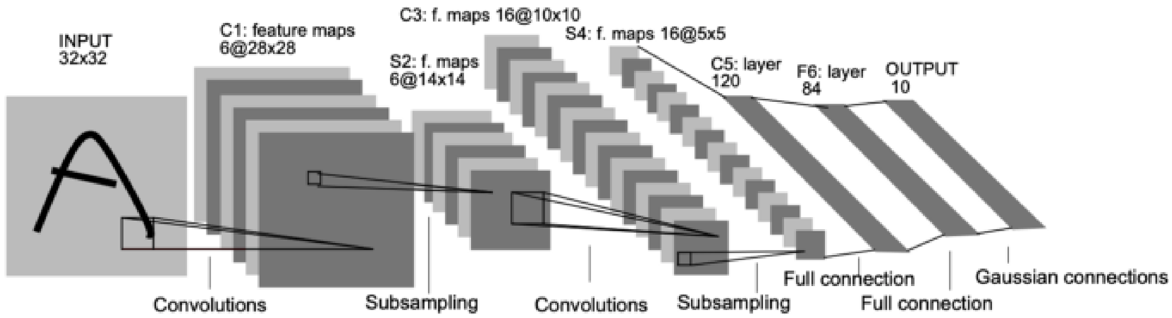
Sharon Laurance

Exercise 1: PyTorch Network Analysis

LeNet-5 is a 7-layer Convolutional Neural Network, trained on grayscale images of size 32 x 32 pixels.



MNIST

Steps required for Digit Prediction using MNIST Dataset.

Step 1: Import the required libraries
Step 2: Define the network parameters
Step 3: Define Transforms on Dataset (Transform it to Tensor and Resize the Image to 32 X 32). Download the datasets from torchvision.datasets. Load the datasets in the train and test loader functions.

```
# 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)
```

Step 4: Define the LeNet5 Network. The convolution, pooling and fully-connected layers are taken from the LeNet5 paper[1]. It takes an input of size (32 X 32)

```python
# Class LeNet which contains layers and forward pass- Convolution Block, Linear Block and Forward Pass
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, stride=1),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=2),
            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=2),
            nn.Conv2d(in_channels=16, out_channels=120, kernel_size=5, stride=1),
            nn.Tanh()
        )
        self.classifier = nn.Sequential(
            nn.Linear(in_features=120, out_features=84),
            nn.Tanh(),
            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
```

Step 5: Here the actual train model is defined.

```python
def train(train_loader, model, criterion, optimizer, device):
    '''
    Function for the training step of the training loop
    '''
    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()
    epoch_loss = running_loss / len(train_loader.dataset)
    return model, optimizer, epoch_loss
```

Step 6: Training_loop() defines the losses and is used to find out the accuracy.

```python
def training_loop(model, criterion, optimizer, train_loader, test_loader, epochs, device,lr, print_every=1):
    '''
    Function defining the entire training loop
    '''
    # set objects for storing metrics
    best_loss = 1e10
    train_losses = []
    test_losses = []
    lr=str(lr)
    # Train model
    for epoch in range(0, epochs):
        # training
        model, optimizer, train_loss = train(train_loader, model, criterion, optimizer, device)
        writer.add_scalar("Epoch/Training_Loss for Learning Rate "+lr, train_loss, epoch)
        train_losses.append(train_loss)
        # Testing
        with torch.no_grad():
            model, test_loss = validate(test_loader, model, criterion, device)
            test_losses.append(test_loss)
            writer.add_scalar("Epoch/Testing_Loss for Learning Rate "+lr, test_loss, epoch)
        if epoch % print_every == (print_every - 1):
            train_acc = get_accuracy(model, train_loader, device=device)
            valid_acc = get_accuracy(model, test_loader, device=device)
            writer.add_scalar("Epoch/Accuracy_Training for Learning Rate "+lr, train_acc, epoch)
            writer.add_scalar("Epoch/Accuracy_Testing for Learning Rate "+lr, valid_acc, epoch)
            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 * valid_acc:.2f}')
    writer.flush()
    return model, optimizer, (train_losses, test_losses)
```

Step 7: Optimiser is fixed and model performance is analysed for different learning rate.

```
# Adam Optimizer is used and learning rate will be varied as [0.1, 0.01, 0.001]
torch.manual_seed(RANDOM_SEED)
model = LeNet5(N_CLASSES).to(DEVICE)

LEARNING_RATE_arr = [0.1, 0.01, 0.001]
optimizer_arr=[]
for i in LEARNING_RATE_arr:
  optimizer = torch.optim.Adam(model.parameters(), lr=i)
  optimizer_arr.append(optimizer)
criterion = nn.CrossEntropyLoss()
```

```
[12] import torch
     from torch.utils.tensorboard import SummaryWriter
     writer = SummaryWriter()
```

Fig: Learning Rate=0.001 Optimiser: Adam Optimiser. Accuracy for Training and Testing increases. Loss for Training and Testing decreases exponentially. This is a good learning rate which gives optimum performance

