ELE 489: Fundamentals of Machine Learning
Prof. Seniha Esen Yuksel
Department of Electrical and Electronics Engineering
Hacettepe University

Gaye AKSU
2200357047

HW-4

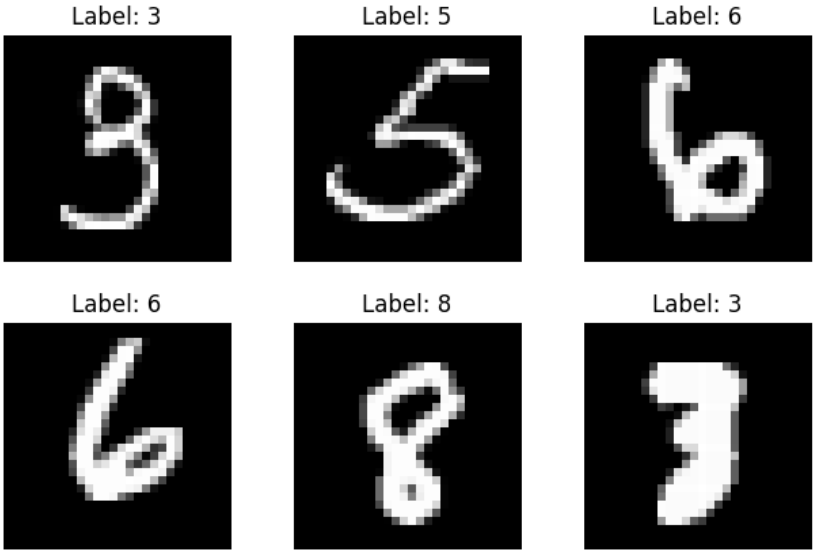
In this homework, we try to recognize handwritten digits. To design and train CNNs, I used PyTorch because it is more suitable for beginners. Also, as it says in the explanations of the homework, I used torchvision.datasets.NMIST library.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
```

Then I load the NMIST dataset to recognize handwritten digits. NMIST dataset is a dataset consisting of handwritten 28x28 pixel grayscale digit images.

```
# 1. Veri seti
transform = transforms.Compose([transforms.ToTensor()])
train_dataset = datasets.MNIST(root='./data', train=True,
transform=transform, download=True)
test_dataset = datasets.MNIST(root='./data', train=False,
transform=transform, download=True)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

examples = enumerate(train_loader)
batch_idx, (example_data, example_targets) = next(examples)

fig = plt.figure()
for i in range(6):
    plt.subplot(2,3,i+1)
    plt.tight_layout()
    plt.imshow(example_data[i][0], cmap='gray', interpolation='none')
    plt.title(f"Label: {example_targets[i].item()}")
    plt.axis('off')
plt.show()
```

These are the some examples from the NMIST dataset.



Now we are ready for the questions.

Q1) Train a Baseline CNN

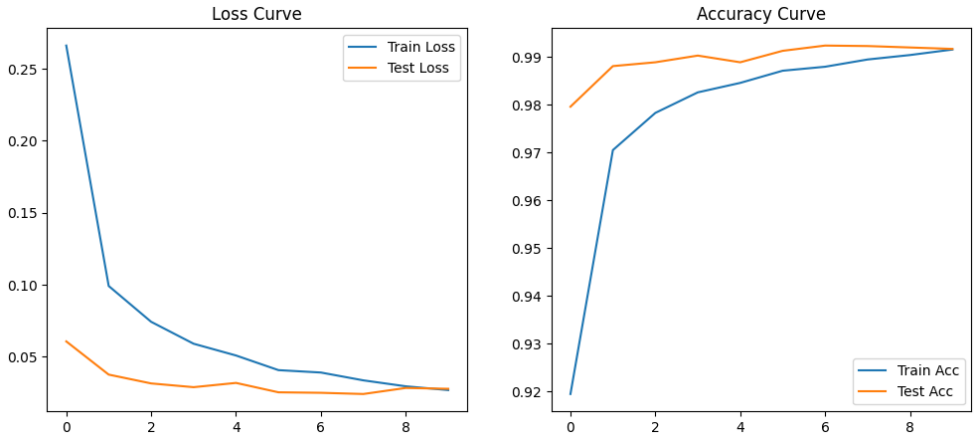
I used the architecture that givved in the explenations of the homework which is;

Conv→ ReLU → MaxPool → Conv → ReLU → MaxPool → Flatten → Dense → ReLU → Dropout(0.5) → Dense(10) → Softmax

```
class BaselineCNN(nn.Module):
    def __init__(self):
        super(BaselineCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 3)
        self.fc1 = nn.Linear(64*5*5, 128)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(128, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = x.view(-1, 64*5*5)
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

After the MNIST dataset for 10 epochs using the Adam optimizer and cross-entropy loss.

