SREYASHI SAHA Matriculation Number: 1747271

Importing all the libraries required to implement the provided architechture on CIFAR10 data set.

In [54]:
```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import torch.nn.functional as F
import matplotlib.pyplot as plt
from datetime import datetime
import gc
import numpy as np
```

In [55]:
```
import torch
import tensorboard
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter()
```

In [56]:
```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
device
```

Out[56]: 'cpu'

Since CUDA is not installed on my device, I check if GPU is availabe and since it is not therefore, my program is trained on the CPU.

Implementing the basic structure provided to us:

```
class LeNet5(nn.Module):

    def __init__(self, n_classes):
        super(LeNet5, self).__init__()

        self.feature_extractor = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=6, kernel_size=5, stride=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,stride=2),
            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=16, out_channels=120, kernel_size=5, stride=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=6,stride=1)
        )

        self.classifier = nn.Sequential(
            nn.Linear(in_features=120, out_features=84),
            nn.ReLU(),
            nn.Linear(in_features=84, out_features=42),
            nn.ReLU(),
            nn.Linear(in_features=42, out_features=n_classes),
        )


    def forward(self, x):
        x = self.feature_extractor(x)
        x = torch.flatten(x, 1)
        logits = self.classifier(x)
        probs = F.softmax(logits, dim=1)
        return logits, probs
```

Architechture On CIFAR10¶

The above architecture has nine layers. Three convolutional layers, two subsampling layers, three fully linked layers make up the layer composition, and a softmax layer.

The input layer, which is the initial layer, is typically not thought of as part of the network because nothing is learned there. The following layer receives photos that are 32x32x3 in size since the input layer is designed to accept images of that size and the number of channels.

The naming convention used by the authors to implement LeNet is as follows:

- Cx — convolution layer,
- Sx — subsampling (pooling) layer,
- Fx — fully-connected layer,
- x — index of the layer

The Formula used to get the output dimensions after each layer is:
(W−F+2P)/S+1 ,¶
where W is the input height/width, F is the filter/kernel size, P is the padding, and S is the stride.
Now let's scheme through the 9 layers to get a better understanding of the Architechture:

1) Layer1(C1): The first convolutional layer has six 5 * 5 kernels with a stride of 1. The window containing the weight values used during the convolution of the weight values with the input values is referred to as the kernel/filter. The local receptive field size of each unit or neuron within a convolutional layer is also represented by the number 5 * 5. The output of this layer is of size 28 * 28 * 6 given the input size (32 * 32 * 3).

2) Layer2(S1): A pooling/subsampling layer with a stride of 2 and six 2 * 2 kernels. Compared to the more conventional max/average pooling layers, the subsampling layer in the original architecture was a little more complicated. A unit in S2 has four inputs, which are added, multiplied by a trainable coefficient, multiplied by a trainable bias, and added. A sigmoidal or tanh or ReLU activation functions are applied to the outcome. The input to this layer is downsampled (14 * 14 * 6) due to non-overlapping receptive fields, which is also referred to as downsampling.

3) Layer3(C2): The second convolutional layer with the same configuration as the first one, but with 16 filters instead of 6. The output of this layer is 10 * 10 * 16.

4) Layer4(C3): 120 5 * 5 kernels are used in the final convolutional layer. Given that the kernels in this layer are 5 * 5 and the input is 5 * 5 * 16, the output is 6 * 6 * 120.

5) Layer5(S2): The second pooling layer. The logic is identical to the previous one, but with 120 filters instead of 6 and with stride=1 and kernel size=6. The output of this layer is of size 1 * 1 * 120.

6) Layer6(F6): The first fully connected layer is Layer 6 (F6), which takes the input of 120 features and returns 84 features.

7) Layer7(F7): The second fully connected layer is Layer 6 (F6), which takes the input of 120 features and returns 84 features.

8) Layer8(F8): The last dense layer, outputs 10 classes after taking 42 input features.

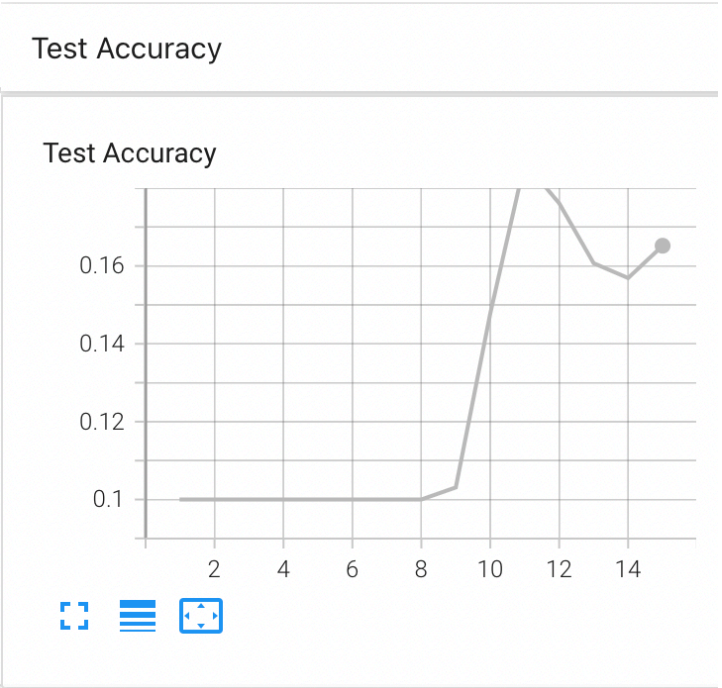
9) Layer9: Next to the last fully connected layer I have applied a softmax layer. Applying Softmax function to an n-dimensional input Tensor rescales them so that the elements of the n-dimensional output Tensor lie in the range [0,1] and sum to 1.

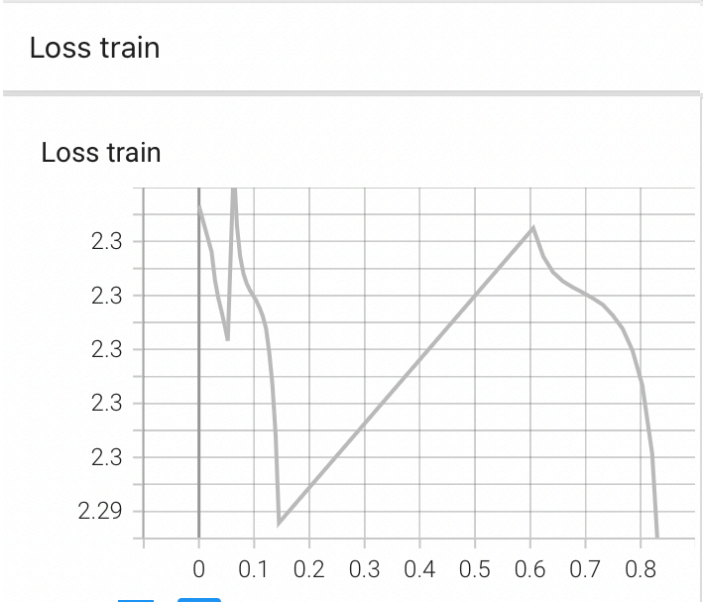
Here we implement the above model and do not apply any kind of Data Augmentation or Transformation. So we split the training dataset into two halves on the basis of index and train out model and find out the respective losses and accuracy. Here I have kept the batch size to be 10.

