Reyhane Shahrokhian 99521361

HomeWork2 of DeepLearning Course

Dr. DavoodAbadi

Question 1:

1-1:

- Overfitting: It occurs when a neural network learns the training data too well, capturing noise, and specific details of the training dataset, rather than generalizing the underlying patterns and relationships. When this happens, the model performs very well on the training data but poorly on unseen data and it means that loss on the training data decreases but on the validation data starts increasing after reaching a minimum. It mostly happens if the model has a very complex structure with too many parameters.
- Underfitting: This happens when a neural network is too simple to capture the underlying patterns and relationships in the data, resulting in a model that performs poorly on both the training and unseen data. Thus the loss on both the training and validation data remains high and doesn't improve significantly. It may occur if the model has too few parameters, layers, or features to represent the complexity of the data.

1-2:

The overfit models tend to be sensitive to small noises in the input data. So we can collect some test data and measure the model’s ability, Then by adding adversarial examples to test data, we can gain insights into the model's generalization abilities and detect overfitting. If the model's

performance degrades significantly on adversarial examples, it suggests that the model has learned specific patterns in the training data that do not generalize well.

1-3:

During training, the elements where the mask is 1 should be dropped out, and during testing, all elements should be used. So:

Output After Applying dropout during training:

0	-0.7	-0.2	0
-2.3	0	0	-0.9
-0.5	0	0	-0.4
0	-0.4	-2.6	0

Output After Applying dropout during testing:

1.6	-0.7	-0.2	1.9
-2.3	2.5	2.5	-0.9
-0.5	3.2	3.7	-0.4
1.3	-0.4	-2.6	1.2

Question 2:

2-1:

By changing the value of k in the k-NN algorithm, we can control the trade-off between bias and variance. Lower values of k lead to low bias but the model is less stable so it has high variance, while with higher values of k, we have a more stable model leading to low variance and high bias. The choice of the optimal k value depends on our problem and dataset, and a balance must be applied to achieve good generalization to unseen data.

2-2:

- False. Regularization is a technique used to prevent overfitting in machine learning models. It also helps in finding a better balance between bias and variance. So regularization is generally employed to improve a model's generalization performance, not weaken it.
- False. Adding a large number of new features can actually increase the risk of overfitting, rather than preventing it. Overfitting occurs when a model becomes too complex and fits the training data noise. Adding a large number of features, especially if they are irrelevant or redundant, can increase the model's complexity, making it more prone to overfitting.
- False. Increasing the regularization parameter λ reduces the risk of overfitting. By increasing λ, the penalty for complex models is increased, which encourages the model to have simpler coefficients or features. This, in turn, reduces the model's capacity to fit noise in the training data and makes it more likely to generalize well to unseen data, thus reducing the risk of overfitting.

2-3:

L1 regularization struggles overfitting by shrinking the parameters towards 0 which makes some features obsolete. A selection of the input features would have weights equal to zero, and the rest would be non-zero, So the L1 norm would be sparse.

L2 regularization struggles overfitting by forcing weights to be small, but not making them exactly 0. Actually L2 regularization returns a non-sparse solution since the weights will be non-zero (although some may be close to 0).

Accordingly we have:

- L2 : cause there is no feature selection and weights are all non-zero and non-sparse.

- L1 : there is feature selection and we have zero weights.
- L2 : weights are non-zero and there isn't feature selection.
- L1 : like the second one, there is feature selection and we have zero weight.

Question 3:

3-1:

Knowledge Distillation is a model compression technique that involves training a smaller model (the student) to mimic the behavior of a larger model (the teacher). Knowledge distillation allows to create a smaller, more lightweight model that requires fewer computational resources and less memory. This is especially important for deployment on resource-constrained devices or in scenarios where speed and efficiency are critical. By transferring the knowledge from a larger, well-generalizing teacher model, the student model can often generalize better.