```
Epoch 1/10: Train Loss=0.2661, Train Acc=91.94%, Test Loss=0.0607, Test Acc=97.96%
Epoch 2/10: Train Loss=0.0992, Train Acc=97.05%, Test Loss=0.0377, Test Acc=98.81%
Epoch 3/10: Train Loss=0.0743, Train Acc=97.83%, Test Loss=0.0316, Test Acc=98.89%
Epoch 4/10: Train Loss=0.0591, Train Acc=98.26%, Test Loss=0.0290, Test Acc=99.03%
Epoch 5/10: Train Loss=0.0510, Train Acc=98.46%, Test Loss=0.0319, Test Acc=98.89%
Epoch 6/10: Train Loss=0.0408, Train Acc=98.71%, Test Loss=0.0254, Test Acc=99.13%
Epoch 7/10: Train Loss=0.0391, Train Acc=98.80%, Test Loss=0.0251, Test Acc=99.24%
Epoch 8/10: Train Loss=0.0337, Train Acc=98.95%, Test Loss=0.0242, Test Acc=99.23%
Epoch 9/10: Train Loss=0.0296, Train Acc=99.04%, Test Loss=0.0284, Test Acc=99.20%
Epoch 10/10: Train Loss=0.0269, Train Acc=99.16%, Test Loss=0.0278, Test Acc=99.17%
```



The loss graphs show that training and test loss decreased well, meaning the model learned correctly. The training and test loss values got close at the end, so the model did not overfit. The accuracy graphs show that the training accuracy increased from 92% to about 99%, and the test accuracy improved from 98% to about 99%. This means the model works well on new data. The final test accuracy was around 99.17%, which is good for this basic CNN model.

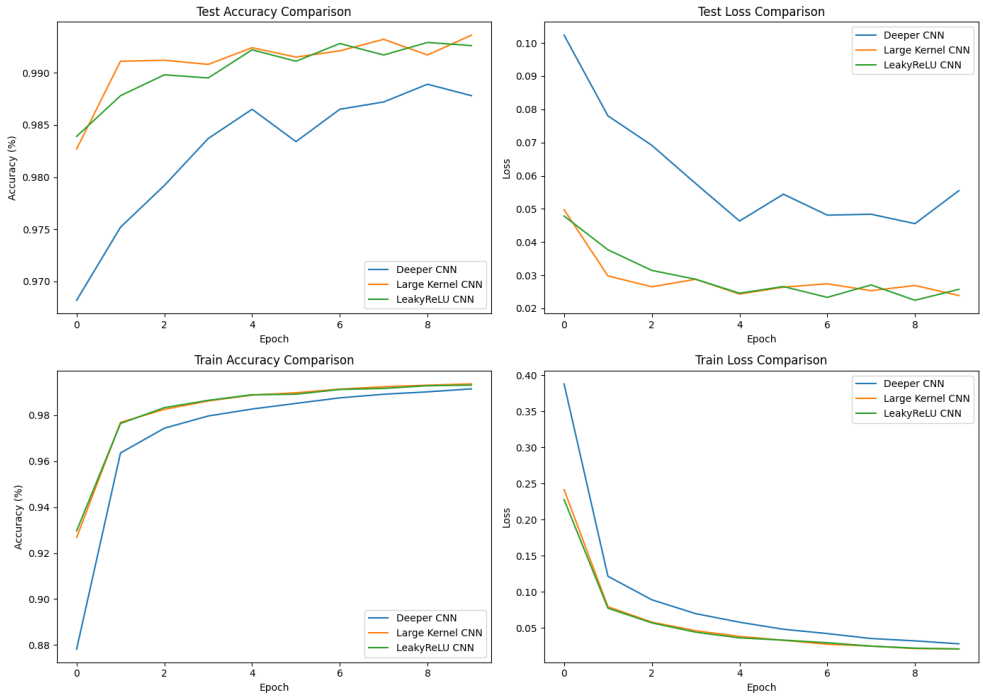
Q2) Try at least 3 architectural variants

For this question, I chose to try Deeper network, different Kernel size (5x5 conv) and LeakyReLU.