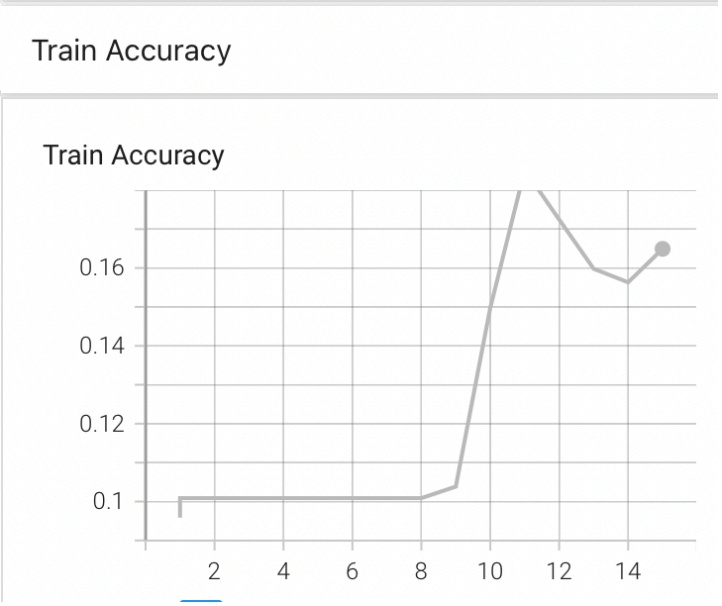
```
set1 = list(range(0,len(train_data),2))
set2 = list(range(1,len(train_data),2))
train_1 = torch.utils.data.Subset(train_data,set1)
train_2 = torch.utils.data.Subset(train_data,set2)
```

```
train_dataloader1 = torch.utils.data.DataLoader(train_1, batch_size=10, shuffle=True, num_workers=2)
train_dataloader2 = torch.utils.data.DataLoader(train_2, batch_size=10, shuffle=True, num_workers=2)
test_dataloader = torch.utils.data.DataLoader(test_data, batch_size=10, shuffle=True, num_workers=2)
```

After implementing this model if we plot the graphs for training and testing loss and accuracy respectively, without applying data augmentation and normalisation, then we get the following graphs.







Exercise 1: Normalization Effect (CNN)

Data augmentation is the process of artificially enlarging your training dataset using carefully chosen transforms.
When done correctly, data augmentation strengthens our trained models and enables them to achieve higher accuracy without the need for a larger dataset.
When compared to another identical model learning on the same dataset for the same amount of epochs, data augmentation can prevent over-fitting.
Random flipping and random cropping are two highly helpful transforms of this kind that are frequently employed in computer vision.

```python
# convert data to a normalized torch.FloatTensor
# transform = transforms.Compose([transforms.ToTensor()])
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize([0, 0, 0], [1, 1, 1])
])

test_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0, 0, 0], [1, 1, 1])
])

# declaring the training and test datasets
train_data = datasets.CIFAR10('data', train=True,download=True,transform = train_transform)
test_data = datasets.CIFAR10('data', train=False,download=True,transform = test_transform)

```

```
Files already downloaded and verified
Files already downloaded and verified
```

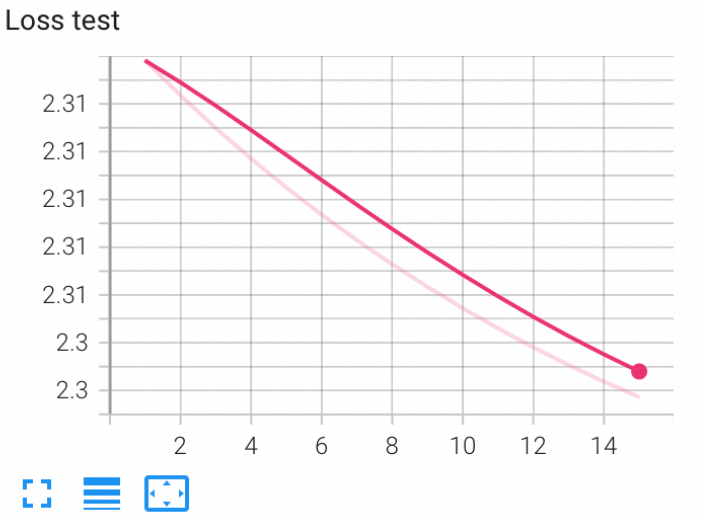
In torchvision, the random horizontal flip and random vertical flip transforms can be used to achieve random flipping, while the random crop transform can be used to achieve random cropping.
RandomHorizontalFlip without arguments will simply randomly flip the image horizontally with probability 0.5.
RandomCrop takes a more detailed set of parameters. Firstly, the size parameter is either a sequence or integer indicating the output size of RandomCrop. If an integer is provided, the output will be a square crop with side length equal to the integer provided. The padding parameter indicates how much padding or white space we want to add to the edges of the image before cropping. If an integer is provided, as in second train transform, then equal padding is added to all sides.
It is important to keep in mind that when generating a Compose transform, both RandomHorizontalFlip and RandomCrop must come before the ToTensor transform because they are intended to act on images rather than tensors.

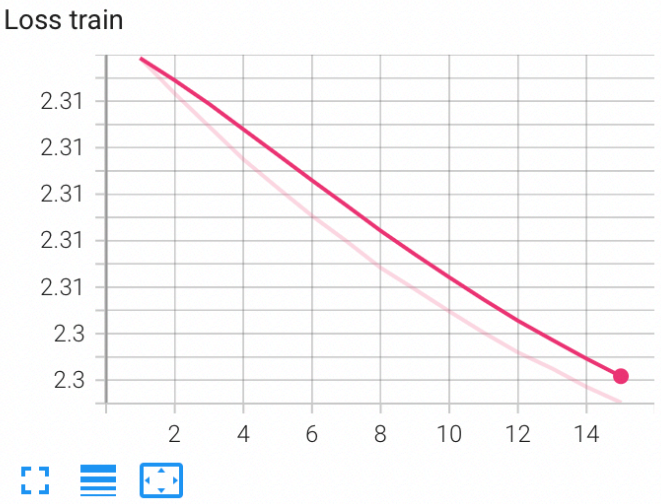
Here first I just apply data augmentation and plot the graphs. Next I just normalise my data and plot the graphs and finally I apply both data augmentation and normalisation to plot the final graphs for the above model.

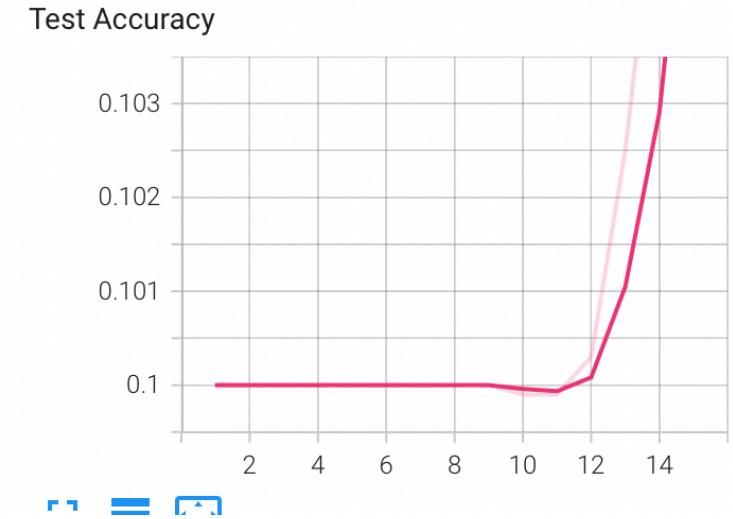
Step1: Apply Data Augmentation Only

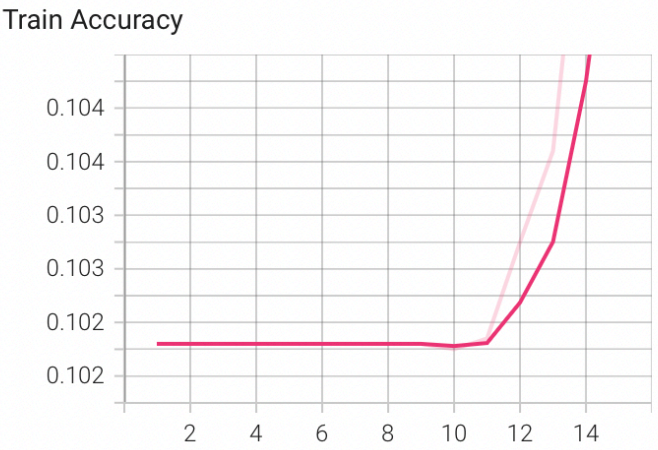
```python
# convert data to a normalized torch.FloatTensor
# transform = transforms.Compose([transforms.ToTensor()])
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
])

test_transform = transforms.Compose([
    transforms.ToTensor(),
])
```