3-2:

The learning process in the knowledge distillation system can be explained as follows:

1. The teacher model is trained on a dataset of labeled data.
2. Once the teacher model is trained, it is used to generate soft labels for the student model. Soft labels are probability distributions over the possible output classes, rather than hard labels, which are single output classes. They contain more information about its knowledge than hard labels would.
3. The student model is then trained to minimize the distillation loss, which is a measure of the difference between the student's predictions and the teacher's soft labels. The student model is also trained on the labeled dataset, but with a lower weight on this loss function.

3-3:

The distillation loss is minimized during the training process to encourage the student model to

learn the teacher's knowledge, while the loss function on the labeled dataset measures the difference between the student's predictions and the hard labels in the dataset. It's also minimized during the training process to ensure that the student model is able to learn from the data. Accordingly, the weights of the student network are updated according to a weighted combination of the distillation loss and the loss function on the labeled dataset. The weights of the two loss functions are hyperparameters that can be tuned to achieve the best performance. This is the equation of student loss:

$$student\_loss = w1 \times distillation\_loss + w2 \times labeled\_dataset\_loss$$

However, mostly the distillation loss is given more weight than the loss function on the labeled dataset. This is because the distillation loss is directly responsible for transferring the teacher's knowledge to the student model.

Question 4:

Code analyzes:

The model class is defined, and its constructor (__init__) initializes the model with input data (x) and labels (y). It also calls three initialization methods: _initalize_parameters, _initalize_moms, and _initalize_RMSs. The _initalize_parameters method initializes the weights and biases with random values using the _random_tensor method. The _initalize_moms method initializes the momentum and the _initalize_RMSs method initializes the root mean square (RMS) values for each parameter in the model.

Next, the _nn method performs the forward pass of the neural network by calculations on input to generate the output. Then, the _loss_func method calculates the loss between the predicted values (preds) and the target values (yb). It computes the mean squared error (MSE) between the two tensors.

The train method trains the model using an optimizer. Inside the method, there is a loop that iterates over different learning rates (lrs). For each learning rate, it trains the model until the loss decreases below 0.1 or the maximum number of iterations (1000) is reached. Within the training loop, the forward pass is performed (_nn), and the loss is calculated (_loss_func). Then, the backward method is called on the loss tensor to compute the gradients. Then, to update the weights and biases of the model the optimizer function is called four times. The optimizer function takes the parameter tensor (a), learning rate (lr), momentum values (moms), and RMS values (__) as arguments. After training with one optimizer, the model is re-initialized by calling the initialization methods (_initalize_parameters, _initalize_moms, _initalize_RMSs) to reset the model's parameters and optimizer-specific variables. Finally, the code generates fake labels for the input data using the generate_fake_labels function. Then, the model is trained twice, first using the SGD optimizer and then using the momentum optimizer.

The SGD optimizer function updates the parameter tensor (a) using stochastic gradient descent. The momentum optimizer function updates the parameter tensor (a) using momentum. It calculates the momentum as a combination of the current gradient and the previous momentum value.

Results analyzes:

- SGD*:* A higher learning rate will cause the weights to be updated more quickly, but it may also lead to instability and oscillations in the loss. The plots show that the MSE loss decreases for all four learning rates, but the rate of decrease is different for each learning rate. The learning rate of 0.001 has the slowest rate of decrease, while the learning rate of 1.0 has the fastest rate of decrease.

  The MSE loss for the learning rate of 0.1 increases at the beginning of training due to the

optimizer overshooting the minimum of the loss function. This can happen when the learning rate is too high, as the optimizer takes large steps towards the minimum and can end up on the other side of it and the reason why the loss eventually decreases is that the optimizer is still able to make progress towards the minimum because the gradient of the loss function is still pointing in the right direction.

- Momentum*:* The momentum function does not depend on learning rate, so all four plots have the same behavior. The loss increases and decreases repeatedly because of the momentum term, which is used to accelerate the learning process. The oscillations are caused by the fact that the momentum term can amplify noise in the gradient updates which can prevent the model from getting stuck in local minima.

