Sreyashi Saha

Mat No: 1747271

Group 1 Assignment 7

Reference: https://towardsdatascience.com/implementing-yann-lecuns-lenet-5-in-pytorch-5e05a0911320

Importing all the libraries required to implement LeNet architechture on MNIST data set.

In [1]:
```
import numpy as np
from datetime import datetime

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader

from torchvision import datasets, transforms

import matplotlib.pyplot as plt

import torch
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter()

# check device
DEVICE = 'cuda' if torch.cuda.is_available() else '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.

In [2]:
```
# pip install torch torchvision torchaudio
```

LeNet Architechture On MNIST

The LeNet-5 CNN architecture has seven layers. Three convolutional layers, two subsampling layers, and two fully linked layers make up the layer composition.

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 32x32 in size since the input layer is designed to accept images of that size. The images in the MNIST collection are 28x28 sized images, which are padded to bring the MNIST images' dimensions into compliance with the input layer's specifications.

The pixel values of the grayscale images used in the study were normalized from 0 to 255 to values between -0.1 and 1.175. To ensure that the batch of images has a mean of 0 and a standard deviation of 1, normalization is used. The benefit of this is a decrease in training time. We'll be normalizing the pixel values of the photos to take on values between 0 and 1 for the image classification with LeNet-5.

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 7 layers to get a better understanding of the LeNet 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* 1).

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(S2): The second pooling layer. The logic is identical to the previous one, but with 16 filters instead of 6. The output of this layer is of size 5 *5* 16.

5) Layer5(C5): 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 1 *1* 120. Layers S4

and C5 are hence totally linked. This is also the reason why some LeNet-5 implementations actually utilize a fully-connected layer as the fifth layer rather than a convolutional one. This layer is kept as a convolutional one because if the network's input is larger than the one used in [1] (the starting input, which in MNIST case is 32 * 32), this layer won't be properly linked because each kernel's output won't be 11.

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 last dense layer, outputs 10 classes after taking 84 input features.

In [3]:
```
# Here we set up some parameters which will be used later in the code
# parameters
RANDOM_SEED = 12
LEARNING_RATE = 0.001
BATCH_SIZE = 32
N_EPOCHS = 15

IMG_SIZE = 32
N_CLASSES = 10
```

Using the current weights, I get the predictions for each batch of data in the train function. This is known as the Forward Pass. I then figure out the loss function's value. I follow this with a backward pass, where the weights are changed in accordance with the loss. This is known as the "Learning Phase". 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 LeNet5 model, along with the optimizer used i.e. Adam, Sgd or RMSprop and the loss obtained after each epoch.

In [4]:
```
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 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.

In [5]:
```
def test(test, 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
```

This funtion 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.

In [6]:
```
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 this 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.

In [7]:
```
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 paramenters 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.

In [8]:
```
def evaluation(model, criterion, optimizer, train_loader, test_loader, ep

    # 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, criteri
        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)
    train_arr = np.array(train_losses)
    train_tensor = torch.as_tensor(train_arr)
    test_arr = np.array(test_losses)
    test_tensor = torch.as_tensor(test_arr)
#     writer.add_scalar("Loss train", train_losses, epochs)
#     writer.add_scalar("Loss test", test_losses, epochs)

    return model, optimizer, (train_arr, test_arr)
```

Below we transform our input data by resizing the images to 32*32(the input size of LeNet-5) matrix and also convert an image to a Tensor, which automatically scales the images to the [0,1] range. Next to this I download the MNIST dataset and store it in train and test datasets. It is necessary to load enormous datasets into memory all at once while working with them. Now due to the system's low memory capacity, having memory outrage is common. Additionally, because large datasets are loaded all at once, programs generally run slowly. By utilizing DataLoader, PyTorch provides a method for automatically batching and parallelizing the data loading process. Data loading is parallelized using a dataloader because doing so not only speeds up the process but also conserves memory. So lastly, I initialised the DataLoaders by providing the dataset, the batch size, and the desire to shuffle the dataset in each epoch. For testing, this does not make a difference so I set it to False.

In [9]:
```
# define transforms
transforms = transforms.Compose([transforms.Resize((32, 32)),
                                 transforms.ToTensor()])

# download and create datasets
train_dataset = datasets.MNIST(root='mnist_data',
                               train=True,
                               transform=transforms,
                               download=True)

test_dataset = datasets.MNIST(root='mnist_data',
                              train=False,
                              transform=transforms)

# define the data loaders
train_loader = DataLoader(dataset=train_dataset,
                          batch_size=BATCH_SIZE,
                          shuffle=True)

test_loader = DataLoader(dataset=test_dataset,
                         batch_size=BATCH_SIZE,
                         shuffle=False)
```

Before implementing our Model which is Lenet 5 let us first review Activation Functions. Layers of neurons make up neural networks. Together, they provide a system that discovers patterns in a dataset that are concealed. Here, each individual neuron processes information in the shape of Wx + b. In this case, x stands for the input vector, which can either be the initial layer's input data or any later, partially processed data (in the downstream layers). The trainable parts of a neural network are represented by b, the bias, and W, the weights vector.

Making a linear operation is equivalent to performing Wx + b. In other words, there is always a linear relationship between an input value and an output value. This is ideal if we require a model to produce a linear decision boundary, but it is difficult otherwise.

In reality, if we merely carry out the operation outlined previously, we won't be able to learn a decision boundary that is not linear (and there are many such use cases, for example in computer vision).

In this situation, activation functions comes to the rescue. They take the neuron input values and transfer this linear input to a nonlinear output, stacking them just after the neurons. As a result, both the individual neurons and the system as a whole can learn nonlinear patterns.

The neuron performs the action Wx + b as input data flows through it. An activation function, such as ReLU, Sigmoid, and Tanh, channels the neuron's output. What the activation function outputs is either passed to the next layer or returned as model output.

In terms of significance on neural networks, the Tanh and Sigmoid activation functions are the most ancient. Tanh translates all inputs into the (-1.0, 1.0) range, as shown in the plot below, with the slope being highest at x = 0. Instead, Sigmoid scales all inputs to the (0.0, 1.0) range, with the steepest slope occurring at x = 0. ReLU is unique. If x = 0.0, this function converts all inputs to 0.0. The input is mapped to x in all other circumstances.

Finally, we define the LeNet-5 architecture using ReLU as our Activation Function

In [10]:
```
class LeNet5(nn.Module):

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

        self.feature_extractor = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, strid
            nn.ReLU(),
            nn.AvgPool2d(kernel_size=2),
            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stri
            nn.ReLU(),
            nn.AvgPool2d(kernel_size=2),
            nn.Conv2d(in_channels=16, out_channels=120, kernel_size=5, st
            nn.ReLU()
        )

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

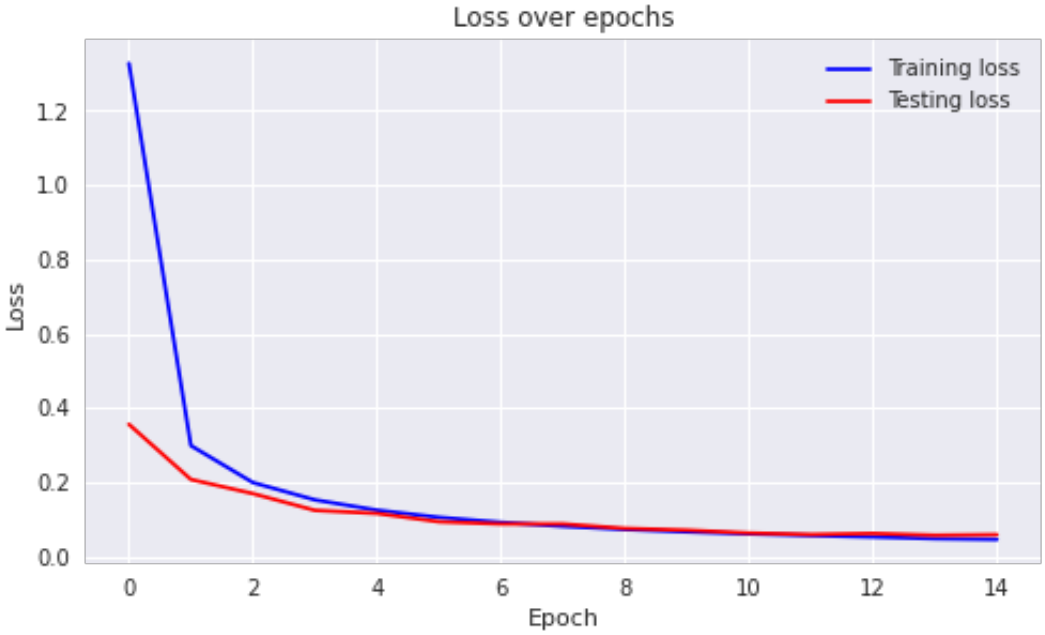
After defining the class, we need to initialise the model (and send it to the correct device), the optimizer (ADAM,Sgd and RMSprop in this case), and the loss function (Cross entropy).

In [11]:
```
torch.manual_seed(RANDOM_SEED)

model = LeNet5(N_CLASSES).to(DEVICE)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
criterion = nn.CrossEntropyLoss()
```

In [12]:
```
model, optimizer, _ = evaluation(model, criterion, optimizer, train_loade
```