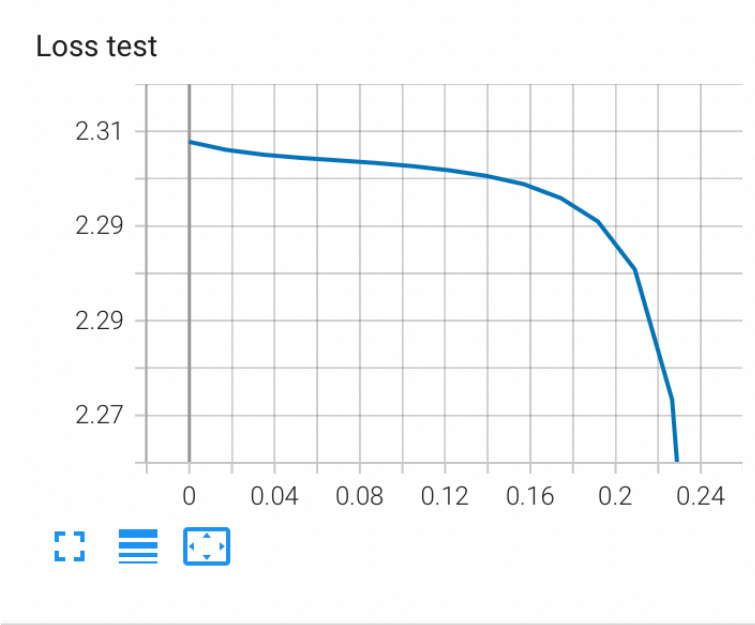


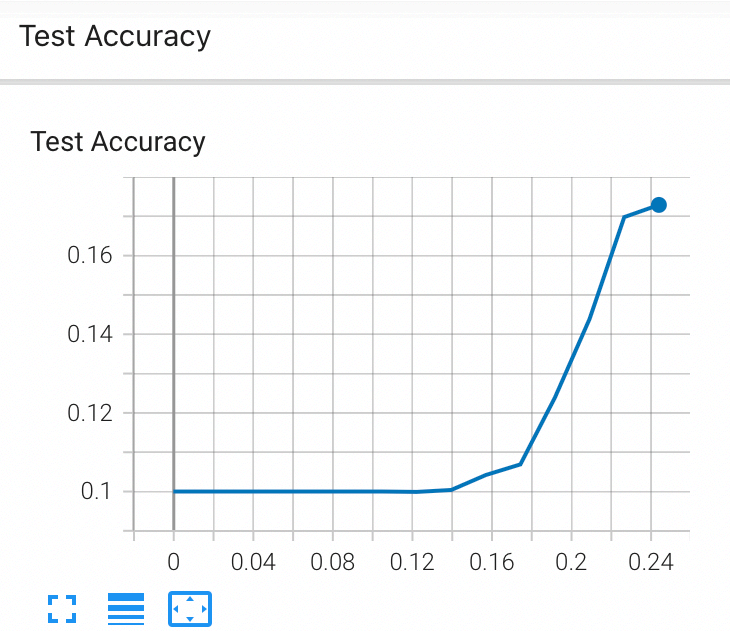


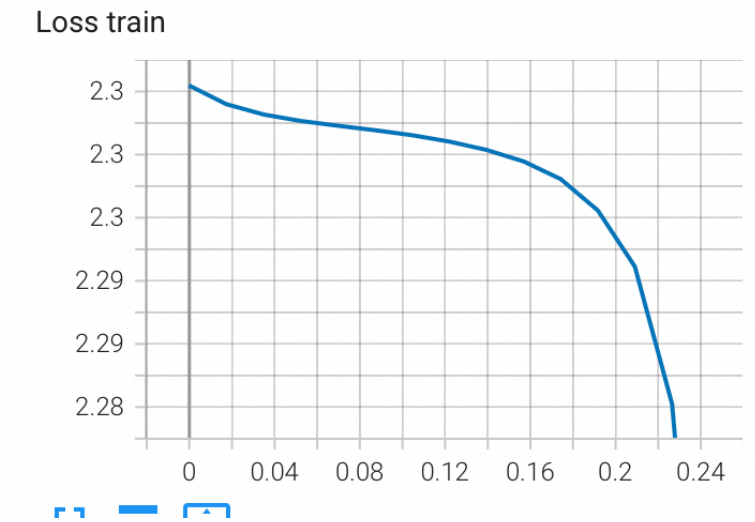


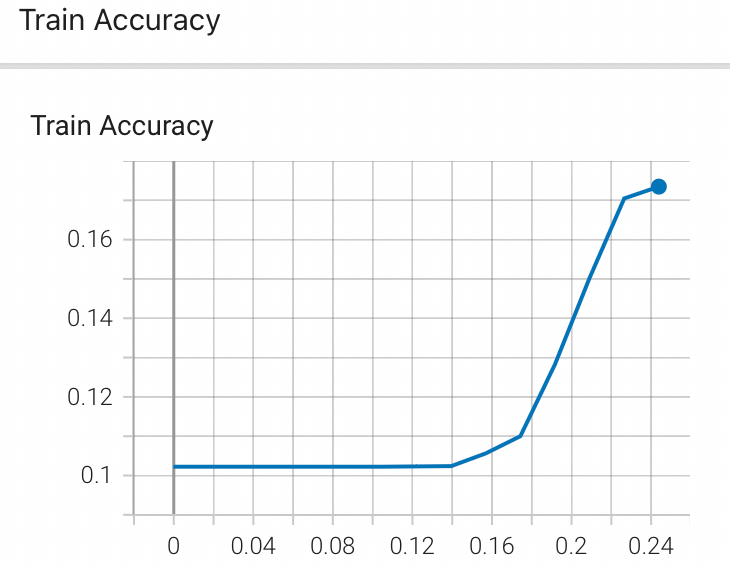
Step2: With Data Normalisation Only

```
train_transform = transforms.Compose([
#     transforms.RandomHorizontalFlip(p=0.5),
#     transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize([0, 0, 0], [1, 1, 1])
])
```