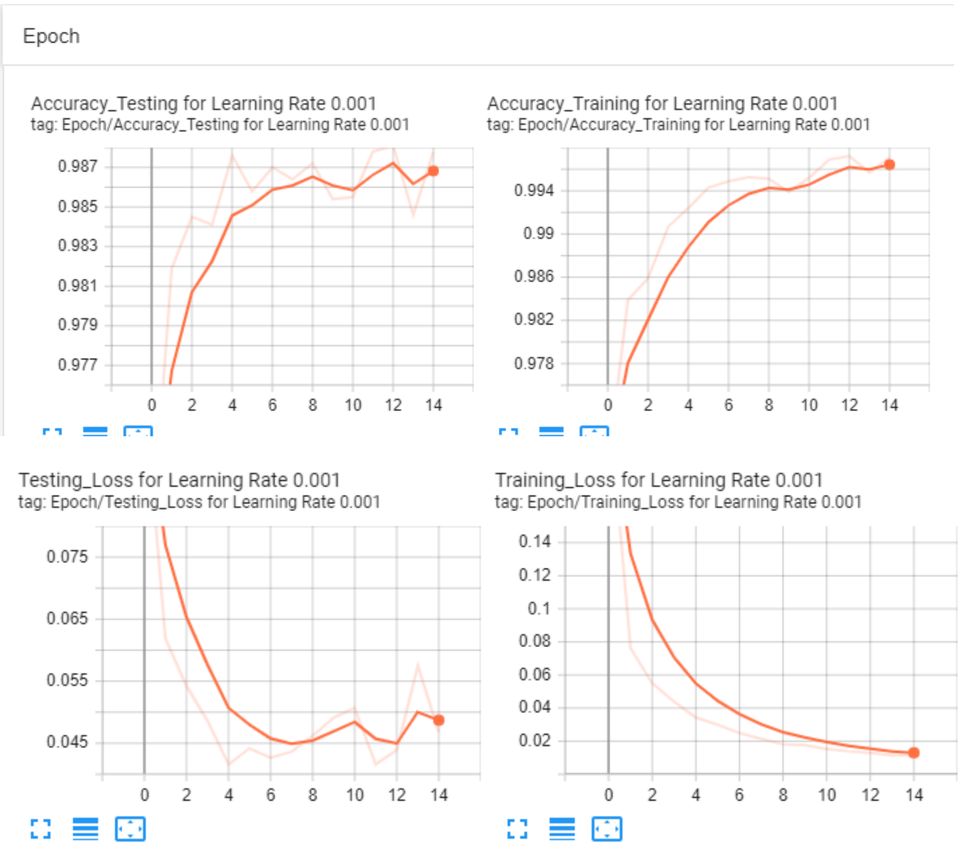


Fig: Learning Rate=0.01 Optimiser: Adam Optimiser. We can see that the graph is not that smooth. The smoothing factor here is 0.7

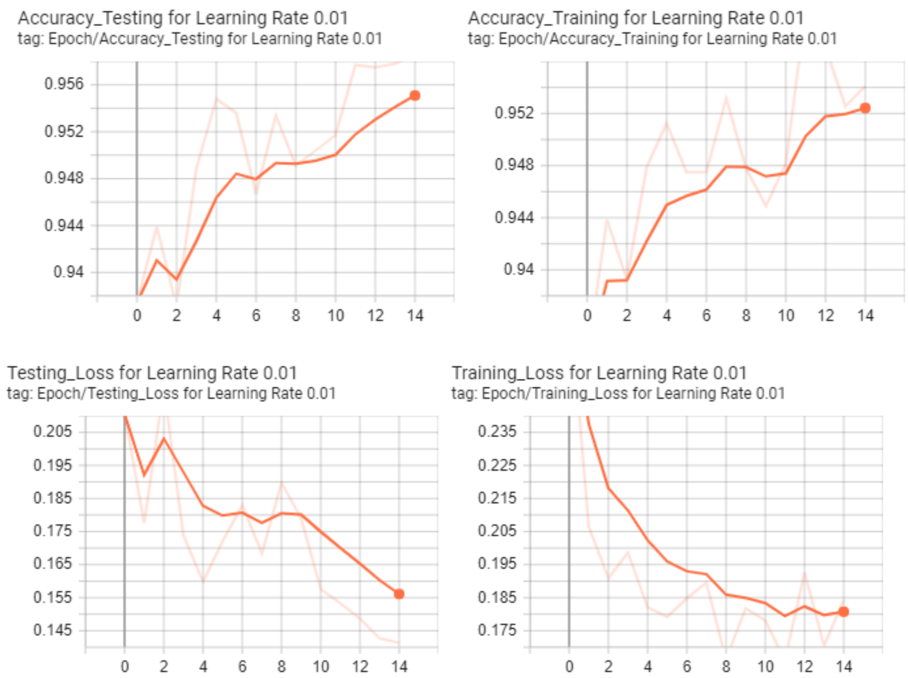
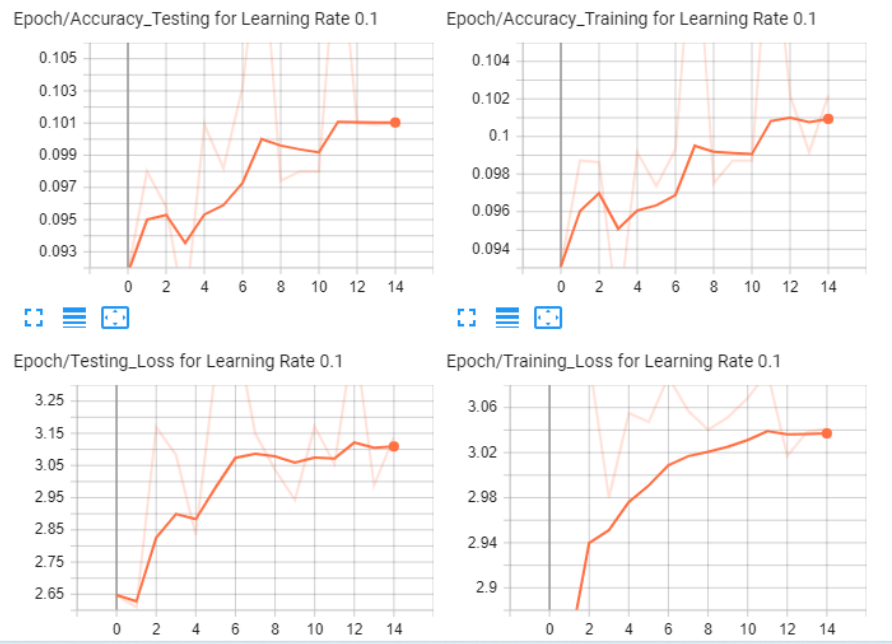


Fig: Learning Rate=0.1 Optimiser: Adam Optimiser. We can see having a huge learning rate affects the model performance. Hence, we can conclude that the learning rate influences model performance. Accuracy increases but the loss does not decrease much. Smoothing rate is 0.9 and testing is done for 15 epochs.