```python
class DeeperCNN(nn.Module):
    def __init__(self):
        super(DeeperCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 3)
        self.conv3 = nn.Conv2d(64, 128, 3)
        self.gap = nn.AdaptiveAvgPool2d((1, 1))  # GAP ekledik
        self.fc1 = nn.Linear(128, 128)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(128, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = self.pool(self.relu(self.conv3(x)))
        x = self.gap(x)
        x = torch.flatten(x, 1)
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

```python
class LargeKernelCNN(nn.Module):
    def __init__(self):
        super(LargeKernelCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 5)    # 5x5 kernel
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 5)
        self.fc1 = nn.Linear(64*4*4, 128)   # Boyutlara dikkat et
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(128, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = x.view(-1, 64*4*4)
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

```python
class LeakyReLUCNN(nn.Module):
    def __init__(self):
        super(LeakyReLUCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 3)
        self.fc1 = nn.Linear(64*5*5, 128)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(128, 10)
        self.leaky_relu = nn.LeakyReLU(negative_slope=0.01)

    def forward(self, x):
        x = self.pool(self.leaky_relu(self.conv1(x)))
        x = self.pool(self.leaky_relu(self.conv2(x)))
        x = x.view(-1, 64*5*5)
        x = self.leaky_relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```



The large kernel CNN had the best test accuracy, reaching about 99.36% at the last epoch.
The LeakyReLU CNN also performed very well, with test accuracy around 99.26%.
The deeper CNN had slightly lower accuracy, about 98.78% on test data.

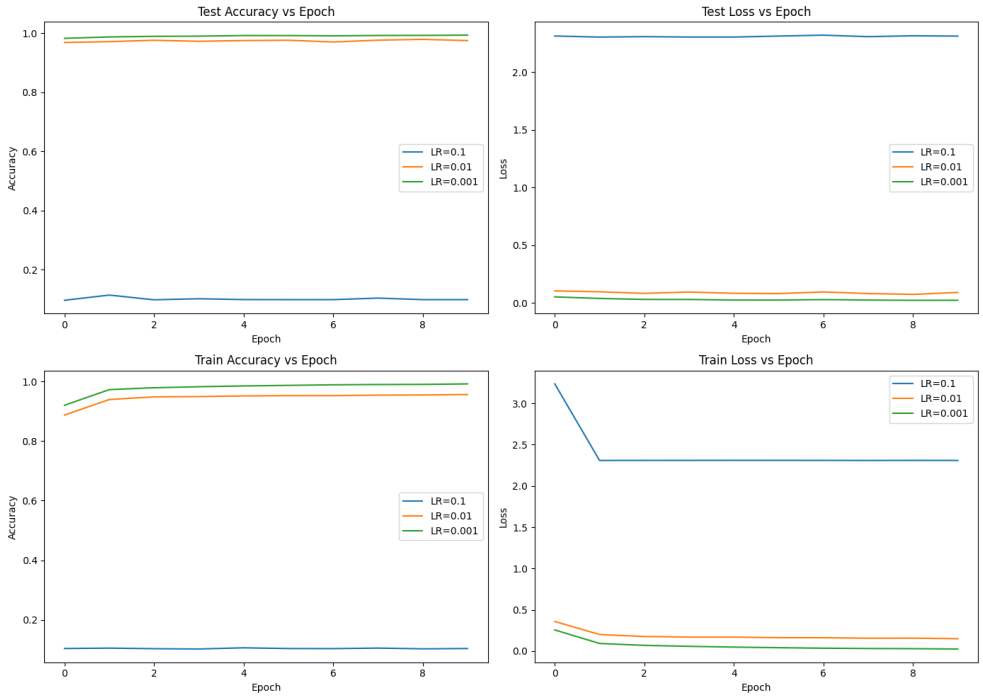
From the results, we can say loss values for large kernel and LeakyReLU CNNs were lower than the deeper CNN, meaning they learned better. In short, using bigger kernels or LeakyReLU activation helped improve the model more than just making the network deeper.