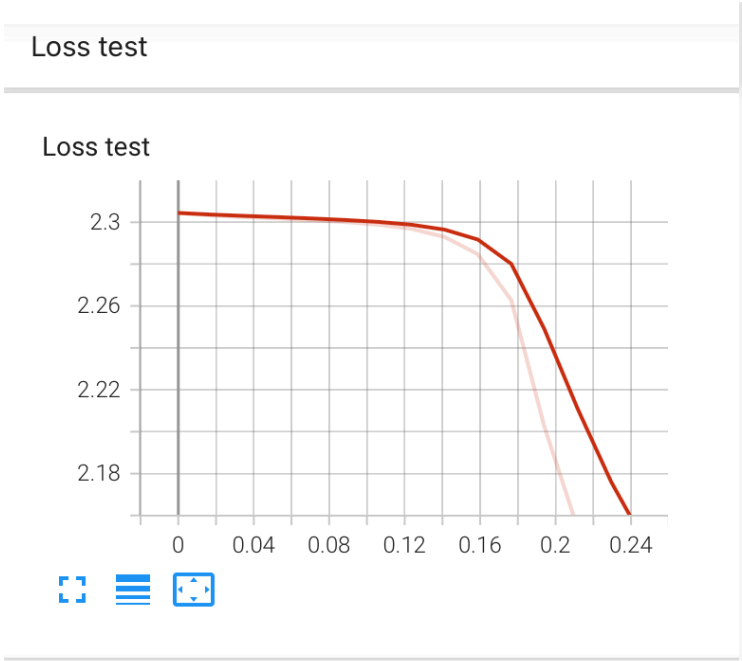


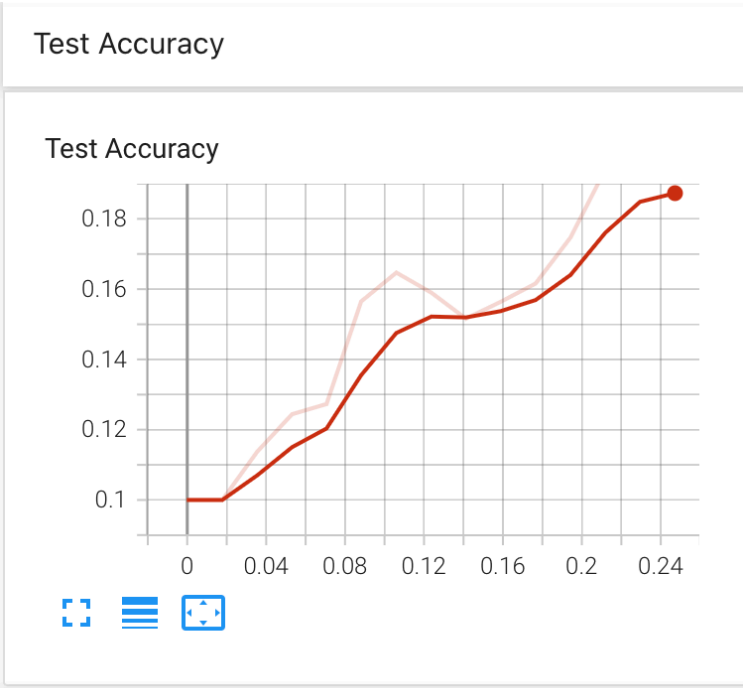


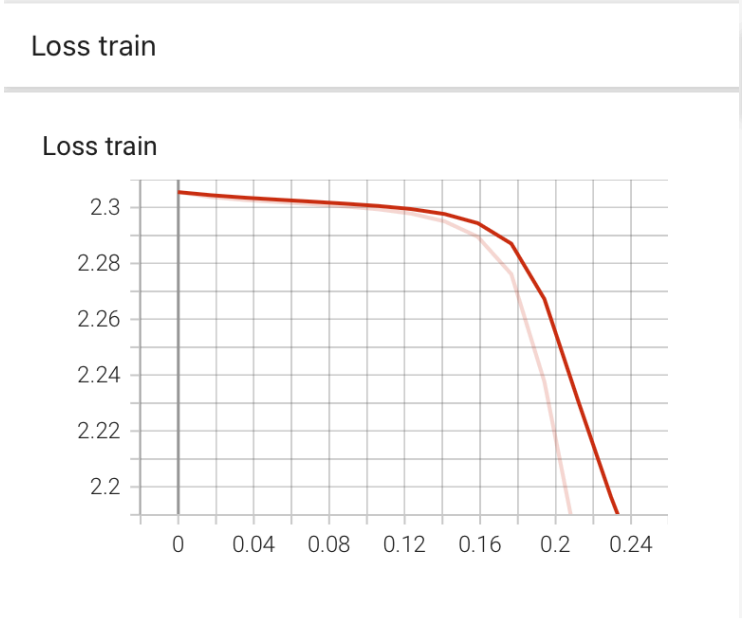


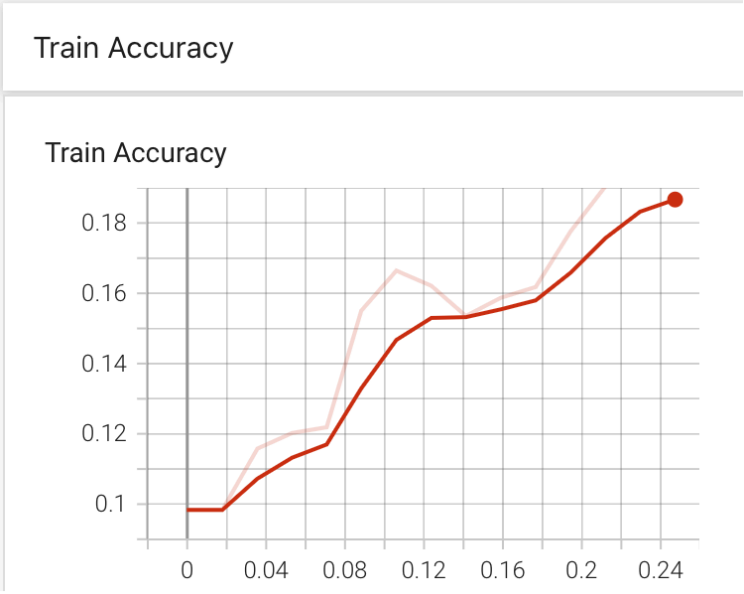
Step3: With Both Data Augmentation and Data Normalisation

The loss function that I have used in my model is **L1Loss()** provided by the **torch.nn** module. It creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y. Both the actual and predicted values are torch tensors having the same number of elements. Both the tensors may have any number of dimensions. This function returns a tensor of a scalar value. The loss function is used to optimize a deep neural network by minimizing the loss.