CIFAR10

Steps required for Prediction using CIFAR10 Dataset.

Step 1: Import the required libraries
Step 2: Define the network parameters

Step 3: Define Transforms on Dataset (Transform it to Tensor). Download the datasets from torchvision.datasets. Load the datasets in the train and test loader functions.
Length of training sample: 50000
Length of testing sample: 10000

```
path = '/tmp'

train_ds = datasets.CIFAR10(root=path,
                            train=True,
                            transform=train_augmentations,
                            download=True
                           )

test_ds = datasets.CIFAR10(root=path,
                           train=False,
                           transform=test_augmentations
                           )

labels = 'airplane automobile bird cat deer dog frog horse ship truck'.split()
```

```
Files already downloaded and verified
```

```
[29] len(train_ds), len(test_ds)
```

```
(50000, 10000)
```

```
[30] train_dataloader = torch.utils.data.DataLoader(train_ds, batch_size=256, shuffle=True, num_workers=2)
     test_dataloader = torch.utils.data.DataLoader(test_ds, batch_size=256, shuffle=True, num_workers=2)
```

Step 4: Define the LeNet model for CIFAR 10. The modification required is in_channel=3. This is because CIFAR10 has 3 channels (R, G,B)

```
model = LeNet().to(device)
print(model)
```

```
LeNet(
  (conv_block): Sequential(
    (0): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
    (1): Tanh()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
    (4): Tanh()
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (linear_block): Sequential(
    (0): Linear(in_features=400, out_features=120, bias=True)
    (1): Tanh()
    (2): Linear(in_features=120, out_features=84, bias=True)
    (3): Tanh()
    (4): Linear(in_features=84, out_features=10, bias=True)
  )
)
```

Step 5: Training class has three functions.

- accuracy()- Used to find training and testing accuracy.
- train_one_epoch()- Used to find training loss.
- valid_one_epoch() -Used to find testing loss.

```
def __init__(self, model, dataloaders, device,lr):
    self.config = {
        'lr':lr,
        'epochs': 10
    }
    self.model = model
    self.train_loader, self.test_loader = dataloaders
    self.loss_fn = nn.CrossEntropyLoss()
    self.optim = torch.optim.Adam(self.model.parameters(), lr = self.config['lr'])
    self.device = device
def accuracy(self, output, y):
    pred_labels = torch.argmax(output, dim=1)
    return (pred_labels == y).sum().item() / len(y)
```

```python
def train_one_epoch(self):
    running_loss = 0
    running_acc = 0
    for x,y in self.train_loader:
        self.optim.zero_grad()
        x = x.to(self.device, dtype=torch.float)
        y = y.to(self.device, dtype=torch.long)
        output = self.model(x)
        loss = self.loss_fn(output, y)
        loss.backward()
        self.optim.step()
        running_loss += loss.item()
        running_acc += self.accuracy(output,y)
        del x,y,output
    train_loss = running_loss/len(self.train_loader)
    train_acc = running_acc/len(self.train_loader)
    return train_loss, train_acc
```

```python
@torch.no_grad()
def valid_one_epoch(self):
    running_loss = 0
    running_acc = 0
    for x,y in self.test_loader:
        x = x.to(self.device, dtype=torch.float)
        y = y.to(self.device, dtype=torch.long)
        output = self.model(x)
        loss = self.loss_fn(output, y)
        running_loss += loss.item()
        running_acc += self.accuracy(output,y)
        del x,y,output
    test_loss = running_loss/len(self.test_loader)
    test_acc = running_acc/len(self.test_loader)
    return test_loss, test_acc
```

Step 6: Define the fit function which performs 10 epochs over the training dataset. Here add_scaler is used to print the losses to the tensorboard. Tensorboard is used for visualizing the Loss vs Epoch Graph and Accuracy vs Epoch Graph.