```
00:08:32 --- Epoch: 0   Train loss: 1.3281      Test loss: 0.3554
Train accuracy: 88.62   Test accuracy: 89.56
00:08:51 --- Epoch: 1   Train loss: 0.2980      Test loss: 0.2067
Train accuracy: 93.13   Test accuracy: 93.56
00:09:10 --- Epoch: 2   Train loss: 0.1984      Test loss: 0.1686
Train accuracy: 94.79   Test accuracy: 94.78
00:09:29 --- Epoch: 3   Train loss: 0.1517      Test loss: 0.1235
Train accuracy: 96.12   Test accuracy: 96.09
00:09:48 --- Epoch: 4   Train loss: 0.1240      Test loss: 0.1152
Train accuracy: 96.45   Test accuracy: 96.35
00:10:07 --- Epoch: 5   Train loss: 0.1046      Test loss: 0.0927
Train accuracy: 97.27   Test accuracy: 97.21
00:10:26 --- Epoch: 6   Train loss: 0.0911      Test loss: 0.0868
Train accuracy: 97.60   Test accuracy: 97.33
00:10:45 --- Epoch: 7   Train loss: 0.0805      Test loss: 0.0868
Train accuracy: 97.56   Test accuracy: 97.38
00:11:04 --- Epoch: 8   Train loss: 0.0720      Test loss: 0.0748
Train accuracy: 97.81   Test accuracy: 97.55
00:11:23 --- Epoch: 9   Train loss: 0.0654      Test loss: 0.0701
Train accuracy: 98.19   Test accuracy: 97.75
00:11:42 --- Epoch: 10  Train loss: 0.0601      Test loss: 0.0623
Train accuracy: 98.53   Test accuracy: 98.01
00:12:01 --- Epoch: 11  Train loss: 0.0556      Test loss: 0.0584
Train accuracy: 98.55   Test accuracy: 98.07
00:12:20 --- Epoch: 12  Train loss: 0.0518      Test loss: 0.0610
Train accuracy: 98.55   Test accuracy: 98.10
00:12:39 --- Epoch: 13  Train loss: 0.0472      Test loss: 0.0563
Train accuracy: 98.73   Test accuracy: 98.15
00:12:58 --- Epoch: 14  Train loss: 0.0451      Test loss: 0.0578
Train accuracy: 98.67   Test accuracy: 98.28
```

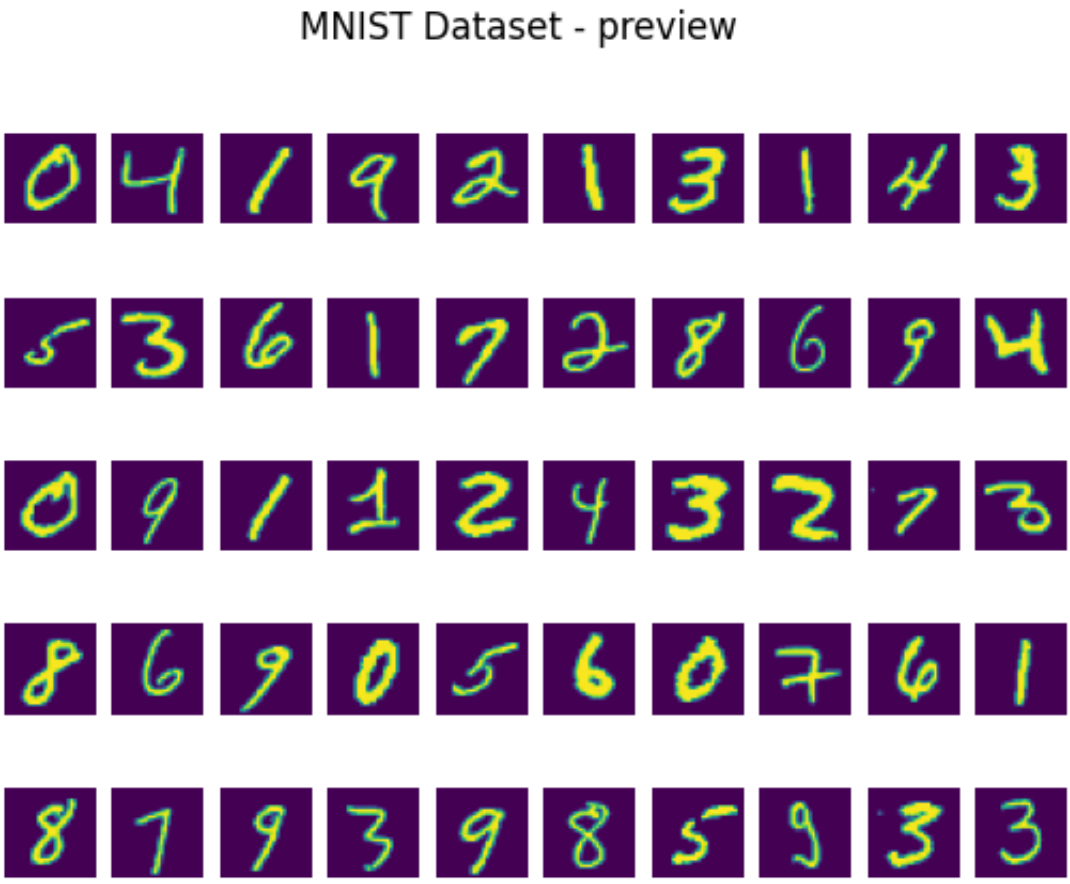
```
/var/folders/mv/39f6vvvn1kb8pjc2zbvyxr5w0000gn/T/ipykernel_7279/308483355
.py:17: UserWarning: Matplotlib is currently using module://matplotlib_in
line.backend_inline, which is a non-GUI backend, so cannot show the figur
e.
  fig.show()
```



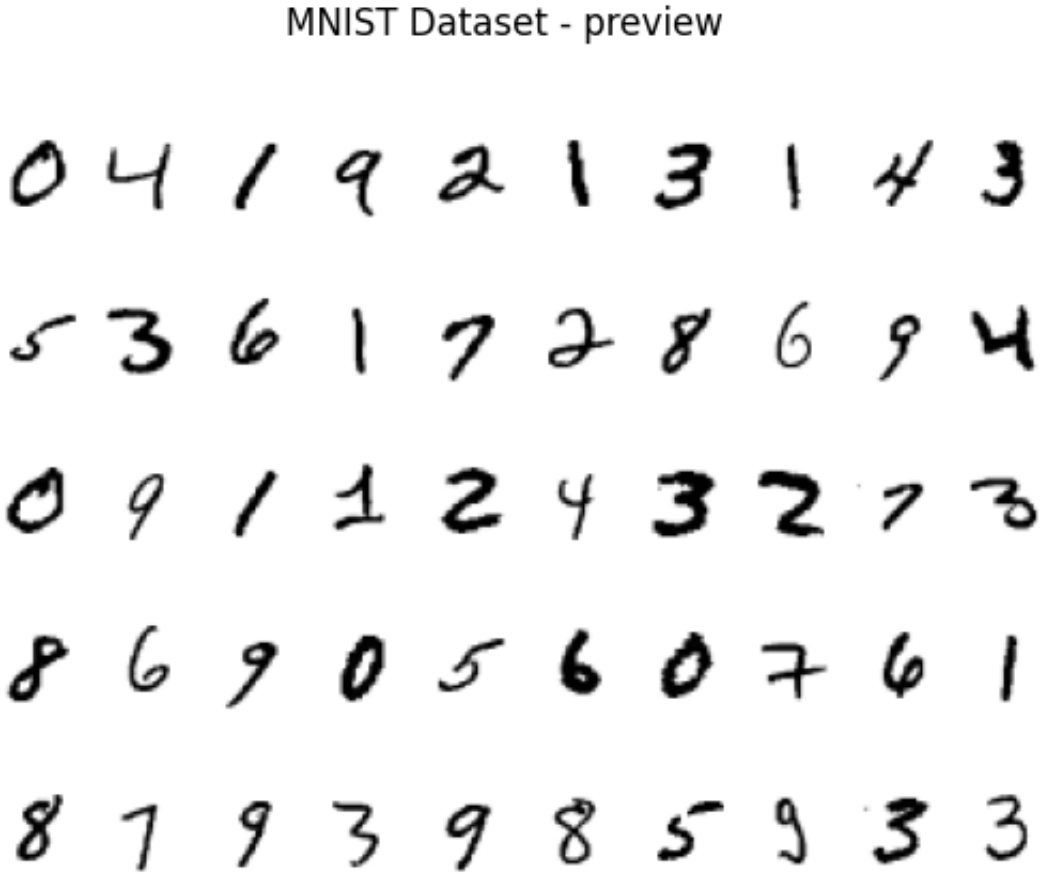
Here I used SGD optimizer and set my learning rate as 0.01. Now in the above plot I can see that the testing loss as well as the training loss decreases after evey epoch. I think the performance may be characterized as being fairly satisfactory overall. The 11th epoch produced the best outcomes (on the testing set).

In [50]:
```
ROW_IMG = 10
N_ROWS = 5
```

In [51]:
```
fig = plt.figure()
for index in range(1, ROW_IMG * N_ROWS + 1):
    plt.subplot(N_ROWS, ROW_IMG, index)
    plt.axis('off')
    plt.imshow(train_dataset.data[index])
fig.suptitle('MNIST Dataset - preview');
```

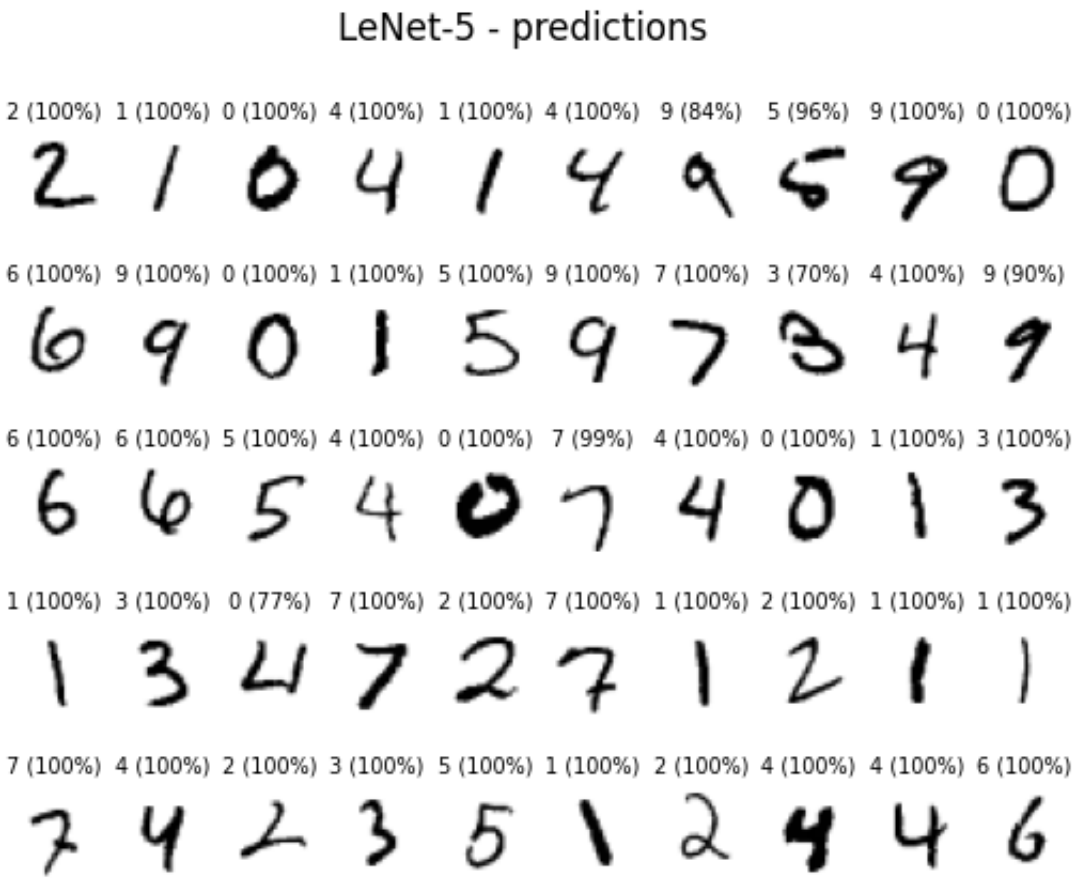


In [52]:
```
fig = plt.figure()
for index in range(1, ROW_IMG * N_ROWS + 1):
    plt.subplot(N_ROWS, ROW_IMG, index)
    plt.axis('off')
    plt.imshow(train_dataset.data[index], cmap='gray_r')
fig.suptitle('MNIST Dataset - preview');
```



The following code presents a series of integers from the testing set together with the predicted label and the probability that the network assigns to that label, allowing us to assess the predictions of our model (in other words, how confident the network is in the prediction).

In [53]:
```
fig = plt.figure()
for index in range(1, ROW_IMG * N_ROWS + 1):
    plt.subplot(N_ROWS, ROW_IMG, index)
    plt.axis('off')
    plt.imshow(test_dataset.data[index], cmap='gray_r')

    with torch.no_grad():
        model.eval()
        _, probs = model(test_dataset[index][0].unsqueeze(0))

    title = f'{torch.argmax(probs)} ({torch.max(probs * 100):.0f}%)'

    plt.title(title, fontsize=7)
fig.suptitle('LeNet-5 - predictions');
```



The network is almost always certain of the label, as shown in the image above, with the number 9 (in the first row, fourth from right), where it is only 84% certain that it is a 9, the number 4 (in the second last row, third from left), where it is only 77% certain that it is a 4, being the exceptions.

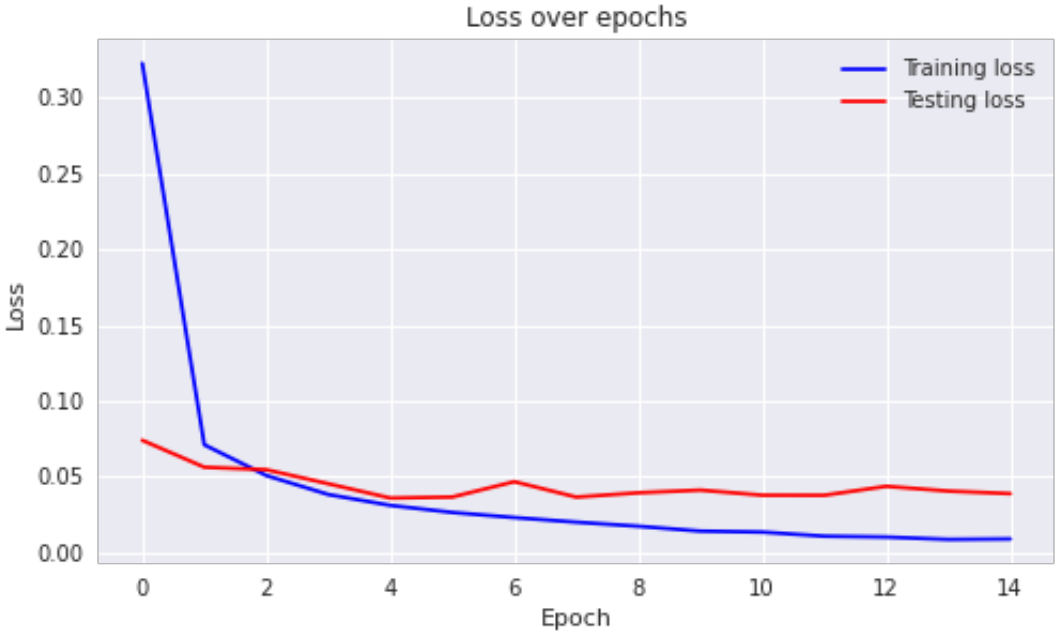
Finally, we define the LeNet-5 architecture using learning rate as 0.1 and SGD as my optimizer.

In [11]:
```
torch.manual_seed(RANDOM_SEED)