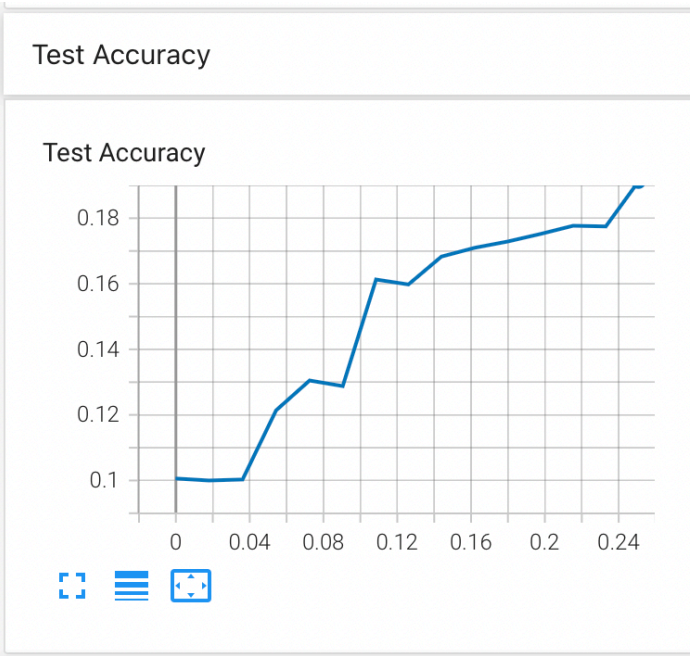


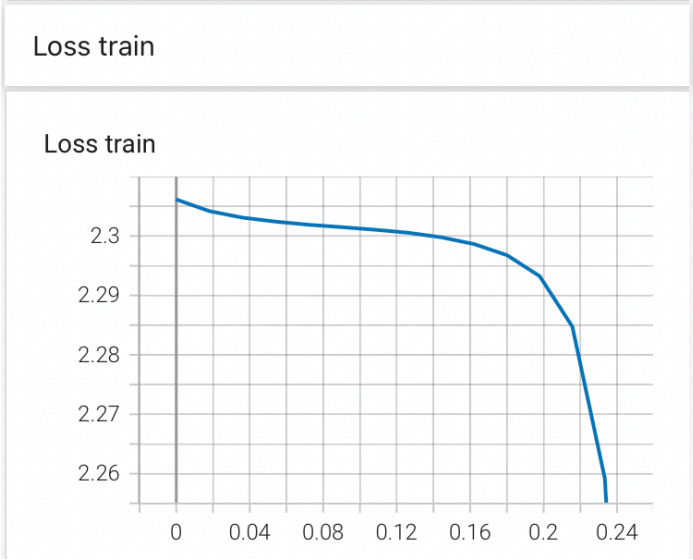


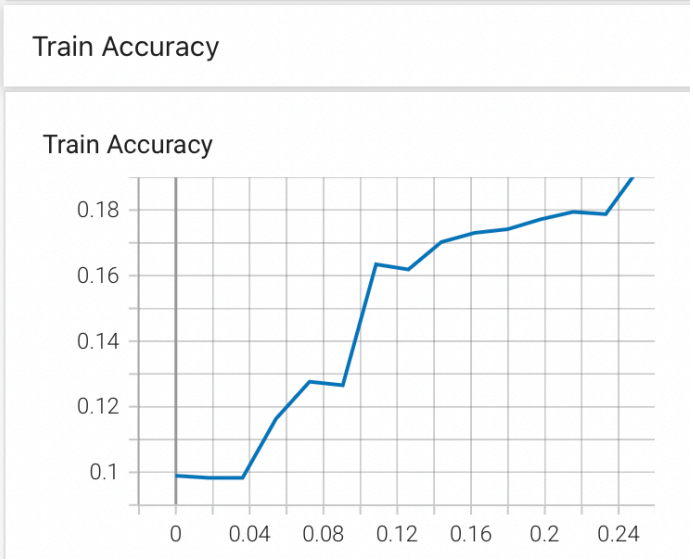


Exercise 2: Network Regularization (CNN)

Dropout is a machine learning technique where you remove (or "drop out") units in a neural net to simulate training large numbers of architectures simultaneously. Importantly, dropout can drastically reduce the chance of overfitting during training. An unregularized network quickly overfits on the training dataset. Training with two dropout layers with a dropout probability of 25% prevents model from overfitting. This brings down the training accuracy, which means a regularized network has to be trained longer.
Dropout improves the model generalization. Even though the training accuracy is lower than the unregularized network, the overall validation accuracy has improved. This accounts for a lower generalization error.

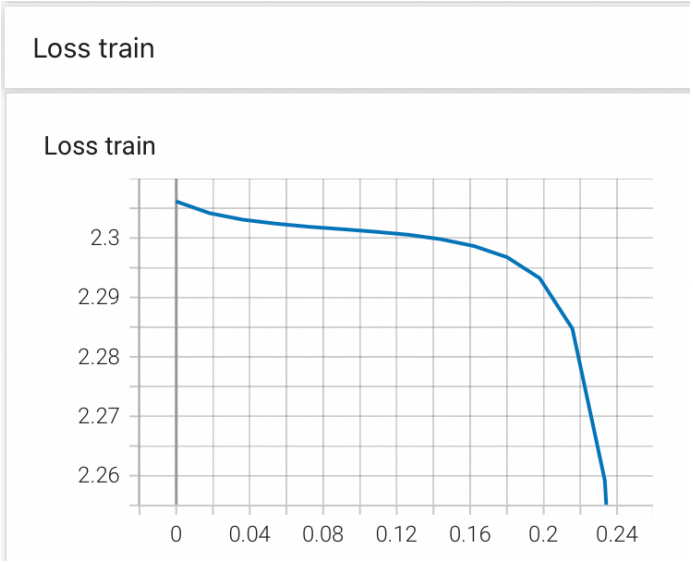


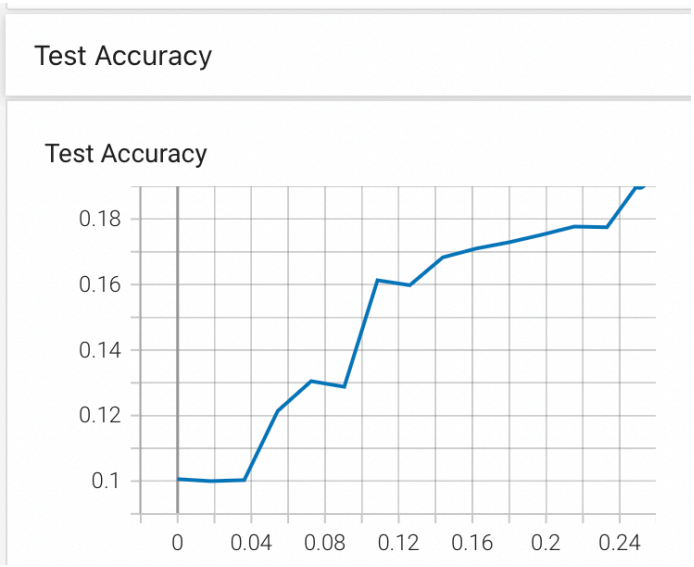


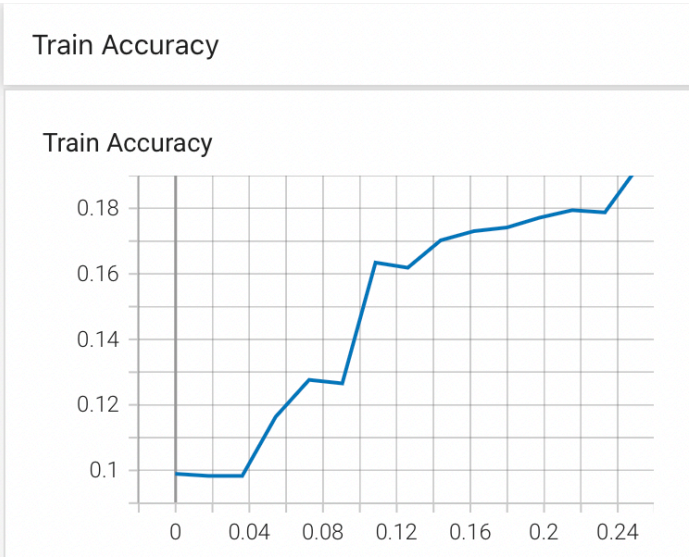


Exercise 3: Optimizers (CNN)