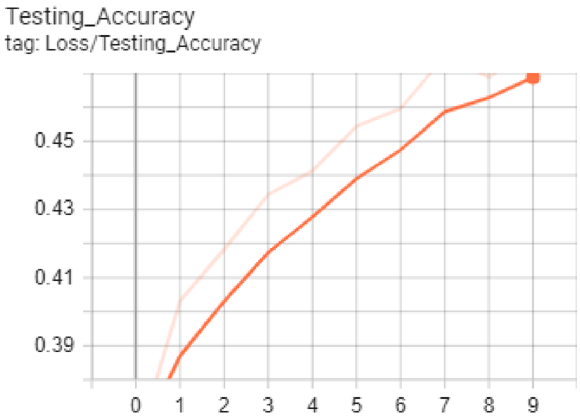
```python
def fit(self,lr):

    train_losses,train_accs = [], []
    test_losses, test_accs = [], []
    for epoch in range(self.config['epochs']):
        print(f"Model is using {'cuda' if next(self.model.parameters()).is_cuda else 'cpu'}")
        self.model.train()
        train_loss, train_acc = self.train_one_epoch()
        tb.add_scalar("Loss/Training_Loss", train_loss, epoch)
        tb.add_scalar("Loss/Training_Accuracy", train_acc, epoch)
        train_losses.append(train_loss)
        train_accs.append(train_acc)
        self.model.eval()
        test_loss, test_acc = self.valid_one_epoch()
        tb.add_scalar("Loss/Testing_Loss", test_loss, epoch)
        tb.add_scalar("Loss/Testing_Accuracy", test_acc, epoch)
        test_losses.append(test_loss)
        test_accs.append(test_acc)
        print(f"------EPOCH {epoch+1}/{self.config['epochs']}------")
        print(f"Training: LOSS: {train_loss} | ACCURACY: {train_acc} | for Learning Rate: {lr} ")
        print(f"Testing: LOSS: {test_loss} | ACCURACY: {test_acc} | for Learning Rate: {lr} \n\n")
        # CLEANUP
        gc.collect()
        torch.cuda.empty_cache()
        writer.flush()
    return (train_losses, train_accs), (test_losses, test_accs)
```

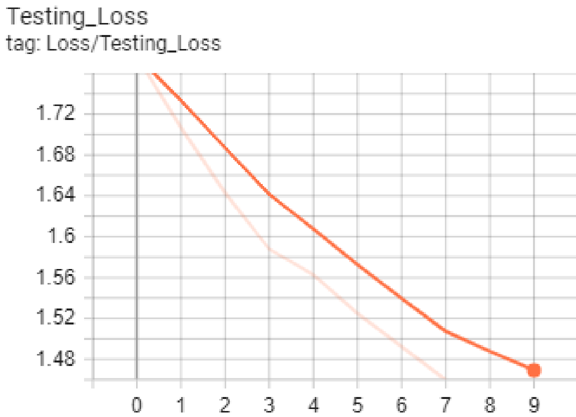
Step 7: Initialize the model and push it to cuda.

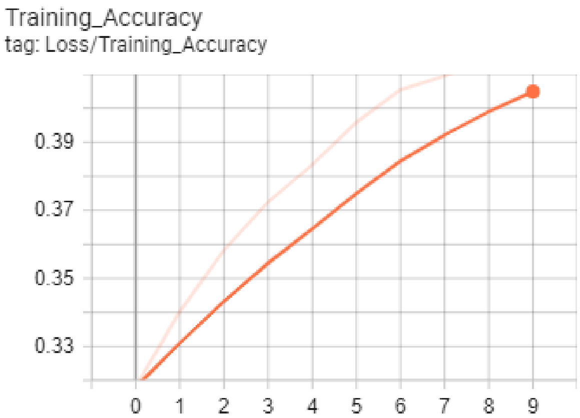
```python
lr_arr=[0.1,0.001,0.0001]
trainer = Trainer(model, (train_dataloader, test_dataloader), device,lr_arr[0])
(train_losses, train_accs), (test_losses, test_accs) = trainer.fit(lr_arr[0])
```

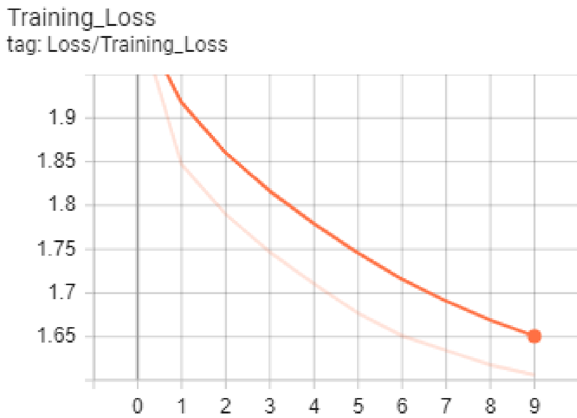
Output Graph for CIFAR 10:

For Learning Rate : 0.001 Optimiser= Adam Optimiser. Accuracy for Training and Testing increases. Loss for Training and Testing decreases exponentially. This is a good learning rate which gives optimum performance