Question 5:

5-1:

This is the model that I defined for this task and as the question mentioned, I use CrossEntropy loss function and SGD optimizer. I used some linear layer and ReLU activation function in order to make non-linearity. I also use Dropouts to prevent the model from overfitting.

```
## Define the model
################ Your code ############
model = nn.Sequential(
    nn.Linear(input_size, 256),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(256, 128),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Linear(64, out_size),
)
#######################################
```

```
################ Your code ############
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)
#######################################
```

Then in the training loop at the forward pass section, I give the model images as input and compare the output and real labels with loss function.

```
# forward pass
################ Your code ############
output = model(images)
loss = criterion(output, labels)
#####################################
```

```
Training loss: 0.6250458441372874
Training loss: 0.41797709244210074
Training loss: 0.37972920659635623
Training loss: 0.35275964931384335
Training loss: 0.3342743981431034
Training loss: 0.3193010955429408
Training loss: 0.3062284154328964
Training loss: 0.29708115294226195
Training loss: 0.28696301066354396
Training loss: 0.27781492680597153
```

After training completed, I test the model with 10 images(I had some problem with d2l so I wrote the test code myself):

```
for images, labels in testloader:
    images = images.view(images.shape[0], -1)
    _, predicted = torch.max(model(images), dim=1)
    for i in range(num_samples):
        print("Predicted label:", predicted[i].item())
        print("True label:", labels[i].item())
        print()
    break
```

```
Predicted label: 1
True label: 1

Predicted label: 7
True label: 7

Predicted label: 7
True label: 7

Predicted label: 6
True label: 6

Predicted label: 2
True label: 6

Predicted label: 8
True label: 8

Predicted label: 7
True label: 7

Predicted label: 0
True label: 0

Predicted label: 0
True label: 0

Predicted label: 3
True label: 1
```

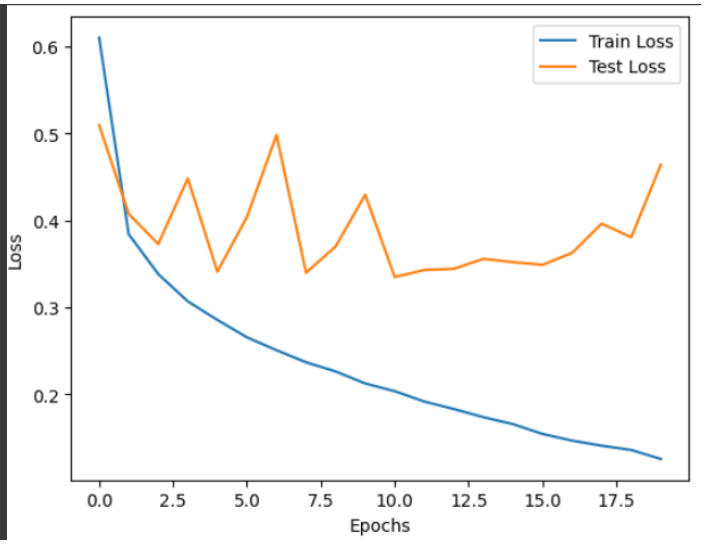
5-2:

In order to change the model in a way that it causes overfitting, I increase the sizes and remove dropouts, so the model will learn more details.

```
model = nn.Sequential(
    nn.Linear(input_size, 1024),
    nn.Tanh(),
    nn.Linear(1024, 512),
    nn.ReLU(),
    nn.Linear(512, 256),
    nn.ReLU(),
    nn.Linear(256, 128),
    nn.ReLU(),
    nn.Linear(128, out_size),
)
```