model = LeNet5(N_CLASSES).to(DEVICE)
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
criterion = nn.CrossEntropyLoss()
```

In [12]:
```
model, optimizer, _ = evaluation(model, criterion, optimizer, train_loade
```

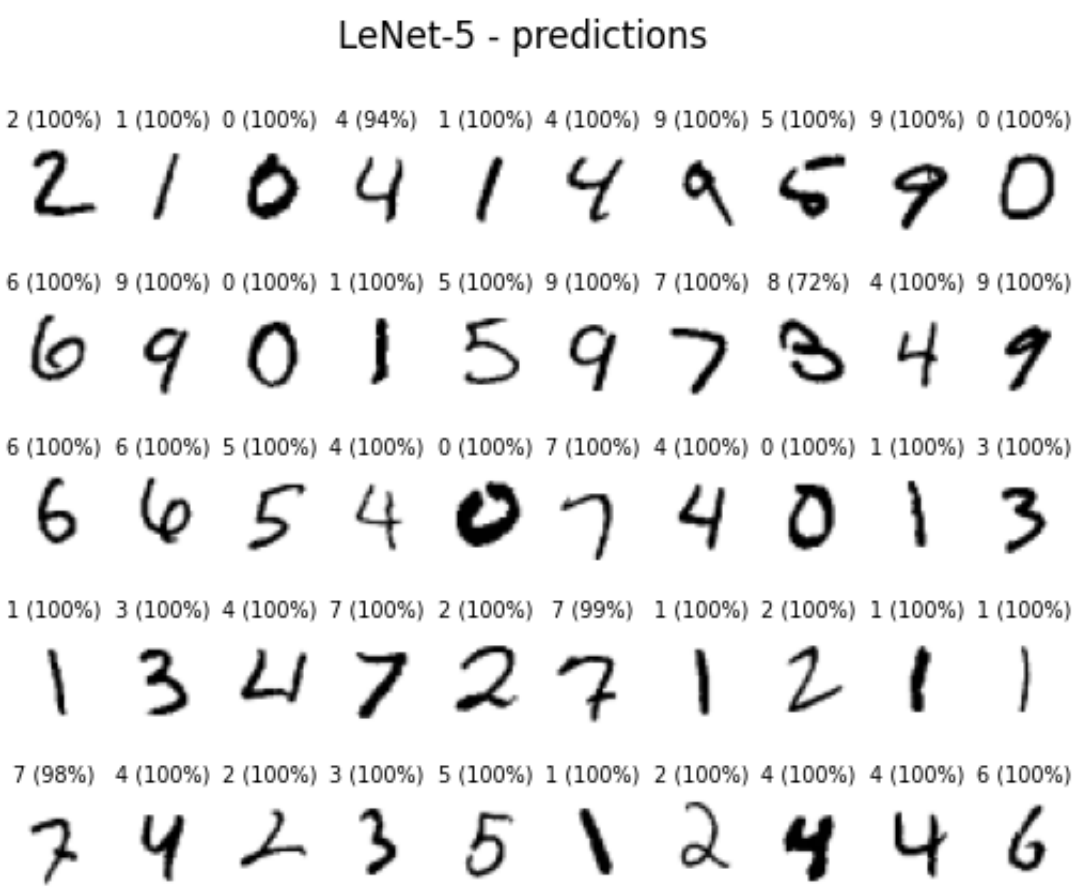
```
12:11:59 --- Epoch: 0   Train loss: 0.3226      Test loss: 0.0741
Train accuracy: 97.88   Test accuracy: 97.94
12:12:19 --- Epoch: 1   Train loss: 0.0711      Test loss: 0.0563
Train accuracy: 98.34   Test accuracy: 98.23
12:12:37 --- Epoch: 2   Train loss: 0.0508      Test loss: 0.0546
Train accuracy: 98.53   Test accuracy: 98.20
12:12:56 --- Epoch: 3   Train loss: 0.0383      Test loss: 0.0454
Train accuracy: 99.03   Test accuracy: 98.48
12:13:15 --- Epoch: 4   Train loss: 0.0311      Test loss: 0.0359
Train accuracy: 99.29   Test accuracy: 98.79
12:13:34 --- Epoch: 5   Train loss: 0.0264      Test loss: 0.0366
Train accuracy: 99.40   Test accuracy: 98.78
12:13:53 --- Epoch: 6   Train loss: 0.0231      Test loss: 0.0468
Train accuracy: 99.22   Test accuracy: 98.74
12:14:12 --- Epoch: 7   Train loss: 0.0200      Test loss: 0.0366
Train accuracy: 99.54   Test accuracy: 98.86
12:14:31 --- Epoch: 8   Train loss: 0.0173      Test loss: 0.0394
Train accuracy: 99.44   Test accuracy: 98.65
12:14:50 --- Epoch: 9   Train loss: 0.0141      Test loss: 0.0412
Train accuracy: 99.58   Test accuracy: 98.88
12:15:12 --- Epoch: 10  Train loss: 0.0135      Test loss: 0.0378
Train accuracy: 99.72   Test accuracy: 98.85
12:15:34 --- Epoch: 11  Train loss: 0.0108      Test loss: 0.0378
Train accuracy: 99.76   Test accuracy: 98.98
12:15:53 --- Epoch: 12  Train loss: 0.0102      Test loss: 0.0437
Train accuracy: 99.73   Test accuracy: 98.78
12:16:12 --- Epoch: 13  Train loss: 0.0087      Test loss: 0.0405
Train accuracy: 99.76   Test accuracy: 98.97
12:16:31 --- Epoch: 14  Train loss: 0.0090      Test loss: 0.0389
Train accuracy: 99.77   Test accuracy: 99.04
```

```
/var/folders/mv/39f6vvvn1kb8pjc2zbvyxr5w0000gn/T/ipykernel_11139/308483355.py:17: UserWarning: Matplotlib is currently using module://matplotlib_inline.backend_inline, which is a non-GUI backend, so cannot show the figure.
  fig.show()
```



Here I have used 0.1 as my learning rate and SGD as my optimizer and now in the above plot I can see that the testing loss shows tiny bumps while the training loss decreases after evey epoch. I think the performance may be characterized as being fairly satisfactory overall. Here again, the 10th epoch produces the best outcomes (on the testing set).

In [14]:
```
ROW_IMG = 10
N_ROWS = 5
fig = plt.figure()
for index in range(1, ROW_IMG * N_ROWS + 1):
    plt.subplot(N_ROWS, ROW_IMG, index)
    plt.axis('off')
    plt.imshow(test_dataset.data[index], cmap='gray_r')

    with torch.no_grad():
        model.eval()
        _, probs = model(test_dataset[index][0].unsqueeze(0))

    title = f'{torch.argmax(probs)} ({torch.max(probs * 100):.0f}%)'

    plt.title(title, fontsize=7)
fig.suptitle('LeNet-5 - predictions');
```



The network is almost always certain of the label, as shown in the image above, with the number 3 (in the second row, third from right), where it is only 72% certain that it is a 3, being the only exception.

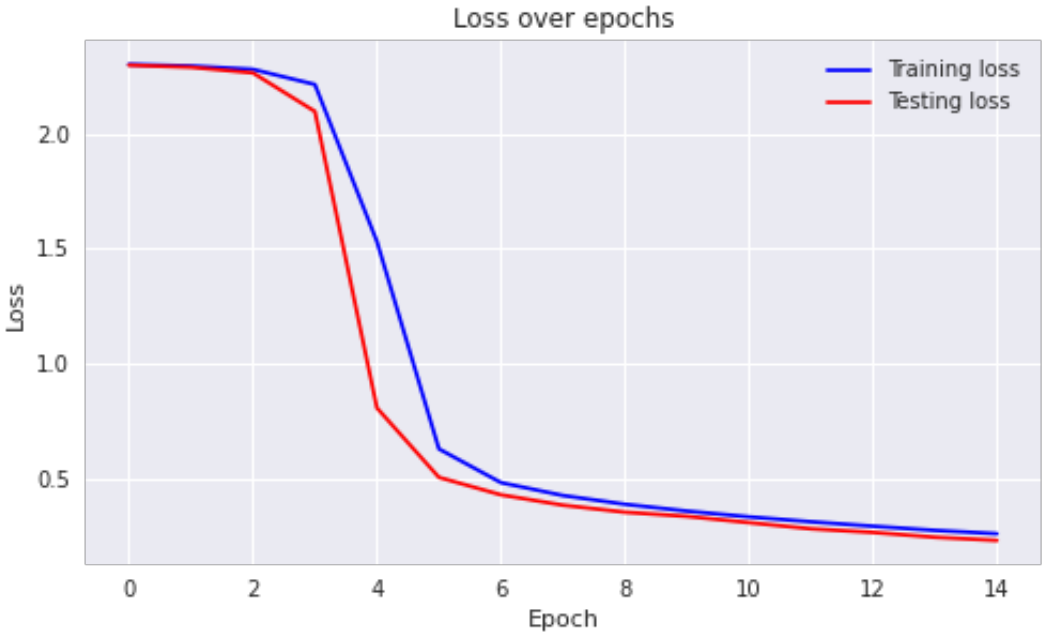
Finally, we define the LeNet-5 architecture using learning rate as 0.01 and RMSprop as my optimizer.

In [11]:
```
torch.manual_seed(RANDOM_SEED)

model = LeNet5(N_CLASSES).to(DEVICE)
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()
```

In [12]:
```
model, optimizer, _ = evaluation(model, criterion, optimizer, train_loade
```

```
12:24:13 --- Epoch: 0   Train loss: 2.3020      Test loss: 2.2988
Train accuracy: 9.07    Test accuracy: 8.96
12:24:32 --- Epoch: 1   Train loss: 2.2951      Test loss: 2.2897
Train accuracy: 35.26   Test accuracy: 36.32
12:24:51 --- Epoch: 2   Train loss: 2.2806      Test loss: 2.2651
Train accuracy: 41.76   Test accuracy: 43.06
12:25:11 --- Epoch: 3   Train loss: 2.2146      Test loss: 2.0973
Train accuracy: 47.33   Test accuracy: 48.55
12:25:30 --- Epoch: 4   Train loss: 1.5314      Test loss: 0.8069
Train accuracy: 77.31   Test accuracy: 78.23
12:25:49 --- Epoch: 5   Train loss: 0.6284      Test loss: 0.5046
Train accuracy: 84.07   Test accuracy: 84.74
12:26:10 --- Epoch: 6   Train loss: 0.4817      Test loss: 0.4285
Train accuracy: 86.77   Test accuracy: 87.25
12:26:29 --- Epoch: 7   Train loss: 0.4256      Test loss: 0.3834
Train accuracy: 88.27   Test accuracy: 88.76
12:26:49 --- Epoch: 8   Train loss: 0.3875      Test loss: 0.3529
Train accuracy: 88.99   Test accuracy: 89.51
12:27:08 --- Epoch: 9   Train loss: 0.3578      Test loss: 0.3347
Train accuracy: 89.59   Test accuracy: 90.20
12:27:27 --- Epoch: 10  Train loss: 0.3327      Test loss: 0.3074
Train accuracy: 90.13   Test accuracy: 90.65
12:27:46 --- Epoch: 11  Train loss: 0.3113      Test loss: 0.2806
Train accuracy: 91.02   Test accuracy: 91.64
12:28:05 --- Epoch: 12  Train loss: 0.2920      Test loss: 0.2648
Train accuracy: 91.62   Test accuracy: 91.93
12:28:25 --- Epoch: 13  Train loss: 0.2738      Test loss: 0.2438
Train accuracy: 92.11   Test accuracy: 92.69
12:28:44 --- Epoch: 14  Train loss: 0.2593      Test loss: 0.2300
Train accuracy: 92.57   Test accuracy: 93.18
```

```
/var/folders/mv/39f6vvvn1kb8pjc2zbvyxr5w0000gn/T/ipykernel_11285/30848335
5.py:17: UserWarning: Matplotlib is currently using module://matplotlib_i
nline.backend_inline, which is a non-GUI backend, so cannot show the figu
re.
  fig.show()
```



Here I have used 0.001 as my learning rate and SGD as my optimizer and now in the above plot I can see that both the testing and training loss shows a huge drop in loss after the 3rd epoch and thereafter decreases gradually. However, the accuracy just reached 93% in 15 epochs, therefore, in the next step I'll increase the number of epochs.

In [13]:
```
ROW_IMG = 10
N_ROWS = 5
fig = plt.figure()
for index in range(1, ROW_IMG * N_ROWS + 1):
    plt.subplot(N_ROWS, ROW_IMG, index)
    plt.axis('off')
    plt.imshow(test_dataset.data[index], cmap='gray_r')

    with torch.no_grad():
        model.eval()
        _, probs = model(test_dataset[index][0].unsqueeze(0))

    title = f'{torch.argmax(probs)} ({torch.max(probs * 100):.0f}%)'

    plt.title(title, fontsize=6)
fig.suptitle('LeNet-5 - predictions');
```

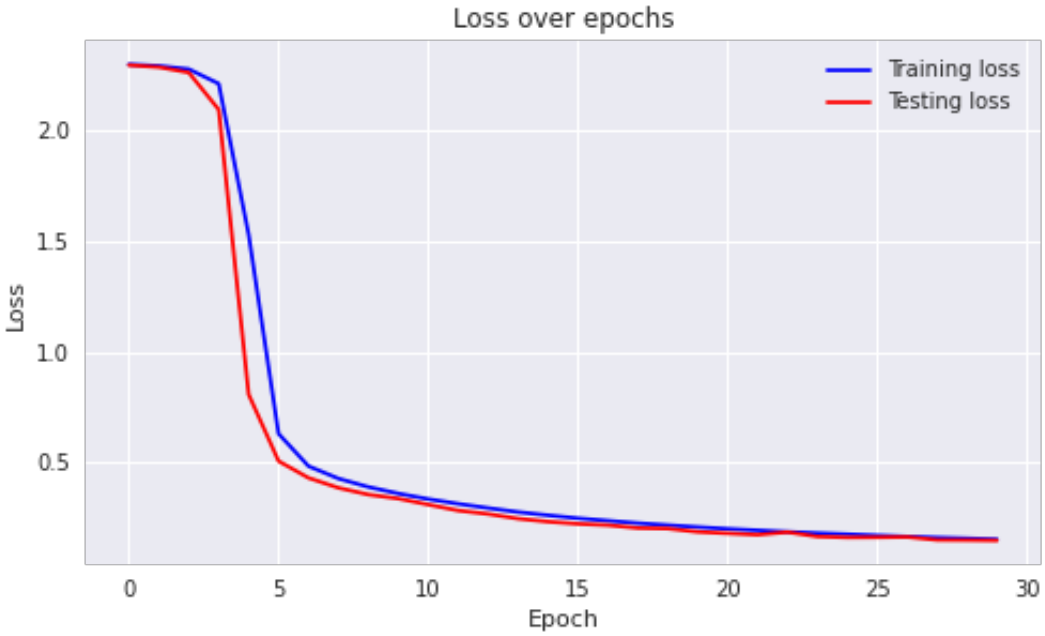
The network is almost always certain of the label, as shown in the image above, with the number 5 (in the first row, third from right), where it is only 52% certain that it is a 5, the number 4 (in the third row, fourth from left), where it is only 77% certain that it is a 4, being the exceptions.

In [11]:
```
torch.manual_seed(RANDOM_SEED)

model = LeNet5(N_CLASSES).to(DEVICE)
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()
model, optimizer, _ = evaluation(model, criterion, optimizer, train_loade
```

