Sharon Laurance

Network Analysis: Image Classification

CNN is implemented for an image classification task. The baseline network is complex and it overfits the data. Only 50% of training samples are considered.

The design of the network is as stated in the question.

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

```
Custom_Network(
  (conv_block): Sequential(
    (0): Conv2d(3, 6, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(6, 12, kernel_size=(3, 3), stride=(1, 1))
    (4): ReLU()
    (5): Conv2d(12, 16, kernel_size=(3, 3), stride=(1, 1))
    (6): ReLU()
    (7): 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): ReLU()
    (2): Linear(in_features=120, out_features=84, bias=True)
    (3): ReLU()
    (4): Linear(in_features=84, out_features=10, bias=True)
    (5): ReLU()
  )
)
```

Code for CIFAR-10 object prediction:

Step 1: Import required libraries

```
# imports the required libraries

import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import matplotlib.pyplot as plt
import gc
```

Step 2: Download the data and transform it to tensor. Only half of training samples are considered for the baseline model. No other transformations are applied.

```
[2] #Only transformation carried out is converting to tensor. No other transformations are done
    train_augmentations = transforms.Compose([transforms.ToTensor()])

    test_augmentations = transforms.Compose([transforms.ToTensor()])
```

```
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()

train_dataset = torch.utils.data.Subset(train_ds, [i for i in range(25000)])
```

```
len(train_dataset), len(test_ds)
```

```
(25000, 10000)
```

Step 3: Define the model for training. This contains Conv2d layers, MaxPool layers and Fully-connected layers. Final Softmax function is not used as nn.CrossEntropyLoss() has a Softmax capability already.

```
class Custom_Network(nn.Module):

    def __init__(self, in_features=3, num_classes=10):
        super(Custom_Network, self).__init__()

        self.conv_block = nn.Sequential( nn.Conv2d(in_channels=in_features,out_channels=6,kernel_size=3,stride=1),
                                         nn.ReLU(),
                                         nn.MaxPool2d(2,2),
                                         nn.Conv2d(in_channels=6,out_channels=12,kernel_size=3,stride=1),
                                         nn.ReLU(),
                                         nn.Conv2d(in_channels=12,out_channels=16,kernel_size=3,stride=1),
                                         nn.ReLU(),
                                         nn.MaxPool2d(2,2)
                                        )

        self.linear_block = nn.Sequential( nn.Linear(16*5*5, 120),
                                           nn.ReLU(),
                                           nn.Linear(120,84),
                                           nn.ReLU(),
                                           nn.Linear(84,10),
                                           nn.ReLU()
                                          )

    def forward(self, x):
        x = self.conv_block(x)
        x = torch.flatten(x,1)
        x = self.linear_block(x)
        return x
```

Step 4: Define the training function

```
    def train_one_epoch(self,regulariser):
        running_loss = 0
        running_acc = 0
        lambda_l1=0.001
        lambda_l2=0.001
        reg_loss=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)
            if regulariser=='L1':
              reg_loss = sum(p.abs().sum()for p in model.parameters())
              loss +=lambda_l1*reg_loss
            elif regulariser=='L2':
              reg_loss = sum(p.pow(2.0).sum()for p in model.parameters())
              loss +=lambda_l2*reg_loss
            else:
              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
```

Function for computing testing loss at each epoch.

```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
```

```python
def fit(self,regulariser):
    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'}")
        '''
        This is a very important step while using dropout. This tells the model that it is in training model. For network regularization,
        dropout is only used while training and not while evaluation.
        '''
        self.model.train()
        train_loss, train_acc = self.train_one_epoch(regulariser)
        tb.add_scalars("Loss With Regularization/Training_Loss", {'train_loss_L2_Reg':train_loss}, epoch)
        tb.add_scalars("Accuracy With Regularization/Training_Accuracy", {'train_acc_L2_Reg':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_scalars("Loss With Regularization/Testing_Loss", {'test_loss_L2_Reg':test_loss}, epoch)
        tb.add_scalars("Accuracy With Regularization/Testing_Accuracy", {'test_acc_L2_Reg':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}")
        print(f"Testing: LOSS: {test_loss} | ACCURACY: {test_acc}\n\n")
        # CLEANUP
        gc.collect()
        torch.cuda.empty_cache()
        tb.flush()
    return (train_losses, train_accs), (test_losses, test_accs)
```

As discussed in lab, loss and accuracy is reported for each epoch.