Then I wrote the training and evaluation code which is similar to the previous part so I'm skipping it in this file. But the results are as follow and the overfitting is clear:

```
Epoch [1/20], Train Loss: 0.6100, Test Loss: 0.5095
Epoch [2/20], Train Loss: 0.3841, Test Loss: 0.4075
Epoch [3/20], Train Loss: 0.3383, Test Loss: 0.3729
Epoch [4/20], Train Loss: 0.3070, Test Loss: 0.4482
Epoch [5/20], Train Loss: 0.2856, Test Loss: 0.3410
Epoch [6/20], Train Loss: 0.2656, Test Loss: 0.4040
Epoch [7/20], Train Loss: 0.2508, Test Loss: 0.4980
Epoch [8/20], Train Loss: 0.2369, Test Loss: 0.3399
Epoch [9/20], Train Loss: 0.2264, Test Loss: 0.3701
Epoch [10/20], Train Loss: 0.2126, Test Loss: 0.4295
Epoch [11/20], Train Loss: 0.2037, Test Loss: 0.3349
Epoch [12/20], Train Loss: 0.1918, Test Loss: 0.3430
Epoch [13/20], Train Loss: 0.1833, Test Loss: 0.3443
Epoch [14/20], Train Loss: 0.1739, Test Loss: 0.3558
Epoch [15/20], Train Loss: 0.1659, Test Loss: 0.3520
Epoch [16/20], Train Loss: 0.1546, Test Loss: 0.3490
Epoch [17/20], Train Loss: 0.1469, Test Loss: 0.3626
Epoch [18/20], Train Loss: 0.1411, Test Loss: 0.3962
Epoch [19/20], Train Loss: 0.1362, Test Loss: 0.3806
Epoch [20/20], Train Loss: 0.1259, Test Loss: 0.4640
```



5-3:

I used RandomHorizontalFlip(), Normalize() and RandomErasing() functions of v2 for data augmentation and also to have the images as torch, I used PILToTensor() and ConvertImageDtype(torch.float).

The RandomHorizontalFlip() randomly flips the image horizontally. The Normalize()

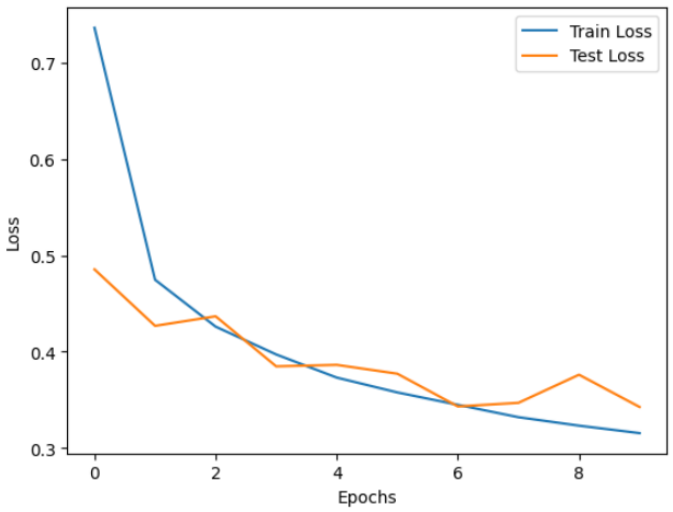
normalizes the image tensor by subtracting the mean and dividing by the standard deviation. This helps to ensure that the model is not overly sensitive to the specific values of the pixels and the RandomErasing() randomly erases a rectangular region of the image, simulating missing data or occlusions. This can help the model to learn to rely on more global features of the image.

```
from torchvision.transforms import v2 as transforms

#define data augmentation transforms
augmentation_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.PILToTensor(),
    transforms.ConvertImageDtype(torch.float),
    transforms.Normalize((0.1307,), (0.3081,)),
    transforms.RandomErasing(),
])
#apply data augmentation to the training dataset
augmented_trainset = datasets.FashionMNIST('./data', train=True, download=True, transform=augmentation_transform)
augmented_trainloader = torch.utils.data.DataLoader(augmented_trainset, batch_size=64, shuffle=True)
```

The results are as below and it is obvious that overfitting problem is solved:

```
Epoch [1/10], Train Loss: 0.7361, Test Loss: 0.4854
Epoch [2/10], Train Loss: 0.4746, Test Loss: 0.4268
Epoch [3/10], Train Loss: 0.4259, Test Loss: 0.4367
Epoch [4/10], Train Loss: 0.3971, Test Loss: 0.3847
Epoch [5/10], Train Loss: 0.3731, Test Loss: 0.3863
Epoch [6/10], Train Loss: 0.3577, Test Loss: 0.3771
Epoch [7/10], Train Loss: 0.3448, Test Loss: 0.3431
Epoch [8/10], Train Loss: 0.3320, Test Loss: 0.3469
Epoch [9/10], Train Loss: 0.3232, Test Loss: 0.3761
Epoch [10/10], Train Loss: 0.3154, Test Loss: 0.3426
```



5-4:

I choose L1 regularization and to implement it a part of code should be added at training and also evaluating code exactly after forward pass.