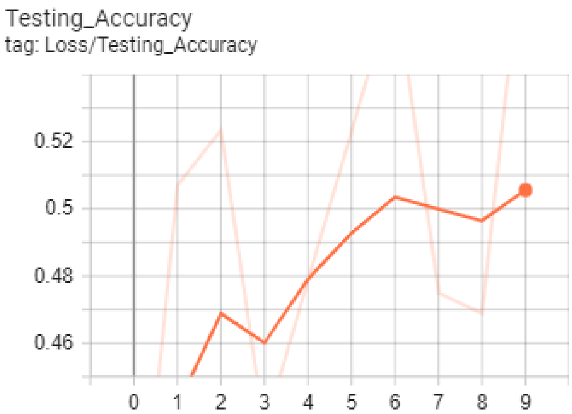


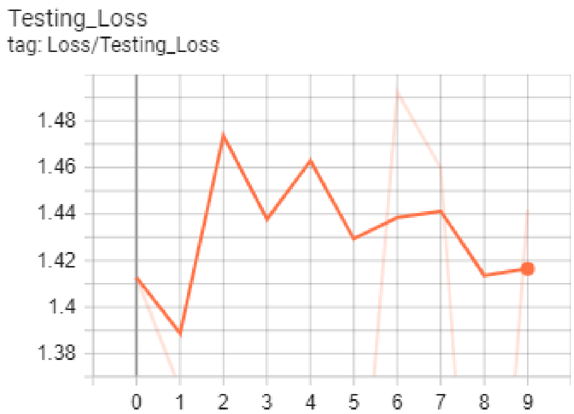


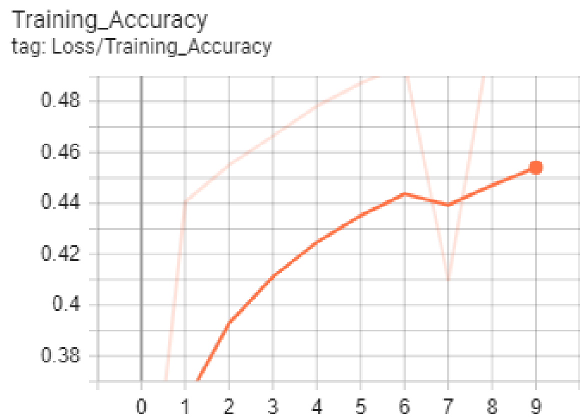


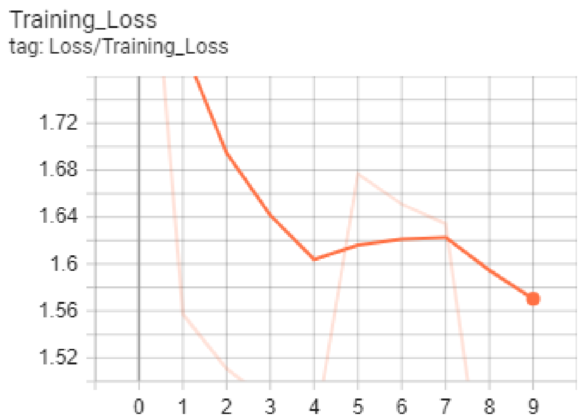


For Learning Rate : 0.01 Optimiser= Adam Optimiser. The increase in accuracy or decrease in loss is not exponential. The higher learning_rate is affecting the performance of the model.

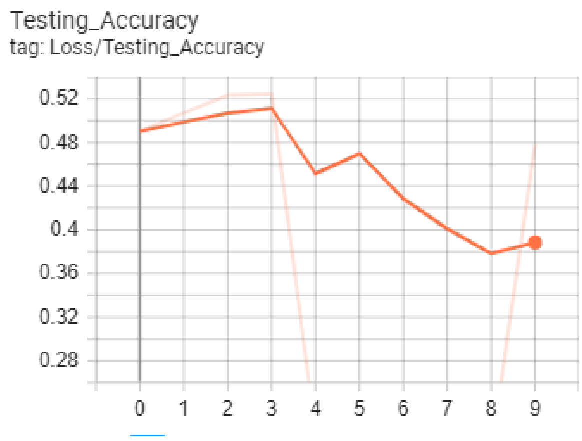


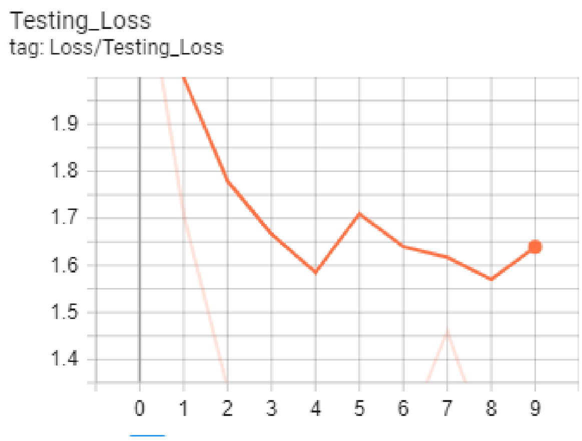


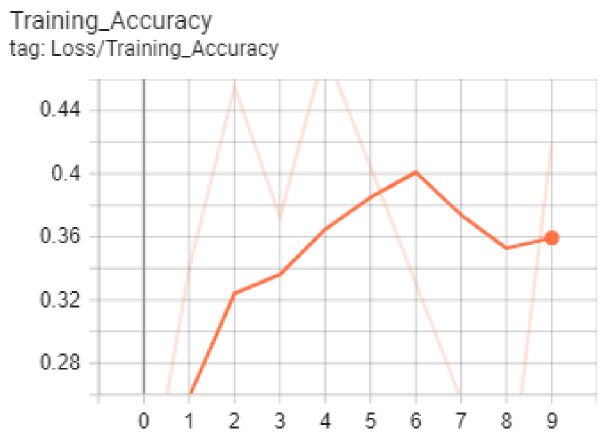


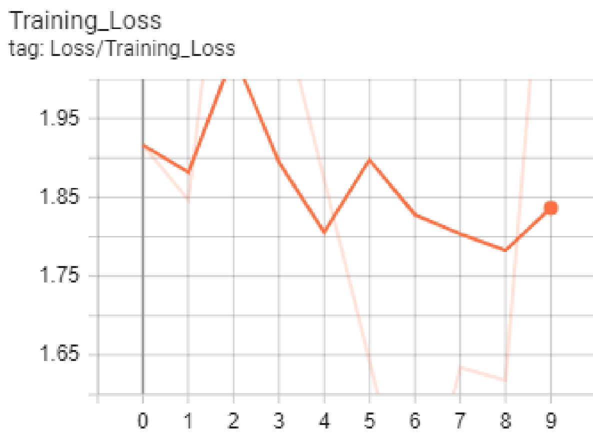


For Learning Rate : 0.1 Optimiser= Adam Optimiser. Test accuracy goes down. This is because of the learning rate. Test and Training loss both decrease, but decrease is not smooth. Smoothing parameter is 0.99.