```
12:33:04 --- Epoch: 0   Train loss: 2.3020      Test loss: 2.2988
Train accuracy: 9.07    Test accuracy: 8.96
12:33:23 --- Epoch: 1   Train loss: 2.2951      Test loss: 2.2897
Train accuracy: 35.26   Test accuracy: 36.32
12:33:43 --- Epoch: 2   Train loss: 2.2806      Test loss: 2.2651
Train accuracy: 41.76   Test accuracy: 43.06
12:34:04 --- Epoch: 3   Train loss: 2.2146      Test loss: 2.0973
Train accuracy: 47.33   Test accuracy: 48.55
12:34:23 --- Epoch: 4   Train loss: 1.5314      Test loss: 0.8069
Train accuracy: 77.31   Test accuracy: 78.23
12:34:43 --- Epoch: 5   Train loss: 0.6284      Test loss: 0.5046
Train accuracy: 84.07   Test accuracy: 84.74
12:35:03 --- Epoch: 6   Train loss: 0.4817      Test loss: 0.4285
Train accuracy: 86.77   Test accuracy: 87.25
12:35:23 --- Epoch: 7   Train loss: 0.4256      Test loss: 0.3834
Train accuracy: 88.27   Test accuracy: 88.76
```

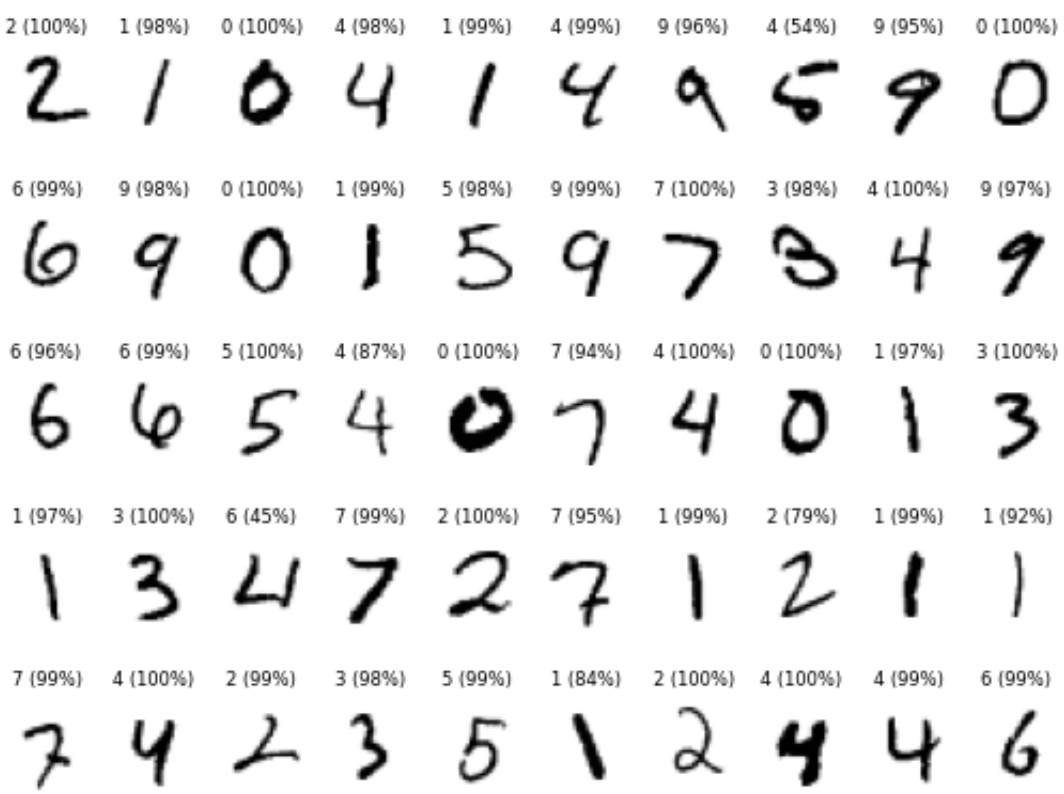
```
12:35:43 --- Epoch: 8   Train loss: 0.3875      Test loss: 0.3529
Train accuracy: 88.99   Test accuracy: 89.51
12:36:02 --- Epoch: 9   Train loss: 0.3578      Test loss: 0.3347
Train accuracy: 89.59   Test accuracy: 90.20
12:36:21 --- Epoch: 10  Train loss: 0.3327      Test loss: 0.3074
Train accuracy: 90.13   Test accuracy: 90.65
12:36:41 --- Epoch: 11  Train loss: 0.3113      Test loss: 0.2806
Train accuracy: 91.02   Test accuracy: 91.64
12:37:00 --- Epoch: 12  Train loss: 0.2920      Test loss: 0.2648
Train accuracy: 91.62   Test accuracy: 91.93
12:37:19 --- Epoch: 13  Train loss: 0.2738      Test loss: 0.2438
Train accuracy: 92.11   Test accuracy: 92.69
12:37:39 --- Epoch: 14  Train loss: 0.2593      Test loss: 0.2300
Train accuracy: 92.57   Test accuracy: 93.18
12:37:58 --- Epoch: 15  Train loss: 0.2456      Test loss: 0.2203
Train accuracy: 92.86   Test accuracy: 93.42
12:38:18 --- Epoch: 16  Train loss: 0.2345      Test loss: 0.2144
Train accuracy: 93.05   Test accuracy: 93.64
12:38:38 --- Epoch: 17  Train loss: 0.2234      Test loss: 0.2015
Train accuracy: 93.38   Test accuracy: 94.02
12:38:57 --- Epoch: 18  Train loss: 0.2144      Test loss: 0.1989
Train accuracy: 93.70   Test accuracy: 94.24
12:39:17 --- Epoch: 19  Train loss: 0.2058      Test loss: 0.1840
Train accuracy: 94.03   Test accuracy: 94.52
12:39:36 --- Epoch: 20  Train loss: 0.1977      Test loss: 0.1772
Train accuracy: 94.22   Test accuracy: 94.67
12:39:56 --- Epoch: 21  Train loss: 0.1910      Test loss: 0.1710
Train accuracy: 94.47   Test accuracy: 94.93
12:40:16 --- Epoch: 22  Train loss: 0.1842      Test loss: 0.1827
Train accuracy: 94.21   Test accuracy: 94.47
12:40:35 --- Epoch: 23  Train loss: 0.1783      Test loss: 0.1628
Train accuracy: 94.83   Test accuracy: 95.13
12:40:55 --- Epoch: 24  Train loss: 0.1734      Test loss: 0.1588
Train accuracy: 94.88   Test accuracy: 95.21
12:41:14 --- Epoch: 25  Train loss: 0.1677      Test loss: 0.1593
Train accuracy: 94.76   Test accuracy: 95.08
12:41:33 --- Epoch: 26  Train loss: 0.1627      Test loss: 0.1609
Train accuracy: 94.76   Test accuracy: 94.99
12:41:53 --- Epoch: 27  Train loss: 0.1580      Test loss: 0.1475
Train accuracy: 95.29   Test accuracy: 95.48
12:42:12 --- Epoch: 28  Train loss: 0.1545      Test loss: 0.1467
Train accuracy: 95.44   Test accuracy: 95.54
12:42:31 --- Epoch: 29  Train loss: 0.1502      Test loss: 0.1452
Train accuracy: 95.60   Test accuracy: 95.67
```

```
/var/folders/mv/39f6vvvn1kb8pjc2zbvyxr5w0000gn/T/ipykernel_11326/308483355.py:17: UserWarning: Matplotlib is currently using module://matplotlib_inline.backend_inline, which is a non-GUI backend, so cannot show the figure.
  fig.show()
```



After increasing the number of epochs I see that the accuracy increased to almost 96% and also we see a gradual decrease in the loss from both the datasets.

In [12]:
```
ROW_IMG = 10
N_ROWS = 5
fig = plt.figure()
for index in range(1, ROW_IMG * N_ROWS + 1):
    plt.subplot(N_ROWS, ROW_IMG, index)
    plt.axis('off')
    plt.imshow(test_dataset.data[index], cmap='gray_r')

    with torch.no_grad():
        model.eval()
        _, probs = model(test_dataset[index][0].unsqueeze(0))

    title = f'{torch.argmax(probs)} ({torch.max(probs * 100):.0f}%)'

    plt.title(title, fontsize=6)
fig.suptitle('LeNet-5 - predictions');
```



The network is almost always certain of the label, as shown in the image above, with the number 5 (in the first row, third from right), where it is only 54% certain that it is a 5, the number 2 (in the second last row, third from right), where it is only 79% certain that it is a 2 and the number 4 in the same row(third from left) where it is 45% sure that it is a 4, being the exceptions.

Cifar-10

The dataset CIFAR-10 contains a collection of photos from 10 different classes. This dataset is frequently used in scientific research to test various machine learning models, particularly for computer vision issues. In order to determine the prediction accuracy that can be achieved, we will attempt to construct a neural network model using Pytorch and test it on the CIFAR-10 dataset.

In [1]:
```
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 [2]:
```
import torch
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter()
```

In [3]:
```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
device
```

Out[3]:
```
'cpu'
```

In [4]:
```
# Here we set up some parameters which will be used later in the code
# parameters
RANDOM_SEED = 12
LEARNING_RATE = 0.001
BATCH_SIZE = 32
N_EPOCHS = 15

IMG_SIZE = 32
N_CLASSES = 10
```

In [5]:
```
# convert data to a normalized torch.FloatTensor
transform = transforms.Compose([transforms.ToTensor(),transforms.Normaliz

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

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

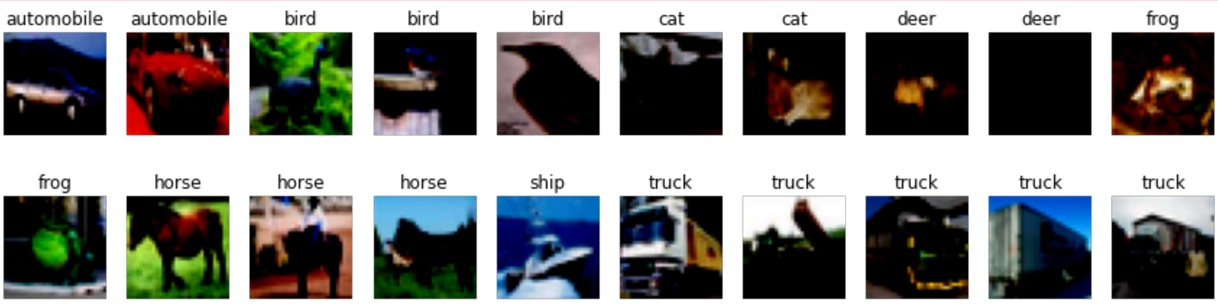
In [6]:
```
# specify the image classes
labels = 'airplane automobile bird cat deer dog frog horse ship truck'.sp
```

Visualizing the Dataset for Feature Extraction

1. After converting the image data to normalised Tensor data I view the images.
2. Matplotlib is then used to display the images.
3. Showing the 20 pictures alongside their associated labels.

In [7]:
```
first_20_samples = sorted([train_data[i] for i in range(20)], key=lambda

figure = plt.figure(figsize=(15,10))
for i in range(1,21):
    img = first_20_samples[i-1][0].permute(1,2,0)
    label = labels[first_20_samples[i-1][1]]
    figure.add_subplot(5,10,i)
    plt.title(label)
    plt.axis('off')
    plt.imshow(img)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
```



In [8]:
```
train_dataloader = torch.utils.data.DataLoader(train_data, batch_size=64,
test_dataloader = torch.utils.data.DataLoader(test_data, batch_size=64, s
```

The logic behind implementing CIFAR10 dataset is the same as MNIST Dataset, except here in the LeNet model we give 3 input channels since the images in Cifar10 dataset are RGB images. The rest of the procedure is same as the MNIST model implemented before.

In [9]:
```
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, strid
            nn.ReLU(),
            nn.AvgPool2d(kernel_size=2),
            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stri
            nn.ReLU(),
            nn.AvgPool2d(kernel_size=2),
            nn.Conv2d(in_channels=16, out_channels=120, kernel_size=5, st
            nn.ReLU()
        )

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

In [10]:
```
model = LeNet5(10).to(device)
print(model)
```

```
LeNet5(
  (feature_extractor): Sequential(
    (0): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU()
    (2): AvgPool2d(kernel_size=2, stride=2, padding=0)
    (3): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
    (4): ReLU()
    (5): AvgPool2d(kernel_size=2, stride=2, padding=0)
    (6): Conv2d(16, 120, kernel_size=(5, 5), stride=(1, 1))
    (7): ReLU()
  )
  (classifier): Sequential(
    (0): Linear(in_features=120, out_features=84, bias=True)
    (1): ReLU()
    (2): Linear(in_features=84, out_features=10, bias=True)
  )
)
```

Using the current weights, I get the predictions for each batch of data in the train function. This is known as the Forward Pass. I then figure out the loss function's value. I follow this with a backward pass, where the weights are changed in accordance with the loss. This is known as the "Learning Phase". 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 LeNet5 model, along with the optimizer used i.e. SGD in my case and the loss obtained after each epoch.

In [11]:
```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 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.

In [12]:
```
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
```

This funtion 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.

In [13]:
```
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 this 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.