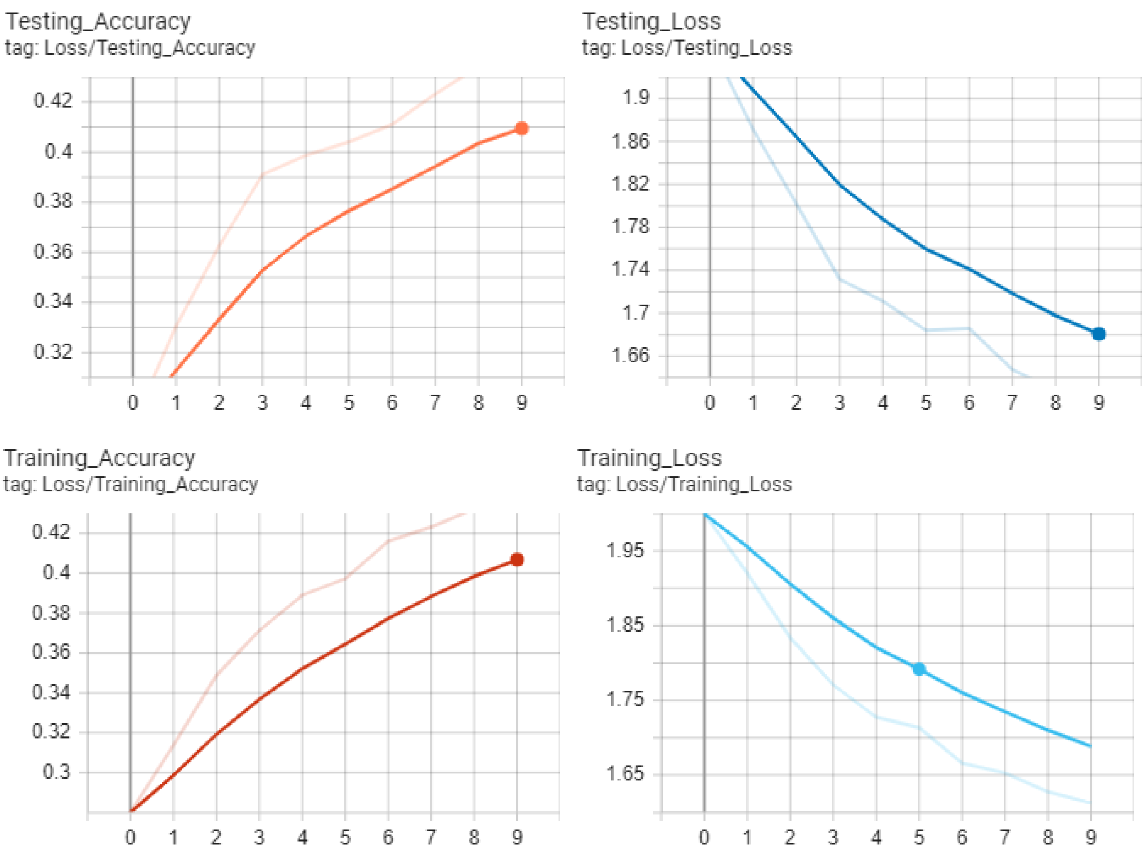
Step 5: fit() function is called and all model parameters are passed inside. Final testing accuracy of baseline model is 43% and test loss is 1.62.

```
[13] from torch.utils.tensorboard import SummaryWriter
     import torchvision

     tb = SummaryWriter()
     tb.close()
```

```
trainer = Trainer(model, (train_dataloader, test_dataloader), device)
regulariser="None"
(train_losses, train_accs), (test_losses, test_accs) = trainer.fit(regulariser)
```

```
Training: LOSS: 1.9995747670835378 | ACCURACY: 0.27994032434402333
Testing: LOSS: 1.952812922000885 | ACCURACY: 0.2912109375


Model is using cuda
------EPOCH 2/10------
Training: LOSS: 1.9201509538961916 | ACCURACY: 0.3136844023323615
Testing: LOSS: 1.8711986392736435 | ACCURACY: 0.33046875


Model is using cuda
------EPOCH 3/10------
Training: LOSS: 1.8336867094039917 | ACCURACY: 0.34865843658892126
Testing: LOSS: 1.801555296778679 | ACCURACY: 0.3626953125


Model is using cuda
```

Baseline Method- In the training and testing graph for Baseline model we can clearly see that the model overfits. This is because the model has no regularization and also only 50% of the training data is considered.



Exercise 1: Normalization Effect (CNN)

Here different transforms are applied on the data to improve data. And then model performance is analysed.