Q3)

Learning rate:

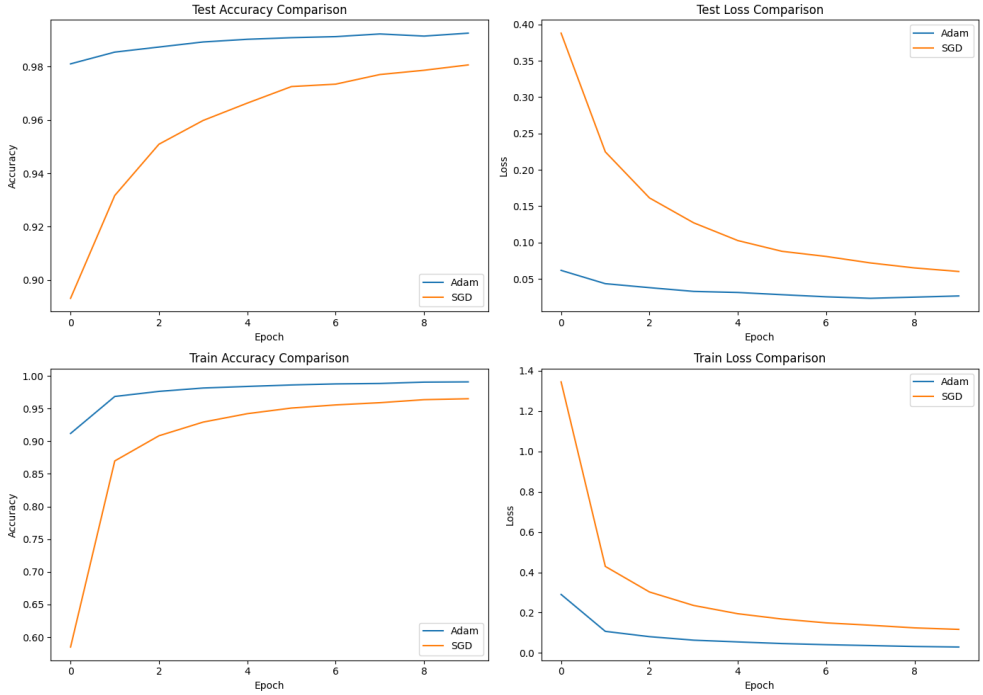
Learning rate controls how much the model changes during training. If it is too high, the model can miss the best solution and fail to learn well. If it is too low, training will be very slow and may get stuck. Finding a good learning rate helps the model learn fast and well.

I test effect of the learning rates with using 0.1, 0.01 and 0.001 learning rates.



The results were as we expected. In the plots, a very high learning rate (0.1) makes the model fail; both train and test accuracy stay at about 10%, and loss stays high. With a lower rate (0.01), the model learns quickly and reaches around 98% accuracy. The smallest rate (0.001) learns more slowly but ends with the highest accuracy (99%+) and lowest loss. As we expected, too large a learning rate breaks training, a medium rate converges well, and a small rate gives stable, accurate results.