In [14]:
```
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 paramenters 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.

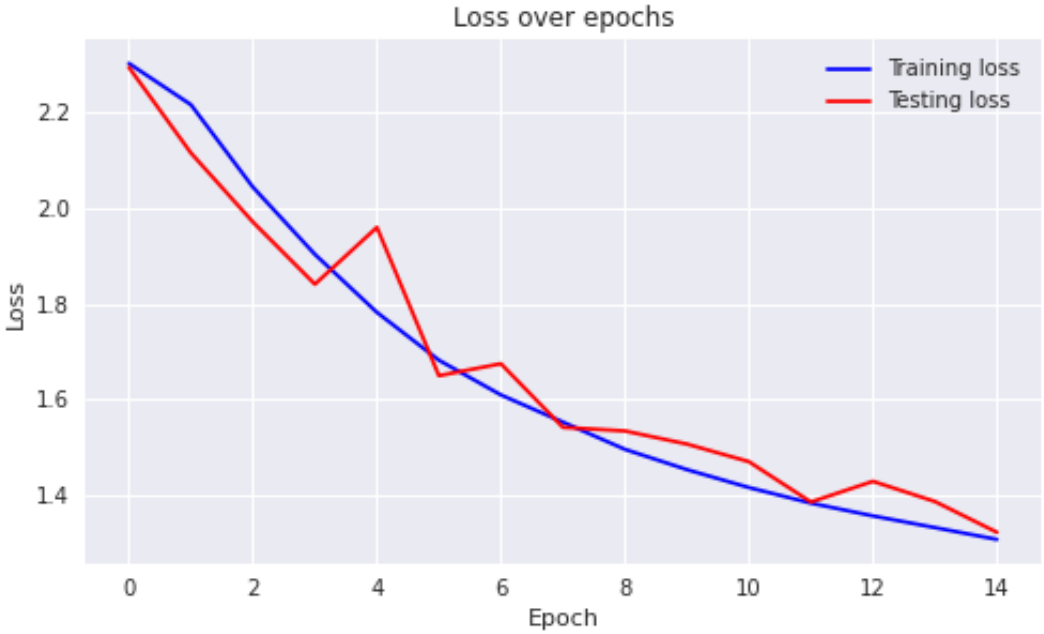
In [15]:
```
def evaluation(model, criterion, optimizer, train_loader, test_loader, ep

    # 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, criteri
        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)
#     train_arr = np.array(train_losses)
#     train_tensor = torch.as_tensor(train_arr)
#     test_arr = np.array(test_losses)
#     test_tensor = torch.as_tensor(test_arr)
#     writer.add_scalar("Loss train", train_losses, epochs)
#     writer.add_scalar("Loss test", test_losses, epochs)

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

In [16]:
```
model = LeNet5(10).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
criterion = nn.CrossEntropyLoss()
```

In [17]:
```
model, optimizer, _ = evaluation(model, criterion, optimizer, train_datal
```

```
13:05:56 --- Epoch: 0   Train loss: 2.3004      Test loss: 2.2923
Train accuracy: 14.28   Test accuracy: 14.32
13:06:58 --- Epoch: 1   Train loss: 2.2145      Test loss: 2.1139
Train accuracy: 22.95   Test accuracy: 23.34
13:07:59 --- Epoch: 2   Train loss: 2.0428      Test loss: 1.9702
Train accuracy: 29.20   Test accuracy: 29.41
13:09:02 --- Epoch: 3   Train loss: 1.9027      Test loss: 1.8406
Train accuracy: 32.52   Test accuracy: 32.95
13:10:04 --- Epoch: 4   Train loss: 1.7820      Test loss: 1.9591
Train accuracy: 30.22   Test accuracy: 30.37
13:11:06 --- Epoch: 5   Train loss: 1.6823      Test loss: 1.6500
Train accuracy: 39.37   Test accuracy: 39.91
13:12:07 --- Epoch: 6   Train loss: 1.6100      Test loss: 1.6750
Train accuracy: 40.05   Test accuracy: 39.45
13:13:09 --- Epoch: 7   Train loss: 1.5534      Test loss: 1.5425
Train accuracy: 43.92   Test accuracy: 43.99
13:14:11 --- Epoch: 8   Train loss: 1.4967      Test loss: 1.5349
Train accuracy: 43.63   Test accuracy: 44.04
13:15:13 --- Epoch: 9   Train loss: 1.4545      Test loss: 1.5076
Train accuracy: 46.47   Test accuracy: 46.32
13:16:14 --- Epoch: 10  Train loss: 1.4171      Test loss: 1.4710
Train accuracy: 46.40   Test accuracy: 45.72
13:17:17 --- Epoch: 11  Train loss: 1.3847      Test loss: 1.3870
Train accuracy: 50.54   Test accuracy: 50.09
13:18:20 --- Epoch: 12  Train loss: 1.3580      Test loss: 1.4300
Train accuracy: 47.59   Test accuracy: 46.90
13:19:22 --- Epoch: 13  Train loss: 1.3337      Test loss: 1.3880
Train accuracy: 50.46   Test accuracy: 49.56
13:20:25 --- Epoch: 14  Train loss: 1.3091      Test loss: 1.3239
Train accuracy: 53.75   Test accuracy: 52.57
```

```
/var/folders/mv/39f6vvvn1kb8pjc2zbvyxr5w0000gn/T/ipykernel_11470/308483355.py:17: UserWarning: Matplotlib is currently using module://matplotlib_inline.backend_inline, which is a non-GUI backend, so cannot show the figure.
  fig.show()
```



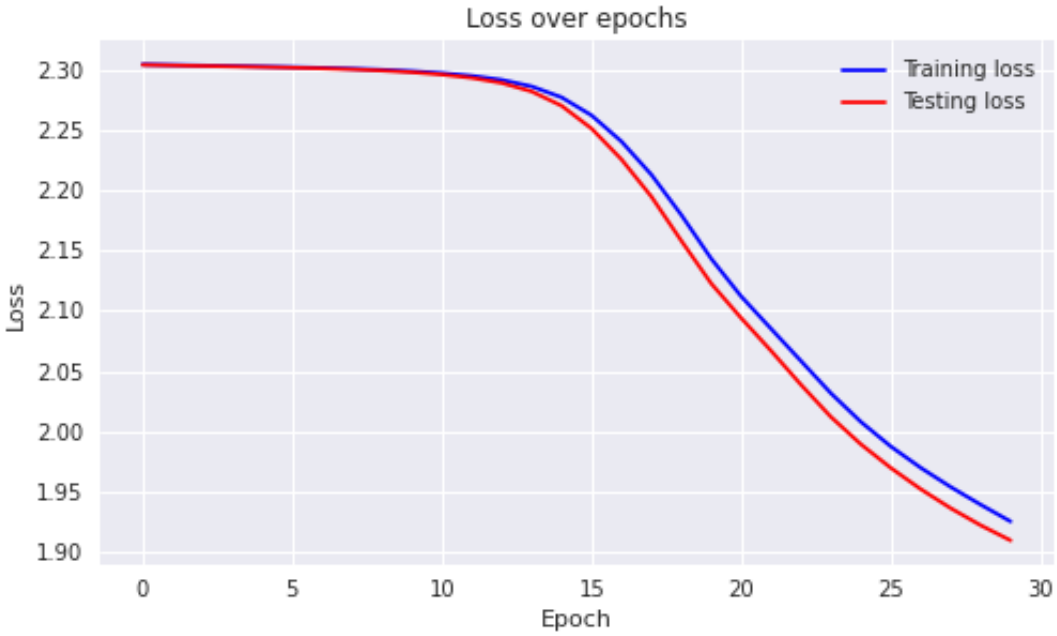
Here I have set my learning rate to be 0.01 and keeping my optimizer as SGD I observe that there are a lot of bumps in the loss obtained from the test data, unlike the train dataset. There's a lot of fluctuations in teh accuracy from both the datasets which is later viewed in TensorBoard.

In [16]:
```
model = LeNet5(10).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()
model, optimizer, _ = evaluation(model, criterion, optimizer, train_datal
```

```
11:07:52 --- Epoch: 0   Train loss: 2.3039      Test loss: 2.3036
Train accuracy: 10.00   Test accuracy: 10.00
11:08:53 --- Epoch: 1   Train loss: 2.3034      Test loss: 2.3031
Train accuracy: 10.01   Test accuracy: 9.99
11:09:54 --- Epoch: 2   Train loss: 2.3030      Test loss: 2.3027
Train accuracy: 10.01   Test accuracy: 9.96
11:10:55 --- Epoch: 3   Train loss: 2.3026      Test loss: 2.3022
Train accuracy: 10.02   Test accuracy: 9.95
11:11:56 --- Epoch: 4   Train loss: 2.3021      Test loss: 2.3018
Train accuracy: 10.04   Test accuracy: 9.91
11:12:57 --- Epoch: 5   Train loss: 2.3016      Test loss: 2.3013
Train accuracy: 10.11   Test accuracy: 9.95
11:13:58 --- Epoch: 6   Train loss: 2.3011      Test loss: 2.3006
Train accuracy: 10.19   Test accuracy: 10.00
11:14:59 --- Epoch: 7   Train loss: 2.3004      Test loss: 2.2999
Train accuracy: 10.17   Test accuracy: 10.19
11:16:00 --- Epoch: 8   Train loss: 2.2995      Test loss: 2.2988
Train accuracy: 10.16   Test accuracy: 10.21
11:17:01 --- Epoch: 9   Train loss: 2.2983      Test loss: 2.2974
Train accuracy: 10.23   Test accuracy: 10.25
11:18:03 --- Epoch: 10  Train loss: 2.2967      Test loss: 2.2955
Train accuracy: 10.74   Test accuracy: 10.85
11:19:04 --- Epoch: 11  Train loss: 2.2945      Test loss: 2.2928
Train accuracy: 12.30   Test accuracy: 12.08
11:20:05 --- Epoch: 12  Train loss: 2.2911      Test loss: 2.2886
Train accuracy: 14.32   Test accuracy: 14.36
11:21:06 --- Epoch: 13  Train loss: 2.2857      Test loss: 2.2815
Train accuracy: 15.73   Test accuracy: 16.08
11:22:07 --- Epoch: 14  Train loss: 2.2766      Test loss: 2.2697
Train accuracy: 16.97   Test accuracy: 17.34
11:23:09 --- Epoch: 15  Train loss: 2.2618      Test loss: 2.2508
Train accuracy: 18.22   Test accuracy: 18.54
11:24:10 --- Epoch: 16  Train loss: 2.2401      Test loss: 2.2253
Train accuracy: 19.07   Test accuracy: 19.57
11:25:11 --- Epoch: 17  Train loss: 2.2126      Test loss: 2.1944
Train accuracy: 20.43   Test accuracy: 20.77
11:26:13 --- Epoch: 18  Train loss: 2.1793      Test loss: 2.1578
Train accuracy: 21.37   Test accuracy: 21.61
11:27:14 --- Epoch: 19  Train loss: 2.1430      Test loss: 2.1224
Train accuracy: 22.01   Test accuracy: 22.20
11:28:15 --- Epoch: 20  Train loss: 2.1118      Test loss: 2.0938
Train accuracy: 22.98   Test accuracy: 23.62
11:29:16 --- Epoch: 21  Train loss: 2.0849      Test loss: 2.0667
Train accuracy: 24.50   Test accuracy: 24.97
11:31:31 --- Epoch: 22  Train loss: 2.0582      Test loss: 2.0386
```

```
Train accuracy: 25.97   Test accuracy: 26.62
11:32:32 --- Epoch: 23  Train loss: 2.0315      Test loss: 2.0119
Train accuracy: 26.54   Test accuracy: 27.73
11:33:33 --- Epoch: 24  Train loss: 2.0078      Test loss: 1.9893
Train accuracy: 27.31   Test accuracy: 28.46
11:34:34 --- Epoch: 25  Train loss: 1.9874      Test loss: 1.9696
Train accuracy: 28.19   Test accuracy: 29.26
11:35:36 --- Epoch: 26  Train loss: 1.9698      Test loss: 1.9521
Train accuracy: 29.08   Test accuracy: 29.93
11:36:37 --- Epoch: 27  Train loss: 1.9539      Test loss: 1.9363
Train accuracy: 29.65   Test accuracy: 30.67
11:37:38 --- Epoch: 28  Train loss: 1.9393      Test loss: 1.9222
Train accuracy: 30.31   Test accuracy: 31.35
11:38:39 --- Epoch: 29  Train loss: 1.9253      Test loss: 1.9096
Train accuracy: 31.15   Test accuracy: 32.32
```

```
/var/folders/mv/39f6vvvn1kb8pjc2zbvyxr5w0000gn/T/ipykernel_10254/308483355.py:17: UserWarning: Matplotlib is currently using module://matplotlib_inline.backend_inline, which is a non-GUI backend, so cannot show the figure.
  fig.show()
```

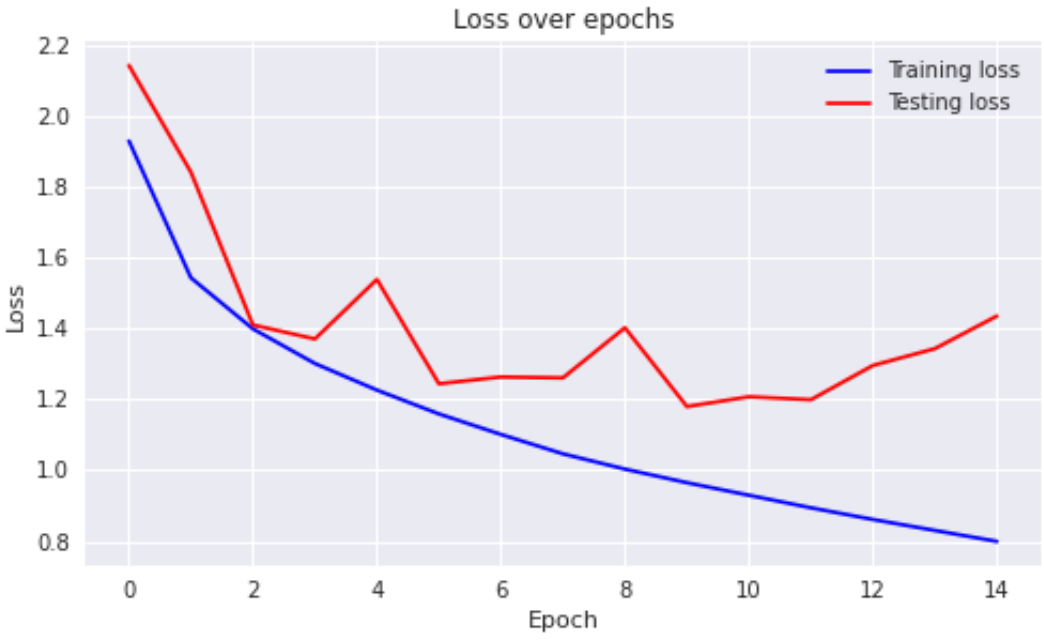


In the above case I have set my learning rate to be 0.001 and number of epochs as 30 and still we don't get a very good output as the results reach only 32% accuracy. Hence we can conclude that with such a small learning rate we need to increase the number of epochs so as to get a reasonable test accuracy. We can also see from the graph above that after the 13th epoch or so there is a gradual decrease in the train and test loss.