Exercise 2: Custom Task

The formula for calculating the output size of the convolutional layer : $(W-F+2P)/S+1$

where W is the input height/width (normally the images are squares, so there is no need to differentiate the two), F is the filter/kernel size, P is the padding, and S is the stride.

Steps required for prediction of addition of two numbers using MNIST Dataset.

Step 1: Load the Libraries
Step 2: Initialize the Parameters

```
#Initialize the Parameters
RANDOM_SEED = 42
LEARNING_RATE = 0.001
BATCH_SIZE = 32
N_EPOCHS = 15
IMG_SIZE = 32
```

Step 3: Define Transforms on Dataset (Transform it to Tensor and Resize the Image to 32 X 32)
Step 4: Download and Create Datasets datasets.MNIST. For loading, the Dataset DataLoader is used. DataLoader has a parameter called collate_fn. This parameter allows you to create separate data processing functions and will apply the processing within that function to the data before it is output. Here collate_batch is used pre-processing. For K=3 and N=10, the batch_size is given as N*K. The images are stacked over each other using torch.stack(). x.view() function is used to reshape it to [N,K,32,32].

```
# 1)Define Transforms on Dataset (Transform it to Tensor and Resize the Image)
transforms = transforms.Compose([transforms.Resize((32, 32)),
                                 transforms.ToTensor()])

# 2)Download and Create Datasets datasets.MNIST
train_dataset = datasets.MNIST(root='mnist_data',
                               train=True,
                               transform=transforms,
                               download=True)

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

```
def collate_batch(batch):
  train_targets = []
  input_tensors=[]
  for item in batch:
    input_tensors.append(item[0])
    train_targets.append(item[1])
  c=torch.stack(input_tensors).view(N,K,32,32)
  train_targets = torch.tensor(train_targets).view(N,K).sum(axis=1).type(torch.float)
  return [c,train_targets]
```

```
[7] train_loader = DataLoader(train_dataset, batch_size=N*K, collate_fn=collate_batch,shuffle=True,drop_last=True)
    test_loader = DataLoader(test_dataset, batch_size=N*K, collate_fn=collate_batch,shuffle=False,drop_last=True)
```

Step 5: The model is defined as follows. The convolution, pooling and fully-connected layers are taken from the LeNet5 paper[1]. In the forward pass, view() function is again used with the same data with a different shape. In the final step, addition of K numbers is done using sum() function.

```python
class LeNet5(nn.Module):
    def __init__(self,K):
        self.K=K
        super(LeNet5, self).__init__(
        )
        self.conv_block = nn.Sequential( nn.Conv2d(in_channels=1,
                                                   out_channels=6,
                                                   kernel_size=5,
                                                   stride=1),
                                         nn.Tanh(),
                                         nn.MaxPool2d(2,2),

                                         nn.Conv2d(in_channels=6,
                                                   out_channels=16,
                                                   kernel_size=5,
                                                   stride=1),
                                         nn.Tanh(),
                                         nn.MaxPool2d(2,2)
                                        )

        self.linear_block = nn.Sequential( nn.Linear(16*5*5, 120),
                                           nn.Tanh(),
                                           nn.Linear(120,84),
                                           nn.Tanh(),
                                           nn.Linear(84,1)
                                         )
    def forward(self, x):
        x = x.view(-1,1,32,32)
        x = self.conv_block(x)
        x = torch.flatten(x,1)
        # print("Before going to linear_block input is",x.shape)
        x = x.view(-1,self.K,400)
        # print("After reshape x is",x.shape)
        x = self.linear_block(x)
        x = x.view(-1,self.K).sum(axis=1)
        return x
```

Step 6: Train () function is where training takes place.

```python
def train(train_loader, model, criterion, optimizer, device):
    '''
    Function for the training step of the training loop
    '''
    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()
    epoch_loss = running_loss / len(train_loader.dataset)
    return model, optimizer, epoch_loss
```

```python
def validate(test_loader, model, criterion, device):
    '''
    Function for the testing step of the training loop
    '''

    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)
    epoch_loss = running_loss / len(test_loader.dataset)
    return model, epoch_loss
```

Step 7: training_loop performing training process. It has 15 epochs. Here add_scaler is used to print the losses to the tensorboard. Tensorboard is used for visualizing the Loss vs Epoch Graph

```python
def training_loop(model, criterion, optimizer, train_loader, test_loader, epochs, device, print_every=1):
    '''
    Function defining the entire training loop
    '''
    # set objects for storing metrics
    best_loss = 1e10
    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("Epoch/Training_Loss", train_loss, epoch)
        train_losses.append(train_loss)
        # Testing
        with torch.no_grad():
            model, test_loss = validate(test_loader, model, criterion, device)
            test_losses.append(test_loss)
            writer.add_scalar("Epoch/Testing_Loss", test_loss, epoch)
        if epoch % print_every == (print_every - 1):
            print(f'{datetime.now().time().replace(microsecond=0)} --- '
                  f'Epoch: {epoch}\t'
                  f'Training loss: {train_loss:.4f}\t'
                  f'Testing loss: {test_loss:.4f}\t')
    return model, optimizer, (train_losses, test_losses)
```

Step 8: Finally, the losses are printed.