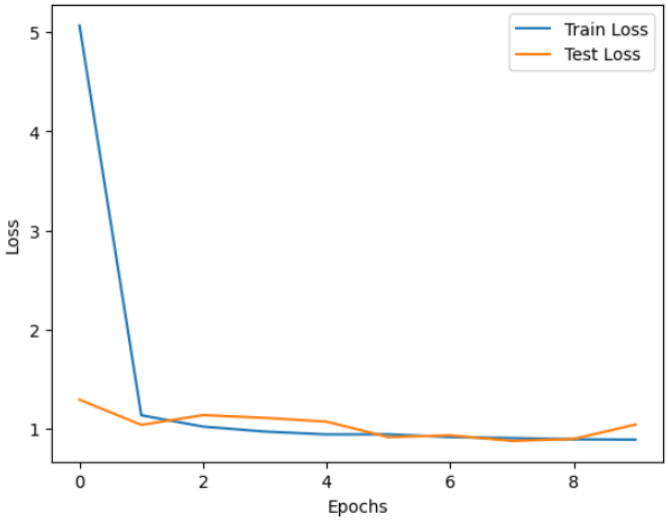
```
for images, labels in trainloader:
    images = images.view(images.shape[0], -1)
    optimizer.zero_grad()
    outputs = model(images)
    loss = criterion(outputs, labels)

    #L1 regularization
    l1_parameters = []
    for parameter in model.parameters():
        l1_parameters.append(parameter.view(-1))
    loss += l1_weight * torch.abs(torch.cat(l1_parameters)).sum()

    loss.backward()
    optimizer.step()
    train_loss += loss.item()
```

Because of adding regularization the value of loss increase but the overfitting has solved:

```
Epoch [1/10], Train Loss: 5.0644, Test Loss: 1.2916
Epoch [2/10], Train Loss: 1.1351, Test Loss: 1.0378
Epoch [3/10], Train Loss: 1.0203, Test Loss: 1.1363
Epoch [4/10], Train Loss: 0.9702, Test Loss: 1.1087
Epoch [5/10], Train Loss: 0.9419, Test Loss: 1.0703
Epoch [6/10], Train Loss: 0.9426, Test Loss: 0.9142
Epoch [7/10], Train Loss: 0.9144, Test Loss: 0.9323
Epoch [8/10], Train Loss: 0.9049, Test Loss: 0.8763
Epoch [9/10], Train Loss: 0.8922, Test Loss: 0.8959
Epoch [10/10], Train Loss: 0.8890, Test Loss: 1.0410
```

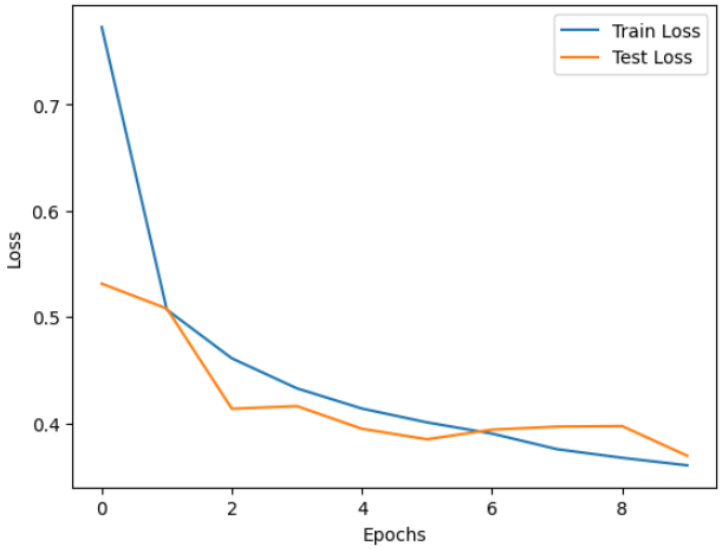


5-5:

In the last part, dropout, regularization and data augmentation are tested to combine with each other. Here is the results:

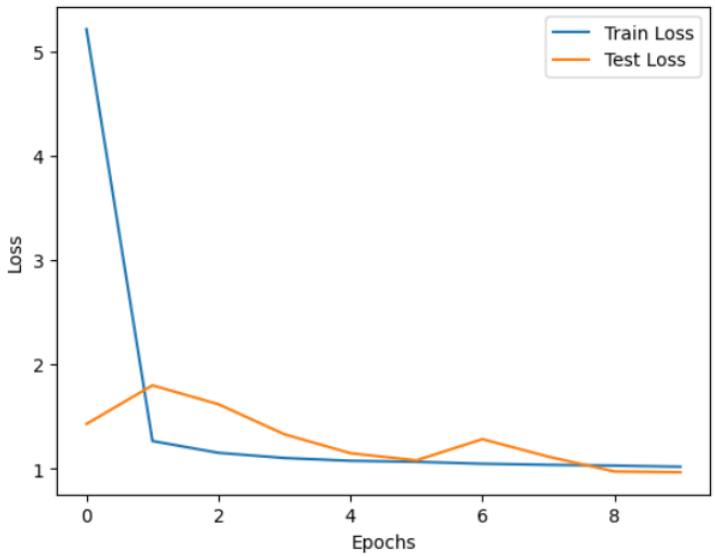
- Data augmentation and dropout:

```
Epoch [1/10], Train Loss: 0.7729, Test Loss: 0.5313
Epoch [2/10], Train Loss: 0.5068, Test Loss: 0.5079
Epoch [3/10], Train Loss: 0.4610, Test Loss: 0.4135
Epoch [4/10], Train Loss: 0.4327, Test Loss: 0.4160
Epoch [5/10], Train Loss: 0.4137, Test Loss: 0.3946
Epoch [6/10], Train Loss: 0.4006, Test Loss: 0.3848
Epoch [7/10], Train Loss: 0.3901, Test Loss: 0.3938
Epoch [8/10], Train Loss: 0.3754, Test Loss: 0.3967
Epoch [9/10], Train Loss: 0.3673, Test Loss: 0.3972
Epoch [10/10], Train Loss: 0.3602, Test Loss: 0.3692
```



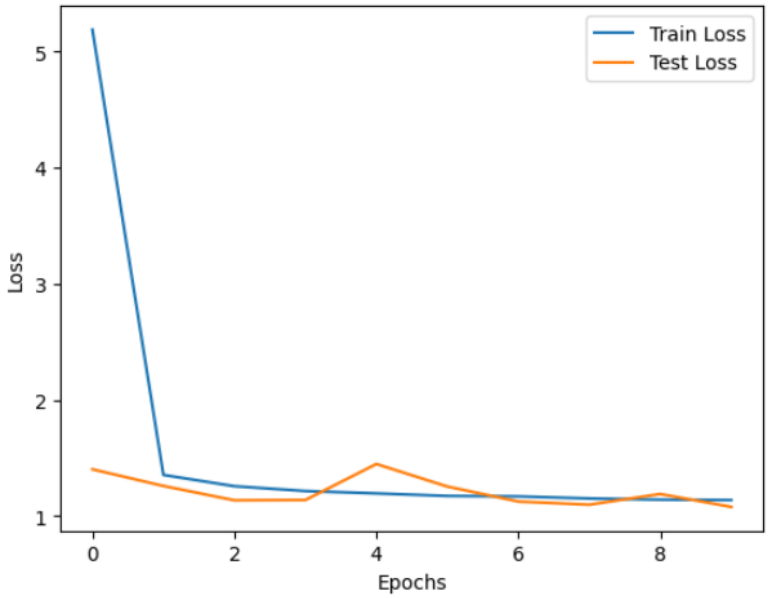
- Data augmentation and regularization:

```
Epoch [1/10], Train Loss: 5.2116, Test Loss: 1.4281
Epoch [2/10], Train Loss: 1.2646, Test Loss: 1.7983
Epoch [3/10], Train Loss: 1.1515, Test Loss: 1.6170
Epoch [4/10], Train Loss: 1.1021, Test Loss: 1.3293
Epoch [5/10], Train Loss: 1.0751, Test Loss: 1.1479
Epoch [6/10], Train Loss: 1.0652, Test Loss: 1.0792
Epoch [7/10], Train Loss: 1.0472, Test Loss: 1.2826
Epoch [8/10], Train Loss: 1.0364, Test Loss: 1.1150
Epoch [9/10], Train Loss: 1.0291, Test Loss: 0.9725
Epoch [10/10], Train Loss: 1.0183, Test Loss: 0.9659
```