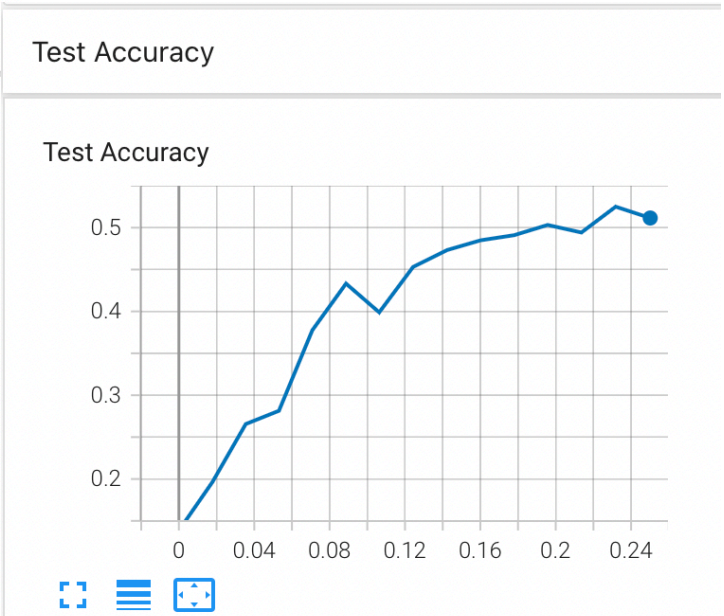
OPTIMIZER: SGD Learning Rate : 0.001

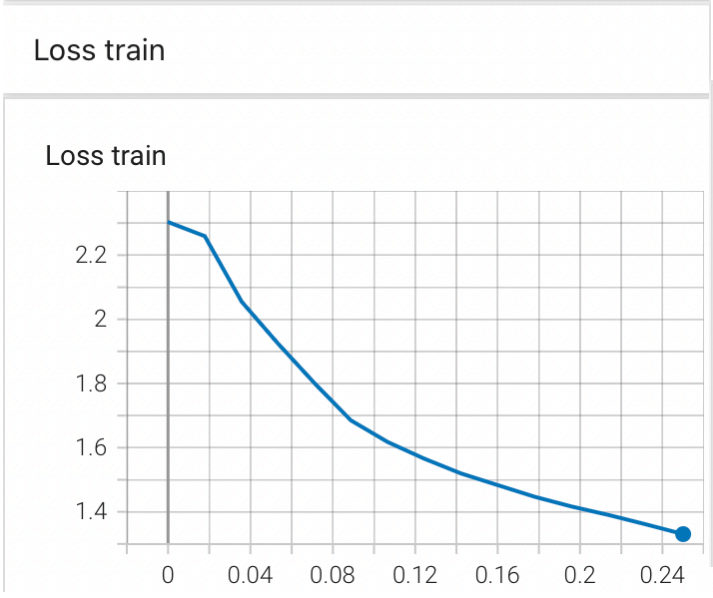


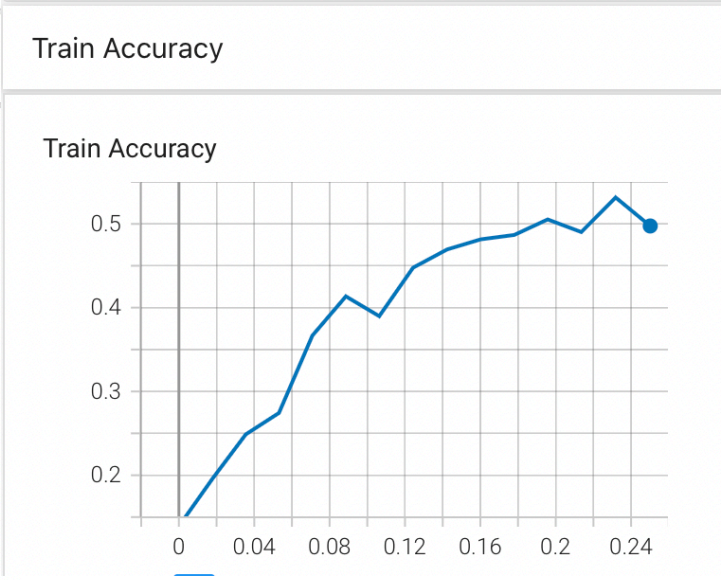




OPTIMIZER: SGD Learning Rate: 0.01

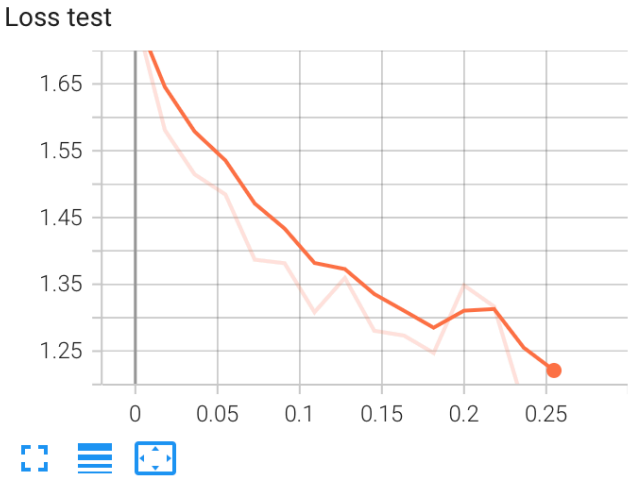




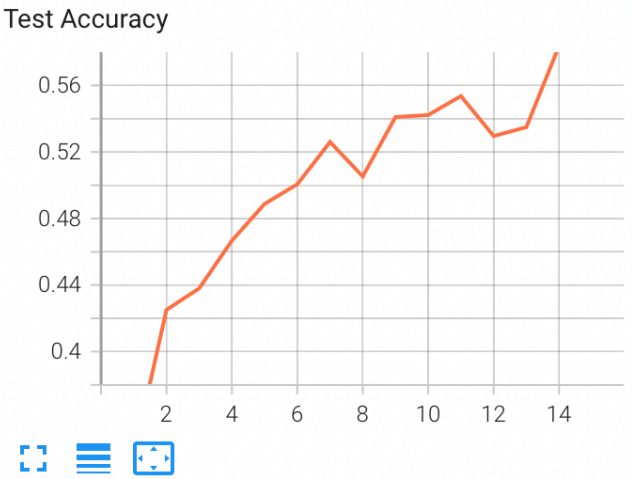


OPTIMIZER: ADAM Learning Rate: 0.001

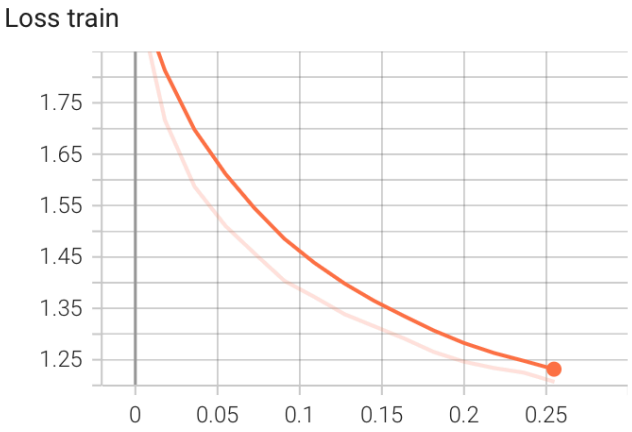
Loss test



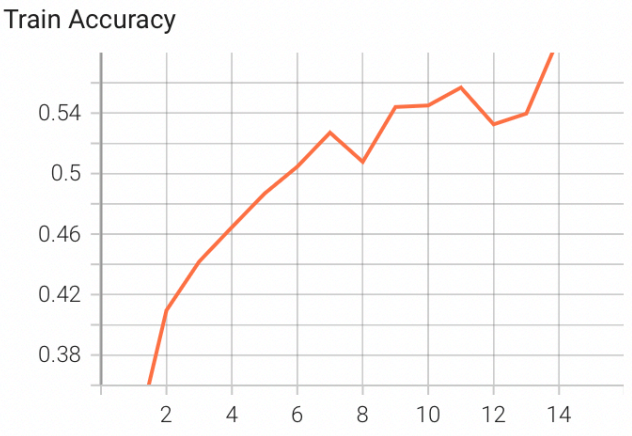
Test Accuracy



Loss train

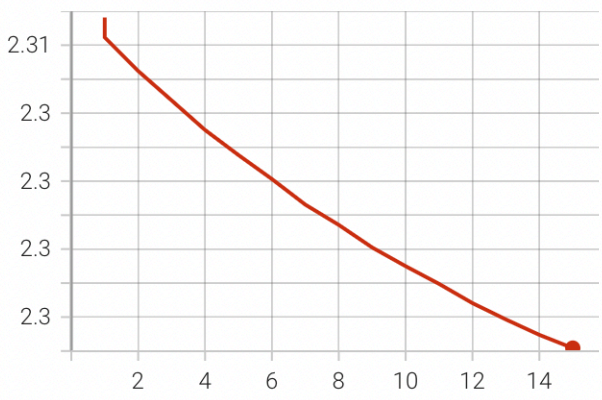


Train Accuracy

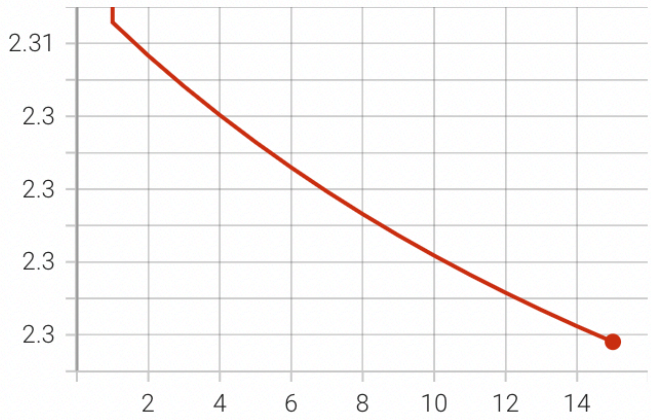


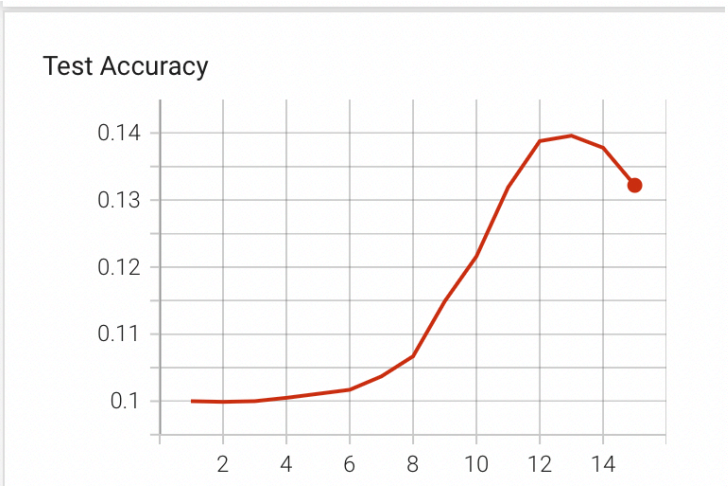
OPTIMIZER: ADAM Learning Rate: 0.01

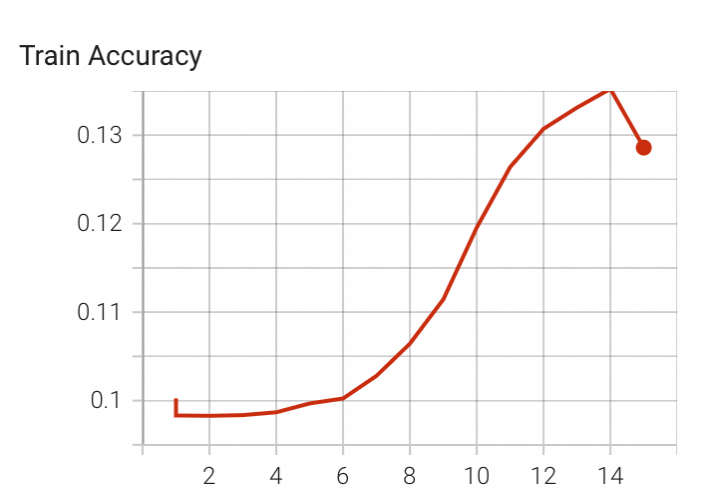
Loss train



Loss test







Other Functions Used to train, test , plot and find the accuracies:

```python
def train(train_loader, model, criterion, optimizer, device):

    model.train()
    running_loss = 0

    for X, y_true in train_loader:

        optimizer.zero_grad()

        X = X.to(device)
        y_true = y_true.to(device)

        # Forward pass
        y_hat, _ = model(X)
        loss = criterion(y_hat, y_true)
        running_loss += loss.item() * X.size(0)

        # Backward pass
        loss.backward()
        optimizer.step()

    Loss = running_loss / len(train_loader.dataset)
    return model, optimizer, Loss
```

The model is in training mode (model.train()) for the training phase, and I have also zeroed out the gradients for each batch. Additionally, I find out the running loss throughout the training phase. I then return the model which I'll be implementing, in this case it is the model provided in the question, along with the optimizer used i.e. SGD in my case and the loss obtained after each epoch.

The main distinction between the testing and training functions is the absence of the actual learning step (the backward pass). Here I am only utilizing the model for evaluation using the model.eval keyword (). Gradients are not a concern because, as in the following method, I have disabled them during the testing phase. In the training loop, I will finally integrate them all.

```
def test(test_loader, model, criterion, device):
    model.eval()
    running_loss = 0

    for X, y_true in test_loader:

        X = X.to(device)
        y_true = y_true.to(device)

        # Forward pass and record loss
        y_hat, _ = model(X)
        loss = criterion(y_hat, y_true)
        running_loss += loss.item() * X.size(0)

    Loss = running_loss / len(test_loader.dataset)

    return model, Loss
```

The function below is used to plot the training and testing losses. First I have converted the losses obtained into array to be later used in plotting. I have set a figure size and then I have plot the training and testing loss in blue and red respectively. Here my X-axis is the number of epochs and the Y-axis is the loss that I calculate.

```
def plot_loss(train_losses, valid_losses):

    # change the style of the plots to seaborn
    plt.style.use('seaborn')

    train_losses = np.array(train_losses)
    valid_losses = np.array(valid_losses)

    fig, ax = plt.subplots(figsize = (8, 4.5))

    ax.plot(train_losses, color='blue', label='Training loss')
    ax.plot(valid_losses, color='red', label='Testing loss')
    ax.set(title="Loss over epochs",
           xlabel='Epoch',
           ylabel='Loss')
    ax.legend()
    fig.show()

    # change the plot style to default
    plt.style.use('default')
```

In the get_accuracy function I find the accuracy of the predictions of the entire data loader. Here we basically find the predicted target values and the torch.max function returns me the value which is closest to 1 from each tensor input. I then take the sum of all those predicted values which are equal to the actual target value from the training or testing dataset. Then I return the average of the total number of correctly predicted value using which we calculate the training or testing accuracy later.

```python
def get_accuracy(model, data_loader, device):
    correct_pred = 0
    n = 0

    with torch.no_grad():
        model.eval()
        for X, y_true in data_loader:

            X = X.to(device)
            y_true = y_true.to(device)

            _, y_prob = model(X)
#             print("Probabilities",y_prob)
            _, predicted_labels = torch.max(y_prob, 1)

            n += y_true.size(0)
            correct_pred += (predicted_labels == y_true).sum()

    return correct_pred.float() / n
```