```python
model, optimizer, _ = training_loop (model, criterion, optimizer, train_loader, test_loader, N_EPOCHS, DEVICE)

imgs,_ = next(iter(test_loader))
outputs = model1(imgs)
for i in range(10):
  writer.add_images(f'batch {i}', imgs[i].view(-1,1,32,32))
  writer.add_scalar(f'output {i}',outputs[i],0)
writer.flush()
writer.close()
```

```
14:00:59 --- Epoch: 0   Training loss: 3.1684   Testing loss: 0.8567
14:01:13 --- Epoch: 1   Training loss: 0.5862   Testing loss: 0.5083
14:01:27 --- Epoch: 2   Training loss: 0.3744   Testing loss: 0.3230
14:01:41 --- Epoch: 3   Training loss: 0.2756   Testing loss: 0.2621
14:01:55 --- Epoch: 4   Training loss: 0.2246   Testing loss: 0.3031
14:02:09 --- Epoch: 5   Training loss: 0.1858   Testing loss: 0.2356
14:02:23 --- Epoch: 6   Training loss: 0.1649   Testing loss: 0.2444
14:02:37 --- Epoch: 7   Training loss: 0.1359   Testing loss: 0.2800
14:02:52 --- Epoch: 8   Training loss: 0.1190   Testing loss: 0.1989
14:03:06 --- Epoch: 9   Training loss: 0.1048   Testing loss: 0.2115
14:03:20 --- Epoch: 10  Training loss: 0.0912   Testing loss: 0.2306
14:03:34 --- Epoch: 11  Training loss: 0.0844   Testing loss: 0.2442
14:03:48 --- Epoch: 12  Training loss: 0.0797   Testing loss: 0.1818