In [16]:
```
model = LeNet5(10).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
criterion = nn.CrossEntropyLoss()
model, optimizer, _ = evaluation(model, criterion, optimizer, train_datal
```

```
11:52:51 --- Epoch: 0   Train loss: 1.9291      Test loss: 2.1421
Train accuracy: 27.11   Test accuracy: 27.21
11:53:52 --- Epoch: 1   Train loss: 1.5430      Test loss: 1.8410
Train accuracy: 35.79   Test accuracy: 36.02
11:54:53 --- Epoch: 2   Train loss: 1.3990      Test loss: 1.4100
Train accuracy: 50.50   Test accuracy: 49.50
11:55:54 --- Epoch: 3   Train loss: 1.3012      Test loss: 1.3704
Train accuracy: 53.18   Test accuracy: 51.05
11:56:55 --- Epoch: 4   Train loss: 1.2262      Test loss: 1.5384
Train accuracy: 49.21   Test accuracy: 47.12
11:57:56 --- Epoch: 5   Train loss: 1.1590      Test loss: 1.2440
Train accuracy: 59.81   Test accuracy: 55.94
11:58:57 --- Epoch: 6   Train loss: 1.1011      Test loss: 1.2633
Train accuracy: 59.68   Test accuracy: 55.69
11:59:58 --- Epoch: 7   Train loss: 1.0464      Test loss: 1.2607
Train accuracy: 61.38   Test accuracy: 55.99
12:00:59 --- Epoch: 8   Train loss: 1.0033      Test loss: 1.4021
Train accuracy: 56.03   Test accuracy: 51.66
12:02:00 --- Epoch: 9   Train loss: 0.9650      Test loss: 1.1798
Train accuracy: 65.68   Test accuracy: 59.13
12:03:01 --- Epoch: 10  Train loss: 0.9296      Test loss: 1.2073
Train accuracy: 67.13   Test accuracy: 59.45
12:04:02 --- Epoch: 11  Train loss: 0.8942      Test loss: 1.1991
Train accuracy: 67.73   Test accuracy: 59.19
12:05:04 --- Epoch: 12  Train loss: 0.8612      Test loss: 1.2956
Train accuracy: 64.92   Test accuracy: 56.60
12:06:05 --- Epoch: 13  Train loss: 0.8303      Test loss: 1.3432
Train accuracy: 63.27   Test accuracy: 55.12
12:07:06 --- Epoch: 14  Train loss: 0.7993      Test loss: 1.4352
Train accuracy: 64.06   Test accuracy: 56.04
```

```
/var/folders/mv/39f6vvvn1kb8pjc2zbvyxr5w0000gn/T/ipykernel_10865/3084833
5.py:17: UserWarning: Matplotlib is currently using module://matplotlib_i
nline.backend_inline, which is a non-GUI backend, so cannot show the figu
re.
  fig.show()
```

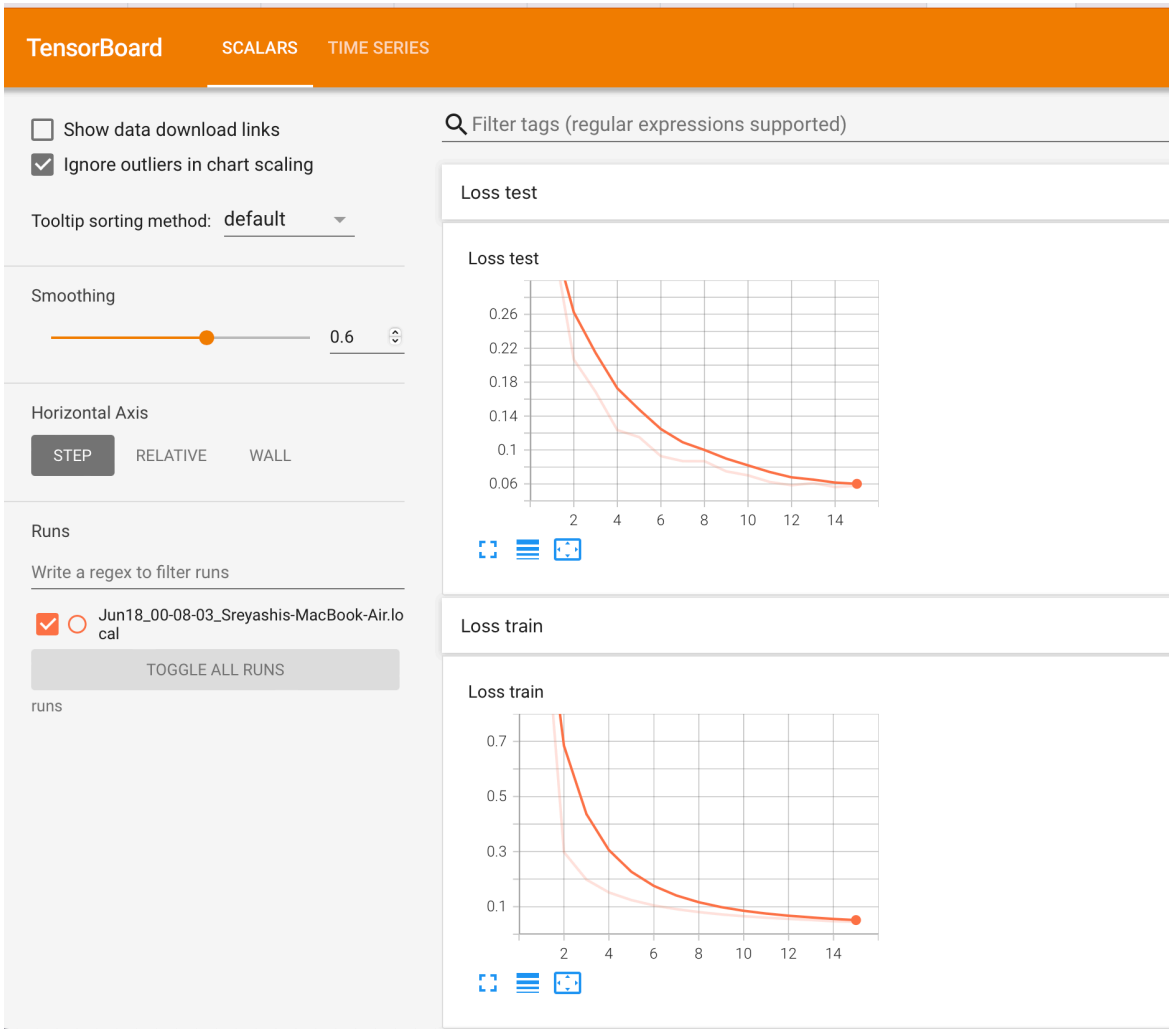


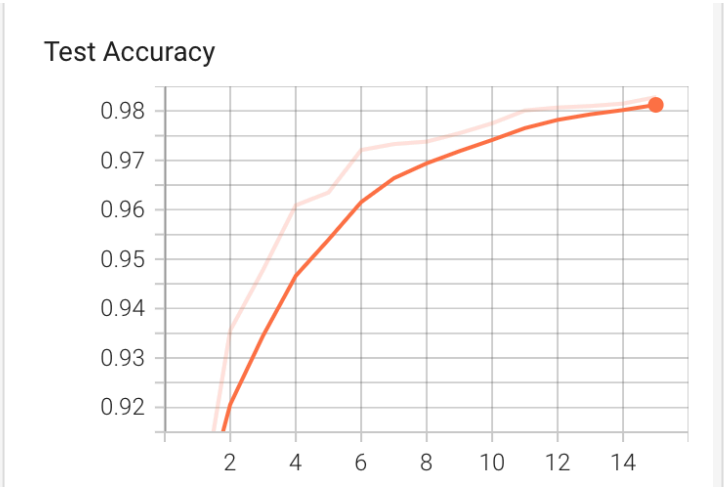
In the above case I have set my learning rate to be 0.1 and number of epochs as 15 and still we don't get a very good output as the results reach only 56% accuracy. Hence we can conclude that Lenet architechture is not modelled for CIFAR10 dataset. We can also see from the graph above that there is a lot of fluctuations in the loss obtained from the test data. We also see quite an amount of fluctuations in the accuracy of both the train and test datasets.

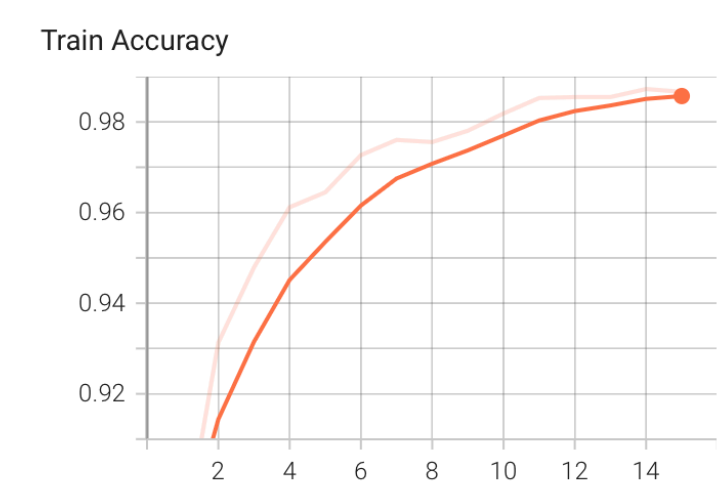
In [ ]:

Viewing Test and Train Losses as well as Test and Train Accuracies in Tensorboard:

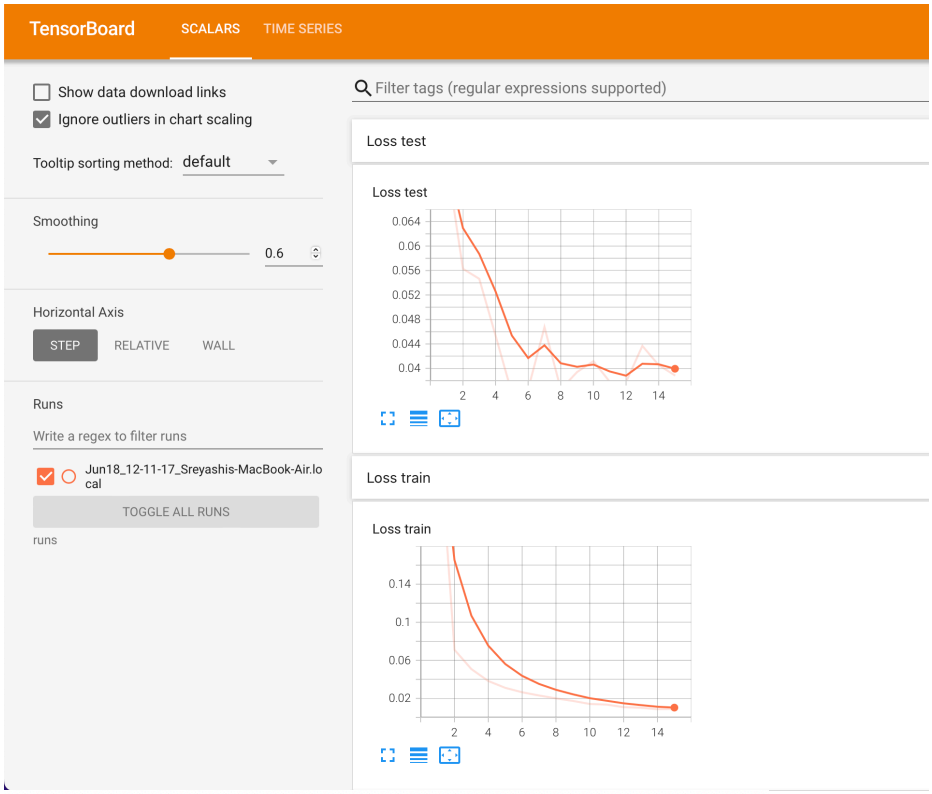
For MNIST DATASET: Optimizer = SGD Learning Rate = 0.01



