84.1 | 0.439202 |
| 2 | 57.4 | 1.984952 | 62.2 | 1.345804 | 77.3 | 0.647902 | 85.1 | 0.405841 |
| 3 | 60.9 | 1.632747 | 64.8 | 1.039650 | 80.0 | 0.570470 | 86.0 | 0.378582 |
| 4 | 62.3 | 1.344259 | 67.4 | 0.895111 | 81.1 | 0.530391 | 86.6 | 0.362380 |
| 5 | 63.7 | 1.160183 | 69.9 | 0.813851 | 82.1 | 0.506492 | 87.3 | 0.349100 |

From the values in the table above, it can be concluded that when we add momentum to SGD the accuracy increases. Moreover, when the momentum increases, the accuracy increases along with it. These values also show that the Adam optimizer works better than SGD, but the behavior of Adam optimizer is more random than SGD.

These tests were carried out for a network with 256 neurons in the hidden layer. The same observations can be made for a hidden layer with 128 or 512 neurons.