14:04:02 --- Epoch: 13  Training loss: 0.0742   Testing loss: 0.1904
14:04:16 --- Epoch: 14  Training loss: 0.0595   Testing loss: 0.2106
```

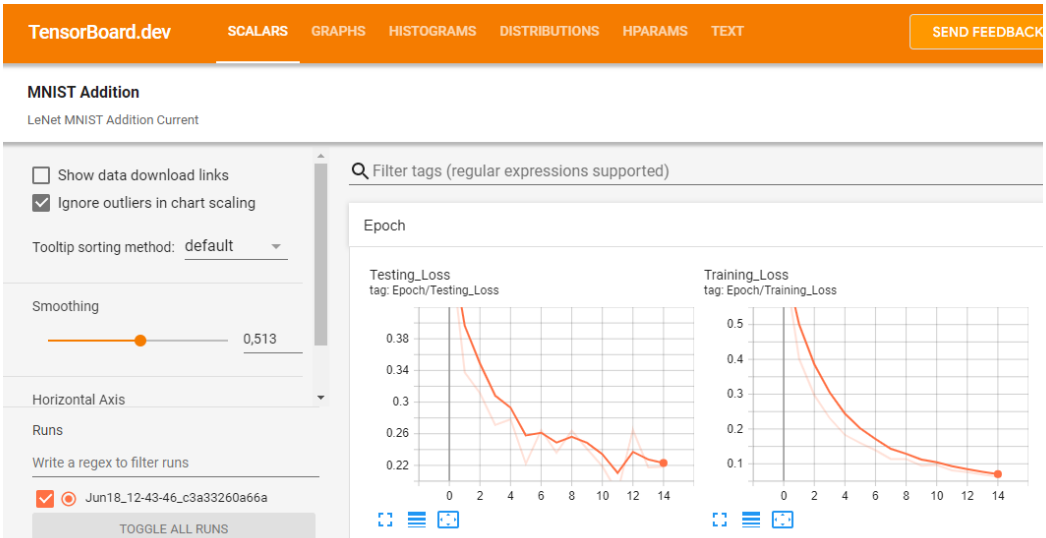
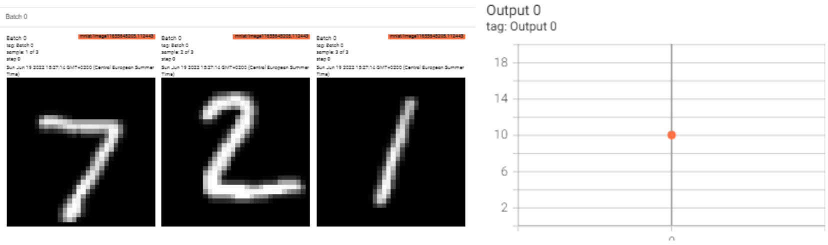
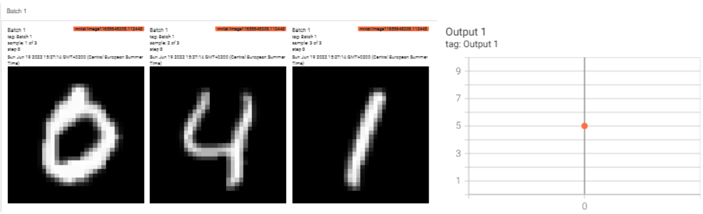


Fig: It shows training and testing loss. As we can see the losses are decreasing with the number of epochs. The smoothing factor is 0.5

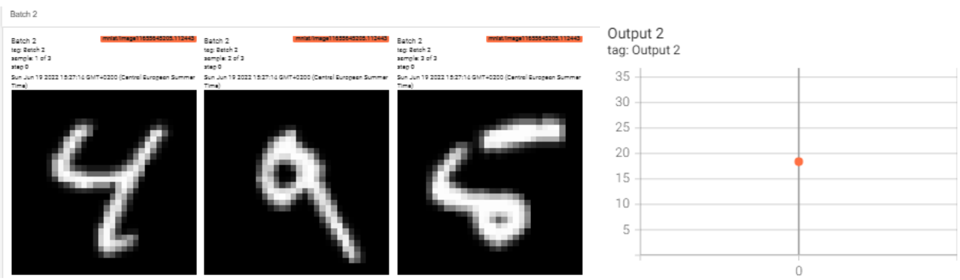
Batch 0: Here K=3



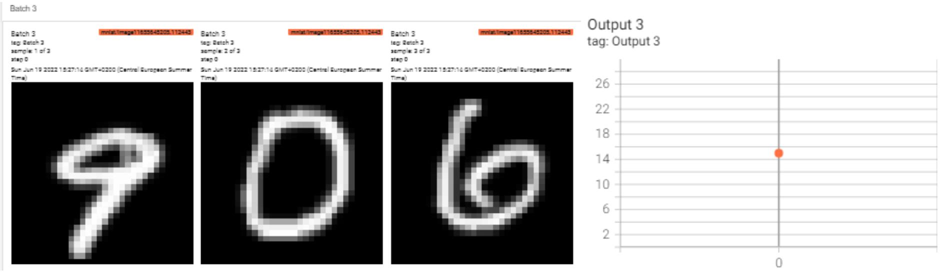
Batch 1:



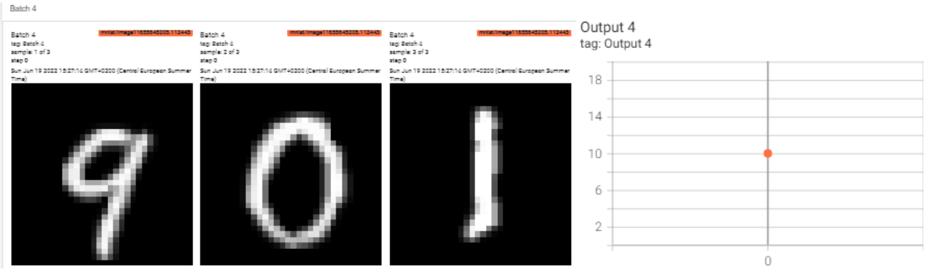
Batch 2:



Batch 3:

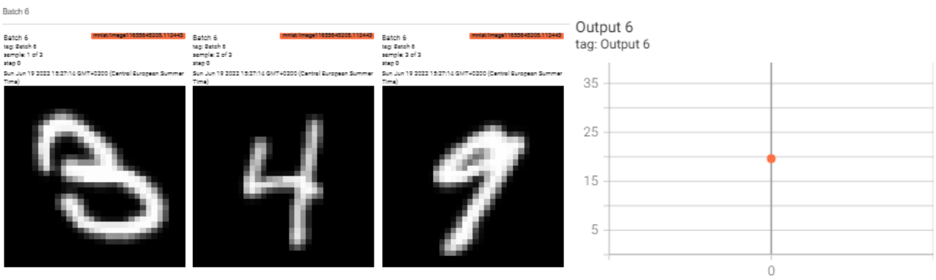


Batch 4:

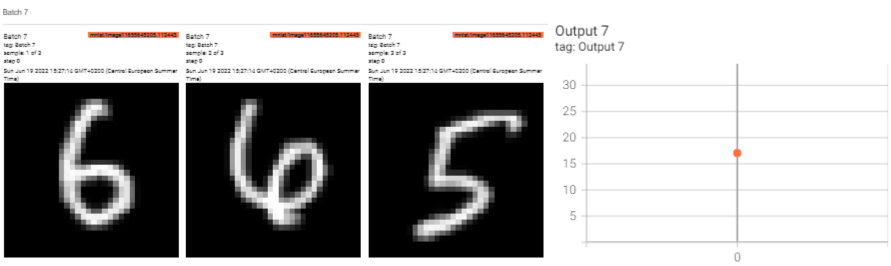


Batch 5:

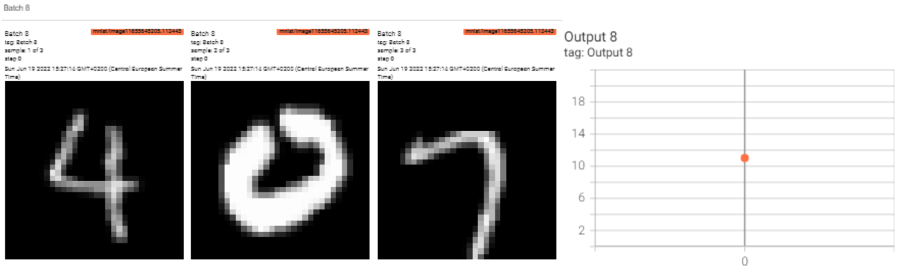
Batch 6:



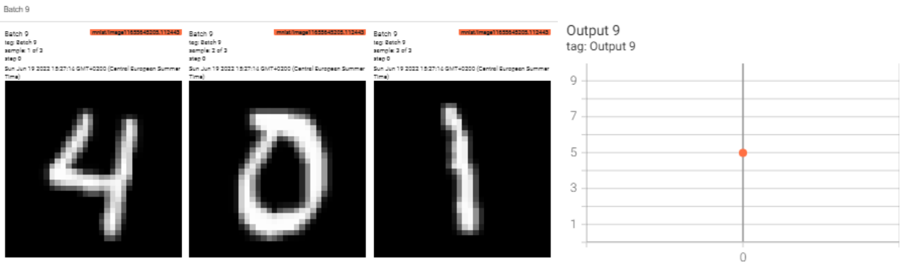
Batch 7:



Batch 8:



Batch 9:



References:

[1] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278-2324.

[2] Gaddam, S. (2022) https://www.kaggle.com/code/shreydan/lenet-5-cifar10-pytorch/notebook

[3] Lewinson, E. (2020) https://towardsdatascience.com/implementing-yann-lecuns-lenet-5-in-pytorch-5e05a0911320

[4] WherlI (2021) https://towardsdatascience.com/how-to-use-datasets-and-dataloader-in-pytorch-for-custom-text-data-270eed7f7c00