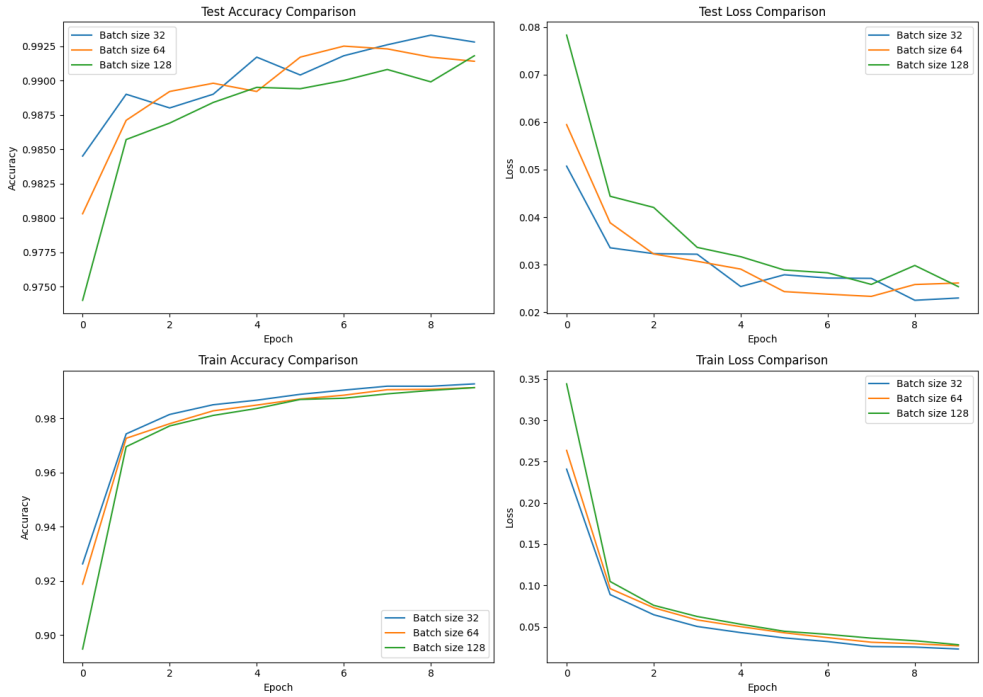
Optimizer (Adam vs SGD):
Optimizers help update model weights to reduce errors. Adam is smart and changes its learning rate automatically, so it learns faster. SGD is simple and slower, but sometimes better with more training.



The results show that the Adam optimizer performs better than SGD for this CNN model. Adam reaches higher accuracy and lower loss faster on both training and test data. As we expected, Adam adapts the learning rate during training, helping the model learn more efficiently. On the other hand, SGD starts slower and does not reach as high of accuracy within the same number of epochs. As we can see in the loss curves, Adam's loss is consistently lower than SGD's, showing better optimization. Overall, these graphs confirm that Adam is a more effective optimizer for this task, and our results are correct and reliable.

Batch Size:
The batch size is how many samples the model uses before updating its weights. Small batch size means more updates, better learning, but slower training. Big batch size trains faster but may not learn as well.

I test the effect of the batch size by using 32, 64, and 128 batch sizes.