Data Augmentation – Here 5 transformations are performed on data

First data augmentation is performed.

Exercise 1: Normalization Effect (CNN)

1. Data Augmentation: the process of artificially 'increasing' our dataset by adding translation, scaling and flipping to the images to fabricate examples for training.
2. Normalization: Normalizing the input data helps remove the dataset artifacts that can cause poor model performance.

```
#Training Network with only Data Augmentations
train_augmentations1 = transforms.Compose([transforms.RandomGrayscale(0.2),
                                            transforms.RandomHorizontalFlip(0.5),
                                            transforms.RandomVerticalFlip(0.2),
                                            transforms.RandomRotation(30),
                                            transforms.RandomAdjustSharpness(0.4),
                                            transforms.ToTensor(),
                                           ])

test_augmentations1 = transforms.Compose([transforms.ToTensor()]
                                          )
```

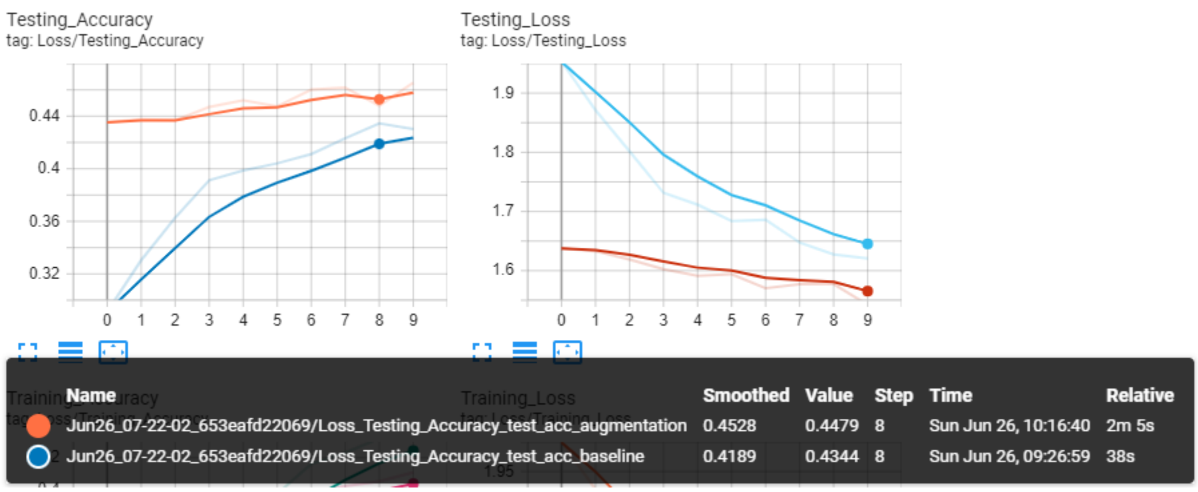
Here the final test accuracy is 46% which is an improvement from baseline model.

```
[20] train_dataloader_with_aug = torch.utils.data.DataLoader(train_ds1, batch_size=256, shuffle=True, num_workers=2)
     test_dataloader_with_aug = torch.utils.data.DataLoader(test_ds1, batch_size=256, shuffle=True, num_workers=2)
```

```
trainer_aug = Trainer(model, (train_dataloader_with_aug, test_dataloader_with_aug), device)

(train_losses, train_accs), (test_losses, test_accs) = trainer_aug.fit(regulariser)
```

```
Training: LOSS: 1.8310919641232004 | ACCURACY: 0.35395806760204085
Testing: LOSS: 1.6377997934818267 | ACCURACY: 0.4353515625


Model is using cuda
------EPOCH 2/10------
Training: LOSS: 1.8014435062603074 | ACCURACY: 0.36695631377551025
Testing: LOSS: 1.6327746242284775 | ACCURACY: 0.43759765625


Model is using cuda
------EPOCH 3/10------
Training: LOSS: 1.78185010078002 | ACCURACY: 0.3734135841836735
Testing: LOSS: 1.6189169466495514 | ACCURACY: 0.43681640625
```

CIFAR-10 Augmentation – The graph represents the data with 5 transforms applied to it. All the training samples are considered. The testing accuracy increases as compared from the baseline from 43% to 46%. The testing loss decreases from 1.62 to 1.54. Thus only augmentation helps achieve better performance.