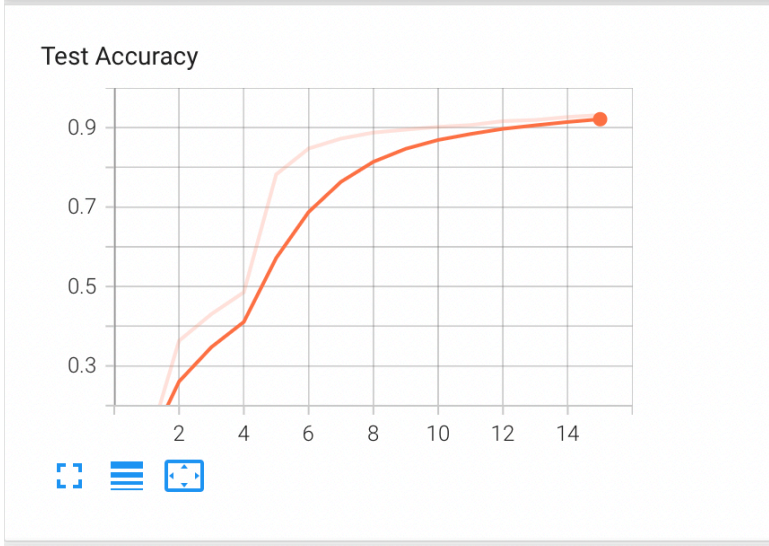




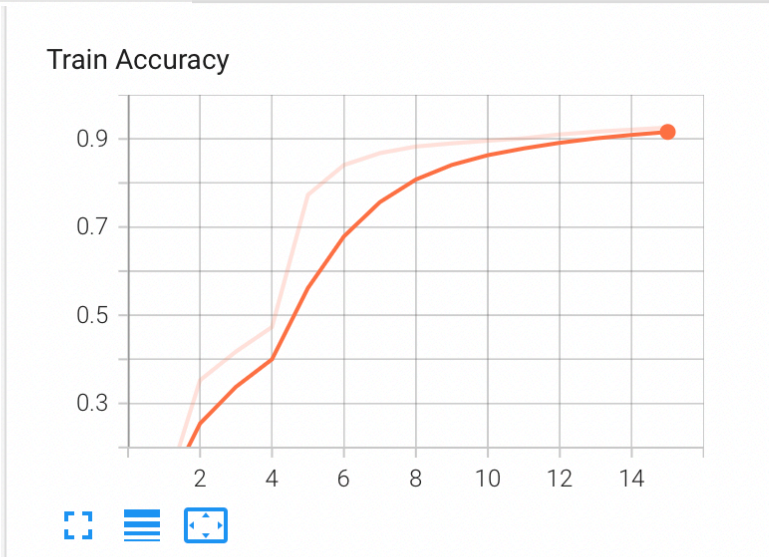
Optimizer = SGD Learning Rate = 0.1



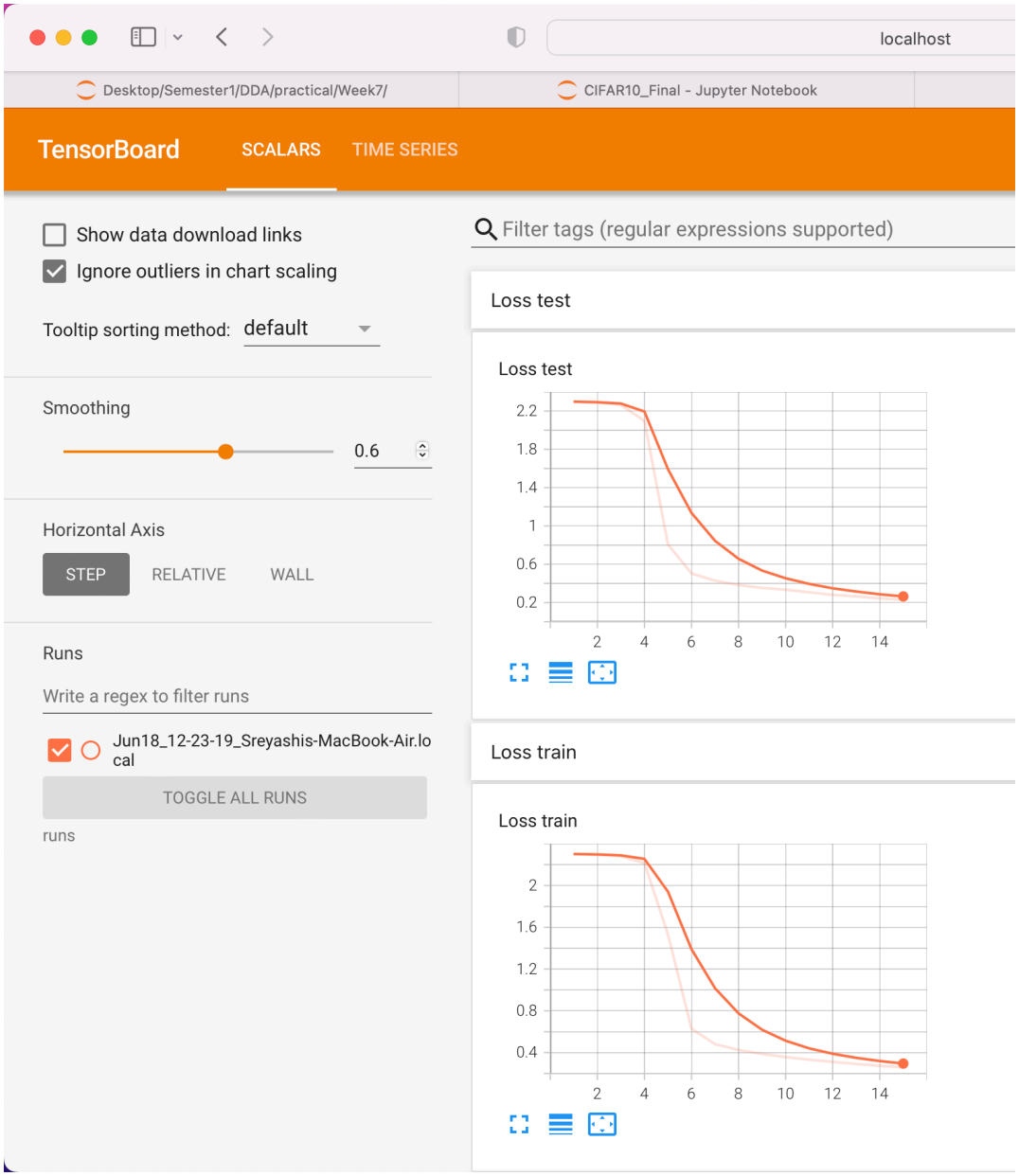
These are the plots for the loss obtained for both test and train datasets.



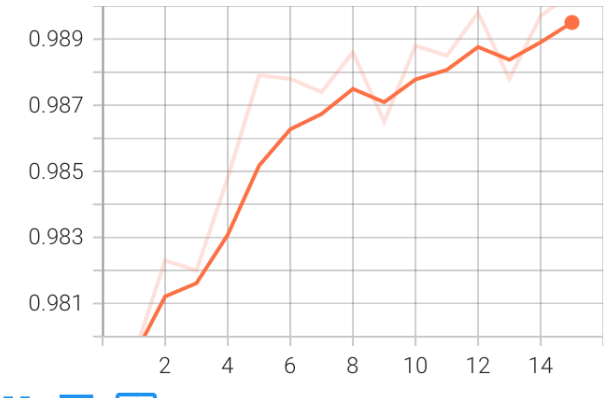
These are the accuracy plots for the train and test datasets respectively



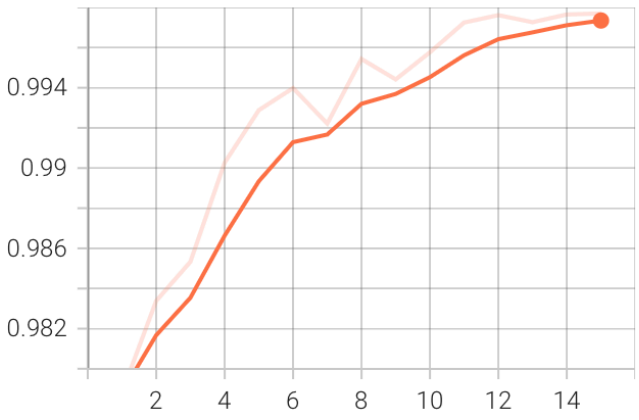
Optimizer = SGD Learning Rate = 0.001 Epochs = 15



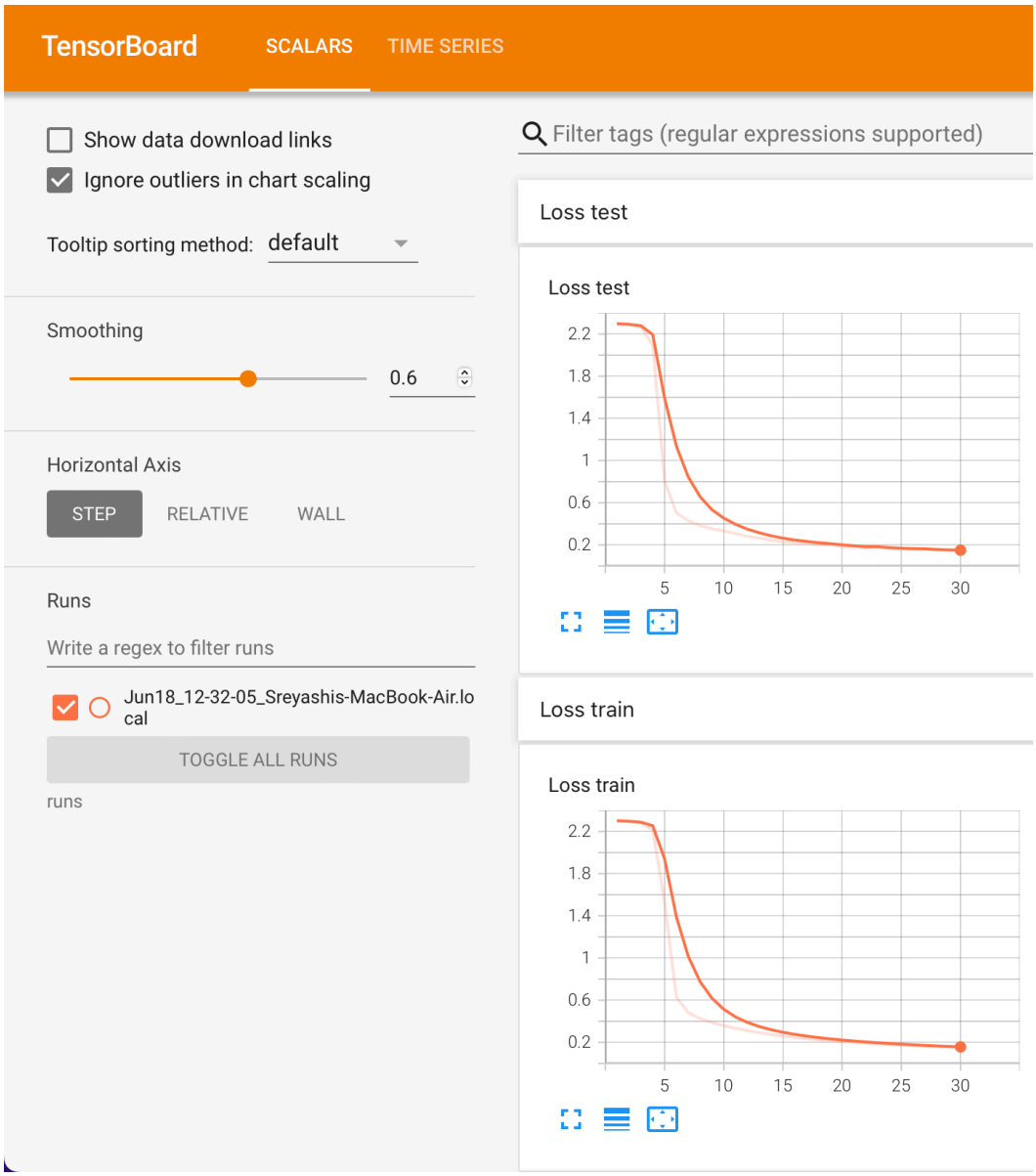
Test Accuracy



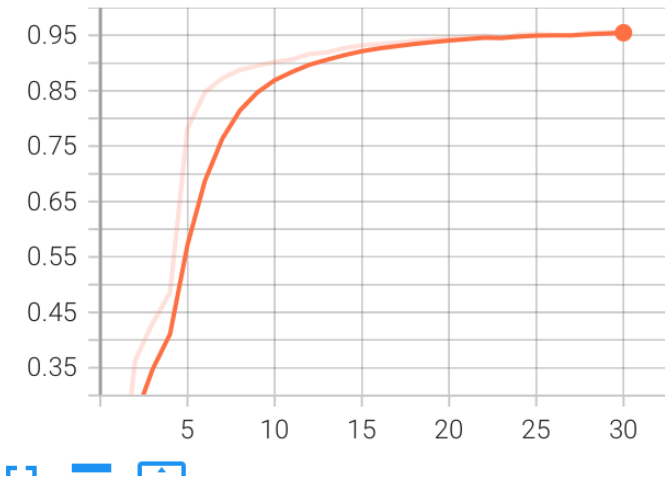
Train Accuracy



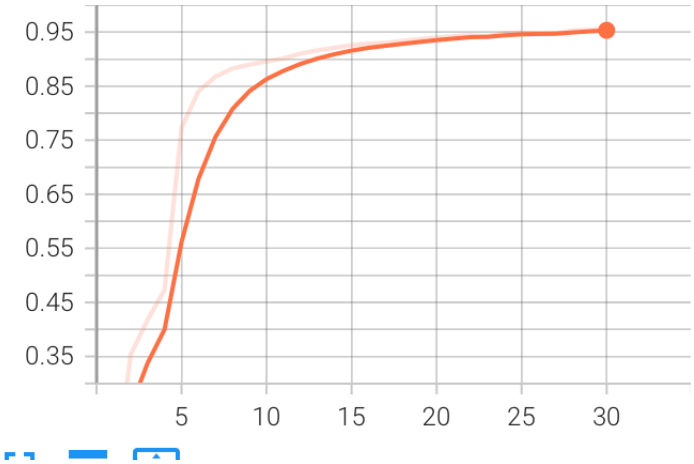
Optimizer = SGD Learning Rate = 0.001 Epochs = 30