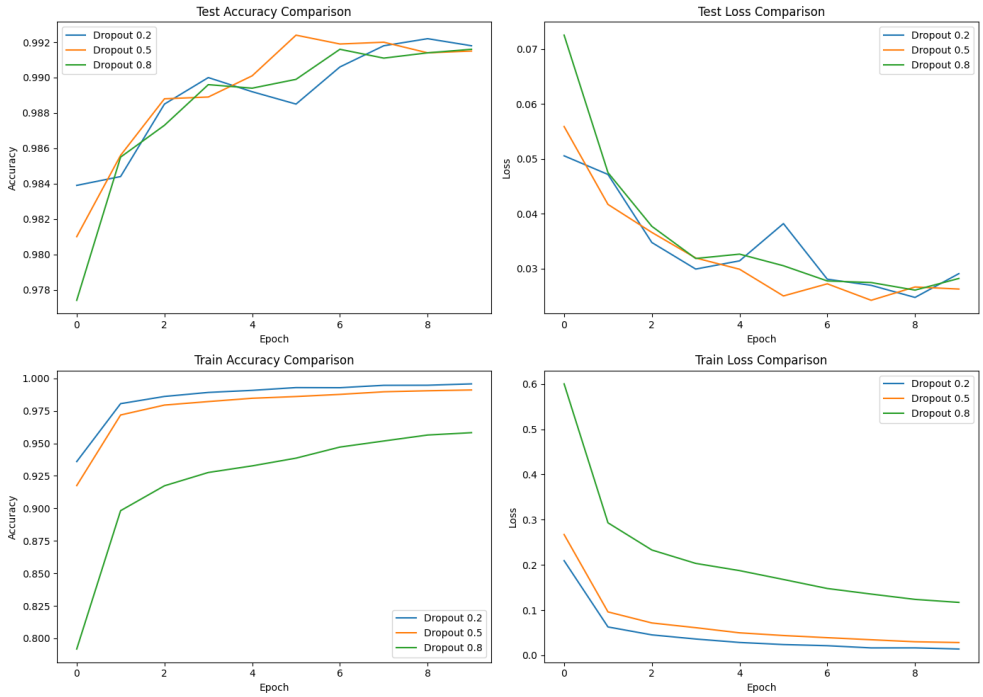


As we can see in the graphs, batch size 64 had the best test accuracy, slightly better than 32 and 128. Medium batch sizes often balance speed and learning quality well. Smaller batch sizes usually help with better learning but take more time. Larger batch sizes train faster but may not generalize as well. Also in the loss curves, batch size 64 had the lowest test loss overall. These results are consistent and show that batch size 64 is a good choice for this task.

Dropout Rate:

Dropout turns off some neurons randomly during training to avoid overfitting. A good dropout stops the model from just memorizing training data.

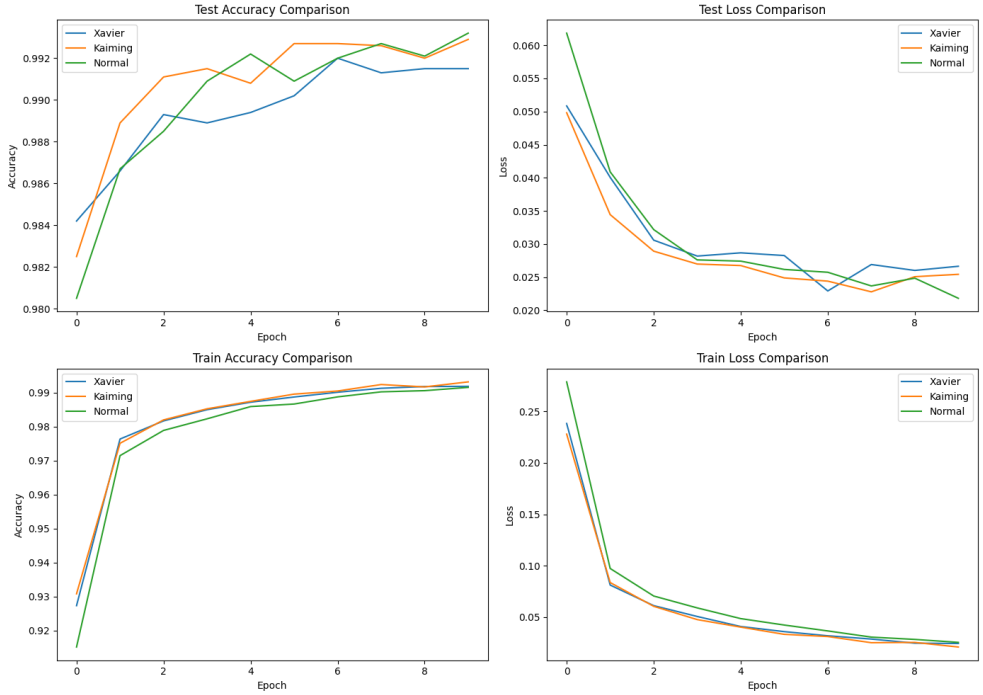
I test the effect of the dropout rate by using 0.2, 0.5, and 0.8 dropout rates.