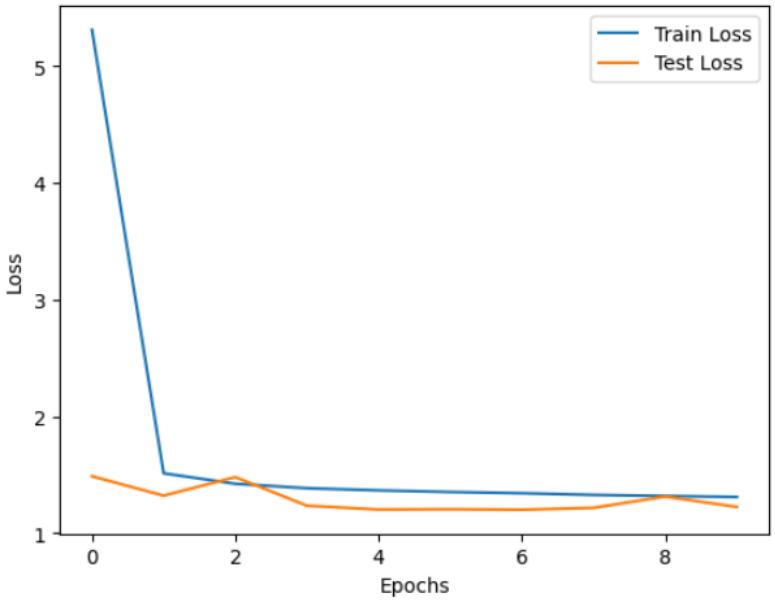
- Dropout and regularization:

```
Epoch [1/10], Train Loss: 5.1873, Test Loss: 1.4018
Epoch [2/10], Train Loss: 1.3533, Test Loss: 1.2585
Epoch [3/10], Train Loss: 1.2563, Test Loss: 1.1350
Epoch [4/10], Train Loss: 1.2147, Test Loss: 1.1374
Epoch [5/10], Train Loss: 1.1945, Test Loss: 1.4476
Epoch [6/10], Train Loss: 1.1724, Test Loss: 1.2534
Epoch [7/10], Train Loss: 1.1685, Test Loss: 1.1245
Epoch [8/10], Train Loss: 1.1507, Test Loss: 1.0970
Epoch [9/10], Train Loss: 1.1405, Test Loss: 1.1891
Epoch [10/10], Train Loss: 1.1368, Test Loss: 1.0780
```



- All three:

```
Epoch [1/10], Train Loss: 5.3097, Test Loss: 1.4868
Epoch [2/10], Train Loss: 1.5122, Test Loss: 1.3208
Epoch [3/10], Train Loss: 1.4225, Test Loss: 1.4796
Epoch [4/10], Train Loss: 1.3845, Test Loss: 1.2346
Epoch [5/10], Train Loss: 1.3660, Test Loss: 1.2021
Epoch [6/10], Train Loss: 1.3522, Test Loss: 1.2043
Epoch [7/10], Train Loss: 1.3415, Test Loss: 1.2004
Epoch [8/10], Train Loss: 1.3271, Test Loss: 1.2165
Epoch [9/10], Train Loss: 1.3178, Test Loss: 1.3124
Epoch [10/10], Train Loss: 1.3096, Test Loss: 1.2243
```



In my opinion combining them in all four ways solved the overfitting problem but the results from combination of data augmentation and dropout seems to be the best among them.