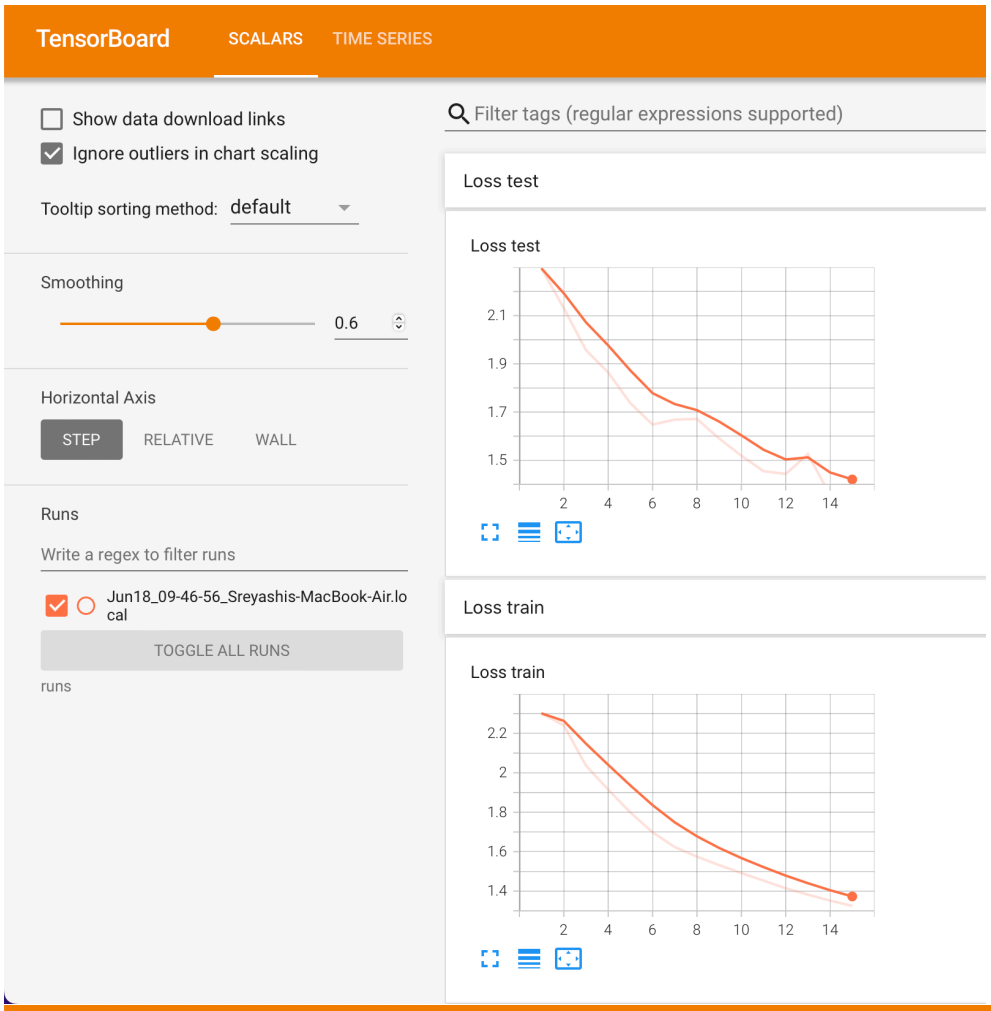
Test Accuracy

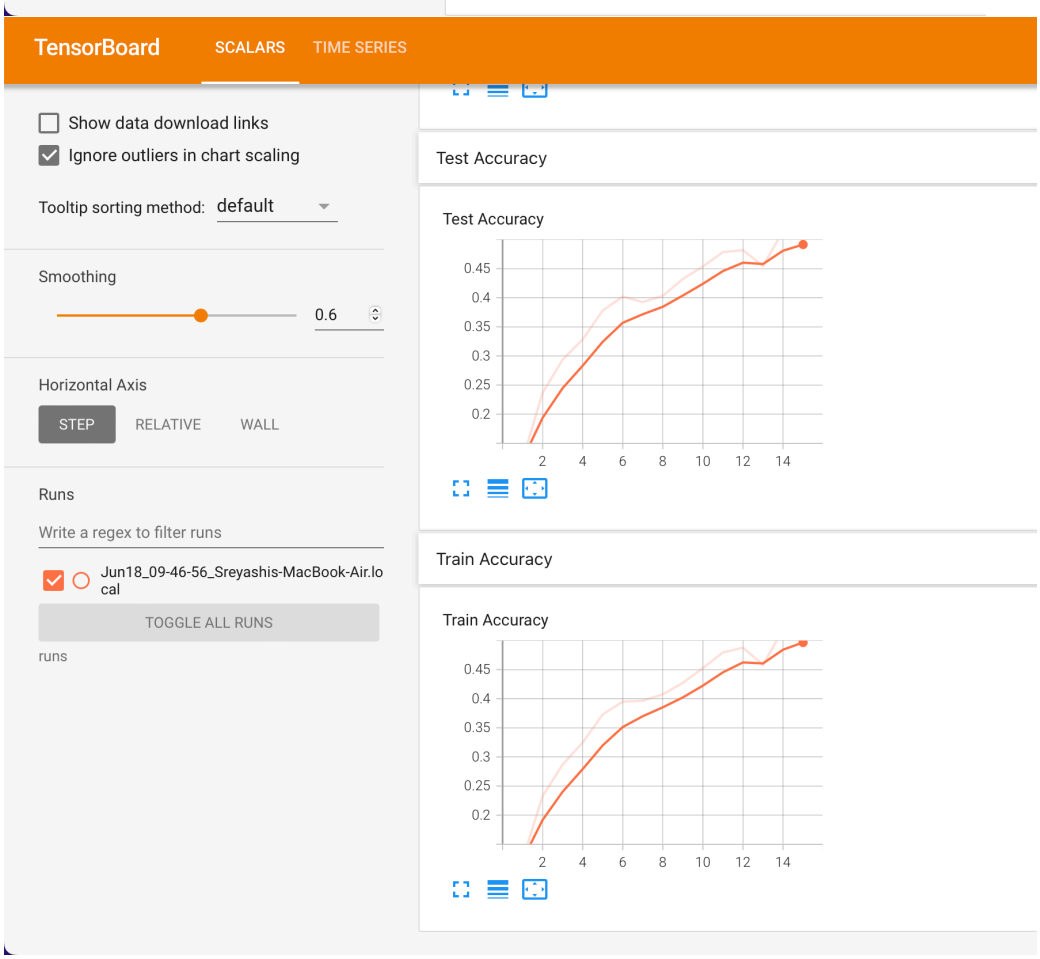


Train Accuracy

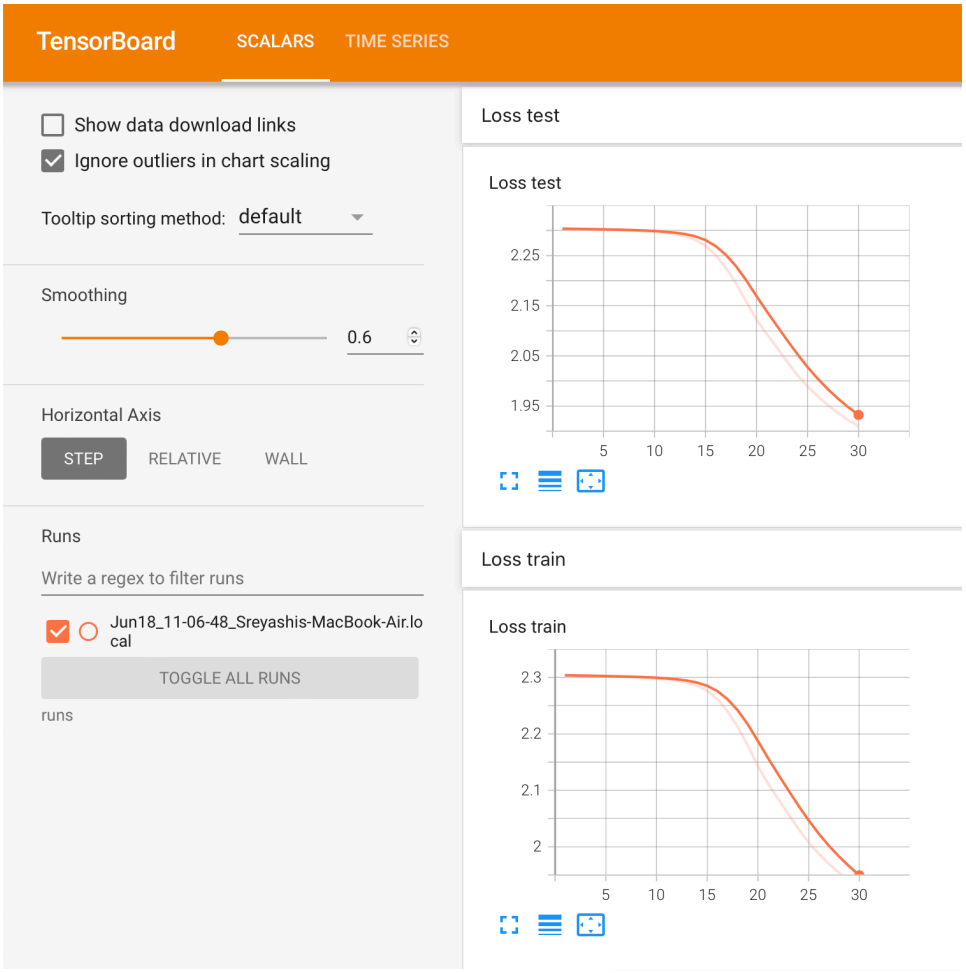


FOR CIFAR10 DATASET: Optimizer = SGD Learning Rate = 0.01

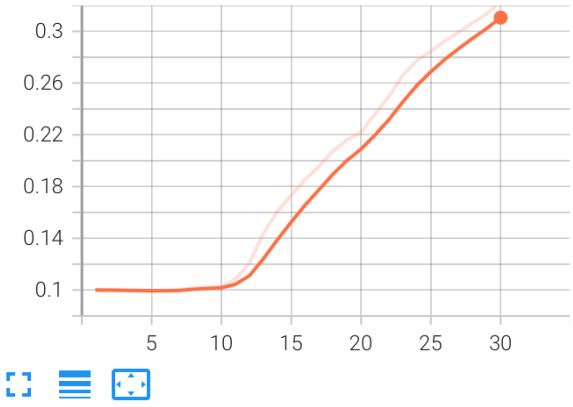




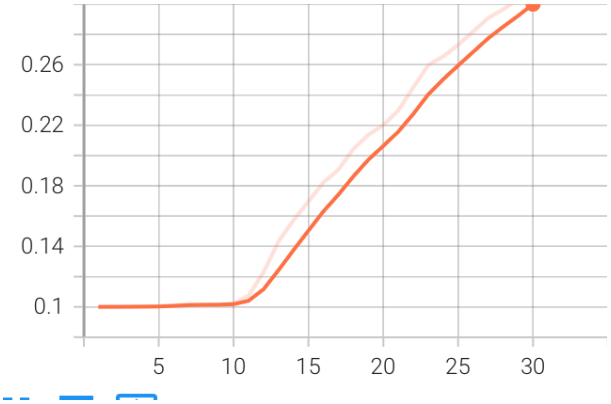
Optimizer = SGD Learning Rate = 0.001



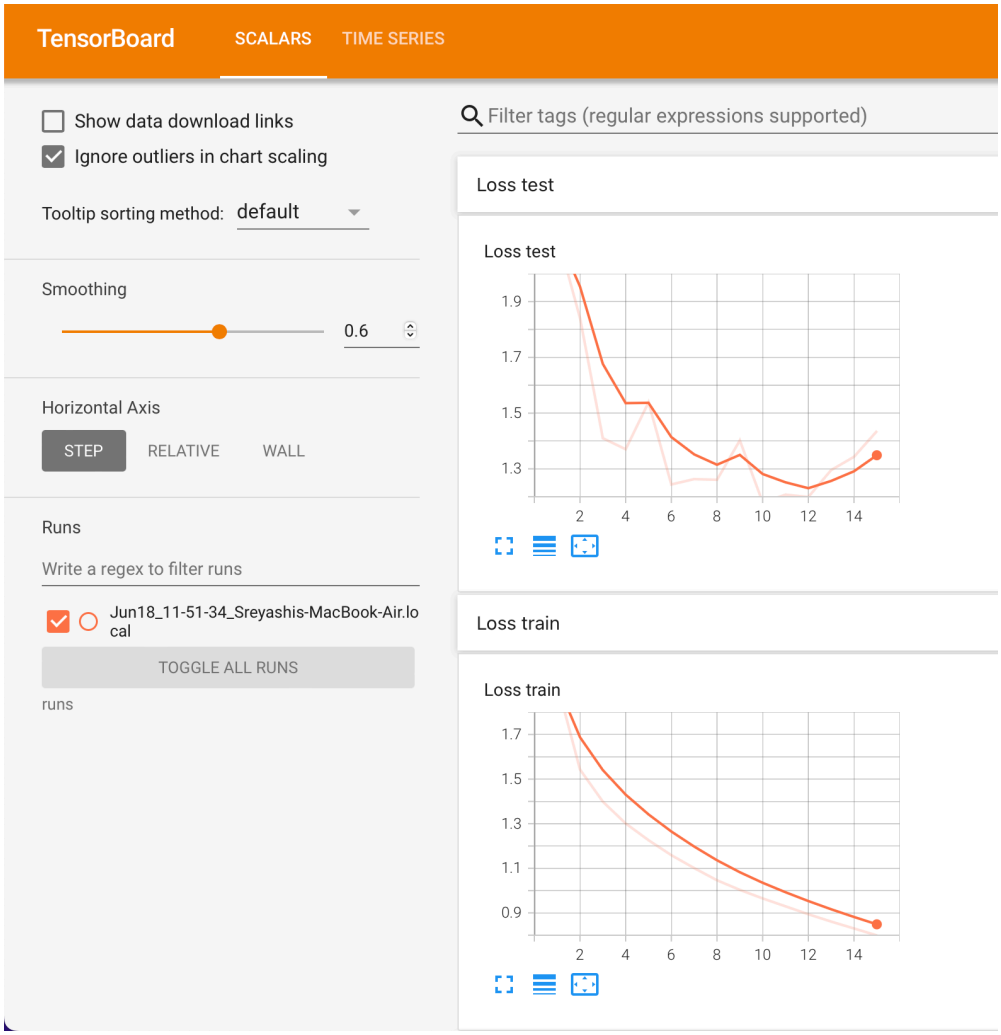
Test Accuracy



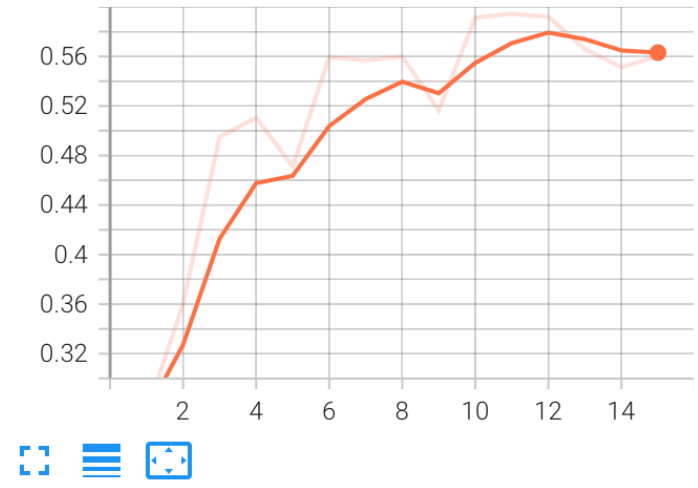
Train Accuracy



Optimizer = SGD Learning Rate = 0.1



Test Accuracy



Train Accuracy