Dropout 0.2 and 0.5 have high accuracy and low loss in both training and testing, meaning good learning and generalization. Dropout 0.8 has lower training accuracy and higher loss, showing the model struggles to learn well with too much dropout. As we expected, too high dropout stops the network from learning enough patterns. Overall, moderate dropout values work best here.

Weight Initialization:
Weight initialization is how weights start before training. Good initialization helps the model start training well. Bad initialization can make training slow or stuck.

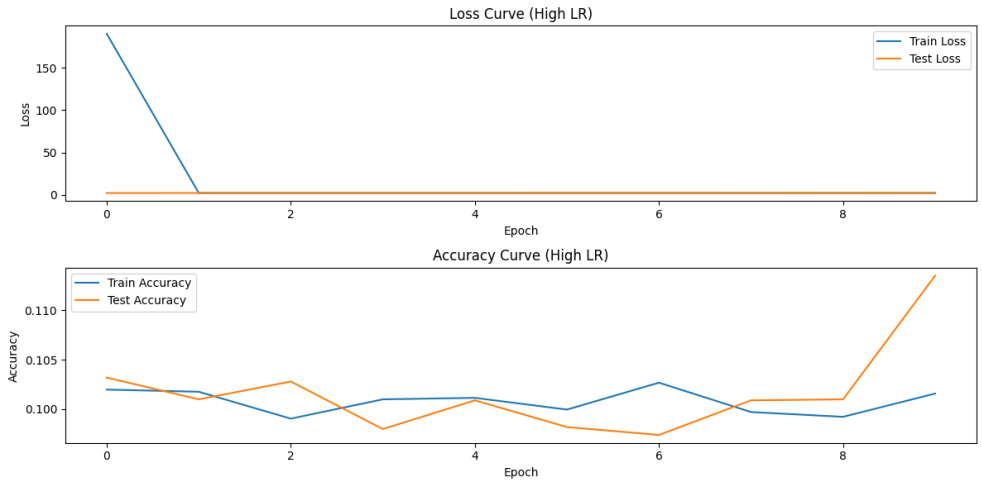
I test the effect of the weight initialization by using Xavier, Kaiming, and Normal weight initializations.



As we expected, Kaiming initialization performed slightly better, reaching the highest accuracy and lowest loss in most epochs. Xavier and Normal also gave good results but were a little behind. AS we know, Kaiming initialization is designed for ReLU activations and helps training start better, which improves learning speed and final accuracy.

Q4) Try to Break the CNN.
I tried to break the CNN with a very high learning rate, very high dropout, and a very high SGD momentum.

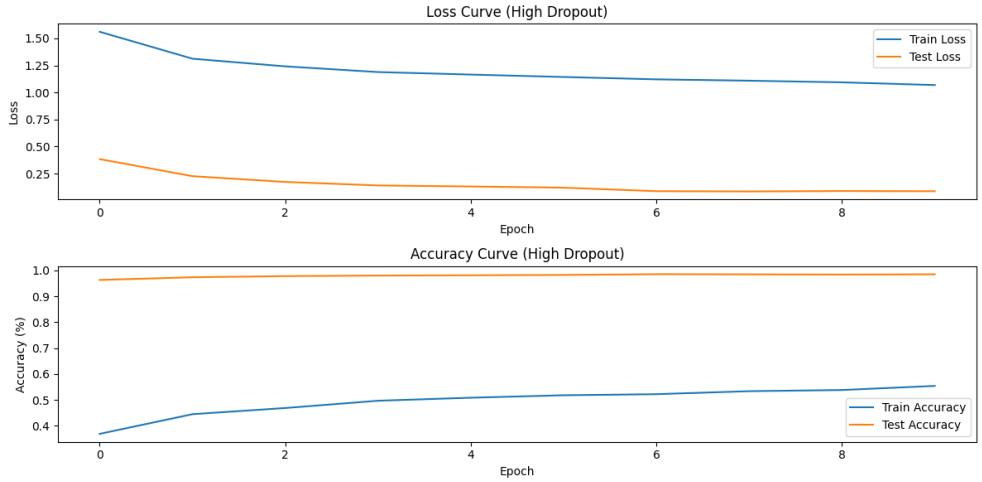
High Learning Rate:
If we use a very high learning rate, the CNN will update its weights by too large amounts each step. Instead of slowly moving towards the best solution, the loss will jump up and down and never settle. This makes the network fail to learn useful patterns, and its accuracy stays low or even gets worse. In theory, big weight changes overshoot the ideal values and break the training process, so the model cannot converge.