In the function below I first pass as parameters the model to implement, the optimizer to use, the train dataset, test dataset, number of epochs, the device on which the computation is done. Inside the function I create two lists where I store the loss values obtained from the train and test datasets. Here I do not update the weights for test dataset, hence, I use torch.no_grad() function while appending the test loss obtained from the test function. Then for each epoch I calculate the train loss, test loss along with the accuracy and print it. I also plot the graph for the training and testing losses obtained. I also use TensorBoard to view the accuracy and loss graphs.

```python
def evaluation(model, criterion, optimizer, train_loader, test_loader, epochs, device, print_every=1):

    # set objects for storing metrics
    train_losses = []
    test_losses = []

    # Train model
    for epoch in range(0, epochs):

        # training
        model, optimizer, train_loss = train(train_loader, model, criterion, optimizer, device)
        writer.add_scalar("Loss train", train_loss, epoch+1)
        train_losses.append(train_loss)

        # validation
        with torch.no_grad():
            model, test_loss = test(test_loader, model, criterion, device)
            writer.add_scalar("Loss test", test_loss, epoch+1)
            test_losses.append(test_loss)

        if epoch % print_every == (print_every - 1):

            train_acc = get_accuracy(model, train_loader, device=device)
            writer.add_scalar("Train Accuracy", train_acc, epoch+1)
            test_acc = get_accuracy(model, test_loader, device=device)
            writer.add_scalar("Test Accuracy", test_acc, epoch+1)
            print(f'{datetime.now().time().replace(microsecond=0)} --- '
                  f'Epoch: {epoch}\t'
                  f'Train loss: {train_loss:.4f}\t'
                  f'Test loss: {test_loss:.4f}\t'
                  f'Train accuracy: {100 * train_acc:.2f}\t'
                  f'Test accuracy: {100 * test_acc:.2f}')


    plot_loss(train_losses, test_losses)

    return model, optimizer, (train_losses,test_losses)
```

Finally I call the models and the functions to get the losses and accuracies:

```
model = LeNet5(10).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
criterion = nn.L1Loss()
```

```
model, optimizer, _ = evaluation(model, criterion, optimizer, train_dataloader1, test_dataloader, 15, device)
```

```
00:09:51 --- Epoch: 0   Train loss: 2.3061      Test loss: 2.3041       Train accuracy: 9.90    Test accuracy: 10.0
6
00:10:55 --- Epoch: 1   Train loss: 2.3042      Test loss: 2.3029       Train accuracy: 9.83    Test accuracy: 10.0
0
00:12:01 --- Epoch: 2   Train loss: 2.3031      Test loss: 2.3023       Train accuracy: 9.83    Test accuracy: 10.0
3
00:13:06 --- Epoch: 3   Train loss: 2.3024      Test loss: 2.3019       Train accuracy: 11.64   Test accuracy: 12.1
4
00:14:12 --- Epoch: 4   Train loss: 2.3019      Test loss: 2.3015       Train accuracy: 12.76   Test accuracy: 13.0
5
00:15:16 --- Epoch: 5   Train loss: 2.3015      Test loss: 2.3012       Train accuracy: 12.66   Test accuracy: 12.8
8
00:16:21 --- Epoch: 6   Train loss: 2.3010      Test loss: 2.3008       Train accuracy: 16.34   Test accuracy: 16.1
3
00:17:25 --- Epoch: 7   Train loss: 2.3005      Test loss: 2.3002       Train accuracy: 16.19   Test accuracy: 15.9
8
00:18:29 --- Epoch: 8   Train loss: 2.2998      Test loss: 2.2993       Train accuracy: 17.02   Test accuracy: 16.8
3
00:19:34 --- Epoch: 9   Train loss: 2.2986      Test loss: 2.2978       Train accuracy: 17.30   Test accuracy: 17.1
0
00:20:38 --- Epoch: 10  Train loss: 2.2968      Test loss: 2.2952       Train accuracy: 17.42   Test accuracy: 17.2
```

```
model = LeNet5(10).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.L1Loss()
```

```
model, optimizer, _ = evaluation(model, criterion, optimizer, train_dataloader1, test_dataloader, 15, device)
```

```
22:11:57 --- Epoch: 0   Train loss: 1.9739      Test loss: 1.7532       Train accuracy: 31.82   Test accuracy: 33.7
2
22:13:01 --- Epoch: 1   Train loss: 1.7164      Test loss: 1.5809       Train accuracy: 40.96   Test accuracy: 42.5
1
22:14:08 --- Epoch: 2   Train loss: 1.5872      Test loss: 1.5148       Train accuracy: 44.18   Test accuracy: 43.8
0
22:15:14 --- Epoch: 3   Train loss: 1.5105      Test loss: 1.4845       Train accuracy: 46.45   Test accuracy: 46.6
7
22:16:19 --- Epoch: 4   Train loss: 1.4565      Test loss: 1.3869       Train accuracy: 48.68   Test accuracy: 48.8
8
22:17:24 --- Epoch: 5   Train loss: 1.4034      Test loss: 1.3816       Train accuracy: 50.46   Test accuracy: 50.0
6
22:18:30 --- Epoch: 6   Train loss: 1.3724      Test loss: 1.3083       Train accuracy: 52.71   Test accuracy: 52.6
0
22:19:36 --- Epoch: 7   Train loss: 1.3384      Test loss: 1.3596       Train accuracy: 50.78   Test accuracy: 50.5
```

```
model = LeNet5(10).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
criterion = nn.L1Loss()
```

```
model, optimizer, _ = evaluation(model, criterion, optimizer, train_dataloader1, test_dataloader, 15, device)
```

```
22:32:25 --- Epoch: 0   Train loss: 2.3031      Test loss: 2.3017       Train accuracy: 13.92   Test accuracy: 13.8
6
22:33:29 --- Epoch: 1   Train loss: 2.2592      Test loss: 2.0837       Train accuracy: 19.60   Test accuracy: 19.6
5
22:34:32 --- Epoch: 2   Train loss: 2.0548      Test loss: 1.9757       Train accuracy: 24.88   Test accuracy: 26.5
6
22:35:36 --- Epoch: 3   Train loss: 1.9252      Test loss: 1.9172       Train accuracy: 27.43   Test accuracy: 28.1
4
22:36:40 --- Epoch: 4   Train loss: 1.8015      Test loss: 1.6834       Train accuracy: 36.67   Test accuracy: 37.7
6
22:37:44 --- Epoch: 5   Train loss: 1.6852      Test loss: 1.5724       Train accuracy: 41.35   Test accuracy: 43.3
2
22:38:47 --- Epoch: 6   Train loss: 1.6177      Test loss: 1.6351       Train accuracy: 38.98   Test accuracy: 39.8
9
22:39:52 --- Epoch: 7   Train loss: 1.5657      Test loss: 1.4824       Train accuracy: 44.77   Test accuracy: 45.3
1
22:40:57 --- Epoch: 8   Train loss: 1.5197      Test loss: 1.4540       Train accuracy: 46.95   Test accuracy: 47.3
3
22:42:01 --- Epoch: 9   Train loss: 1.4825      Test loss: 1.4097       Train accuracy: 48.15   Test accuracy: 48.4
9
```

```
model = LeNet5(10).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
criterion = nn.L1Loss()
model, optimizer, _ = evaluation(model, criterion, optimizer, train_dataloader1, test_dataloader, 15, device)
```

```
23:00:57 --- Epoch: 0   Train loss: 2.3049      Test loss: 2.3053       Train accuracy: 10.17   Test accuracy: 10.0
0
23:02:02 --- Epoch: 1   Train loss: 2.3048      Test loss: 2.3054       Train accuracy: 10.17   Test accuracy: 10.0
0
23:03:07 --- Epoch: 2   Train loss: 2.3052      Test loss: 2.3051       Train accuracy: 10.09   Test accuracy: 10.0
0
23:04:11 --- Epoch: 3   Train loss: 2.3047      Test loss: 2.3045       Train accuracy: 10.17   Test accuracy: 10.0
0
23:05:16 --- Epoch: 4   Train loss: 2.3050      Test loss: 2.3051       Train accuracy: 10.22   Test accuracy: 10.0
0
23:06:20 --- Epoch: 5   Train loss: 2.3047      Test loss: 2.3061       Train accuracy: 10.17   Test accuracy: 10.0
0
23:07:24 --- Epoch: 6   Train loss: 2.3047      Test loss: 2.3042       Train accuracy: 10.22   Test accuracy: 10.0
0
```

In the above I print the training and testing losses and accuracies for every epoch. And I keep my epochs to 15 and change the Optimizer and learning rates to observe the results.