When we used a very high learning rate, the training completely failed. The training loss started extremely high and then dropped to zero in one step, which is not real learning. At the same time, the test loss stayed almost flat, and both train and test accuracy stayed around 10%, the same as random guessing for ten classes. These curves show that the model could not learn any patterns because the weight updates were too large and broke the training process.

High Dropout:

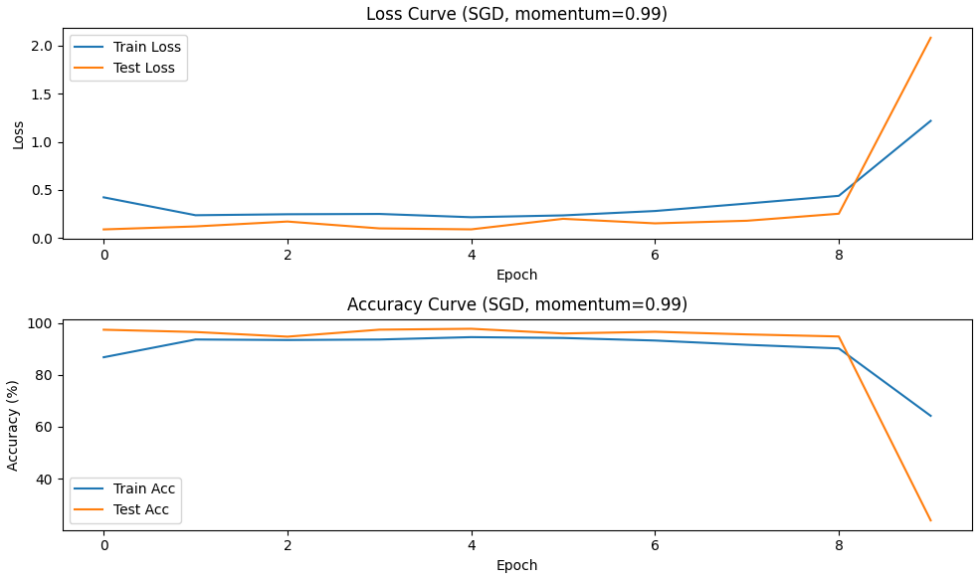
If we use a very high dropout rate, the network will randomly turn off most of its neurons during training. This means it has too few active units to learn important features from the data. As a result, the model will struggle to find good patterns and its training will be slow or fail. In theory, dropping too many neurons breaks the network's ability to build strong connections, so it cannot converge to a good solution, and its accuracy stays low.



When we used a very high dropout rate, the model struggled to learn. The training loss stayed high (above 1.0) and the training accuracy only rose from about 40% to 55%, because most neurons were turned off each step. In contrast, the test loss went down and test accuracy stayed around 98%, since dropout is only applied in training, not in testing. These results show that too much dropout prevents the network from learning the data patterns properly.

High SGD Momentum:

If we use a very high momentum in SGD, the updates will keep too much speed and overshoot the best values. Instead of slowly settling into a minimum, the loss will oscillate and never converge. This makes the network fail to learn stable patterns and its accuracy stays low. In theory, too much momentum adds inertia so the weights cannot settle correctly.



As we expected, using a very high momentum with SGD causes instability. The training loss stays low at first, but then increases sharply at the end, and the test loss rises a lot. Accuracy goes up in the middle but suddenly drops near the last epochs. This means the model starts to fail and cannot learn well in the end.

Appendix:

You can see the code that I wrote for this homework in this GitHub link.

https://github.com/AksuGaye/ELE489_HW4.git

I wrote the code in Jupyter therefore, there are a lot of lines and repeated code blocks.