Normalization- Normalizing the input data helps remove the dataset artifacts that can cause poor model performance.

```
[24] #Training Network with Only Normalization
     train_augmentations2 = transforms.Compose([transforms.ToTensor(),
                                             transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
                                            ])

     test_augmentations2 = transforms.Compose([transforms.ToTensor(),
                                            transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))]
                                          )
```

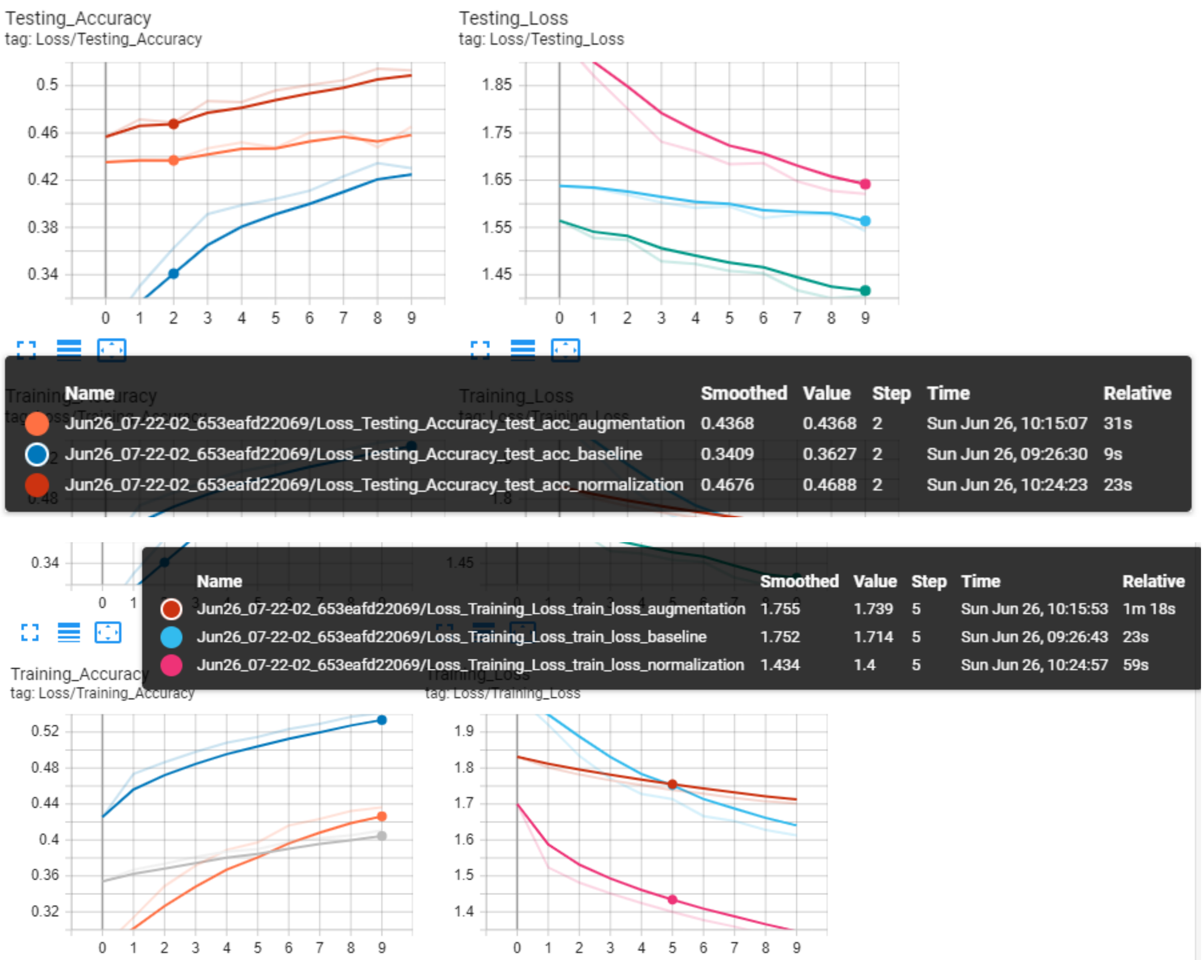
Here the final testing accuracy is 51% and the test loss is 1.4. Training accuracy is 54% and training loss is 1.32.

```
[26] train_dataloader_with_normalization = torch.utils.data.DataLoader(train_ds2, batch_size=256, shuffle=True, num_workers=2)
     test_dataloader_with_normalization = torch.utils.data.DataLoader(test_ds2, batch_size=256, shuffle=True, num_workers=2)
```

```
# model_norm = Custom_Network().to(device)
trainer_with_normalization= Trainer(model, (train_dataloader_with_normalization, test_dataloader_with_normalization), device)

(train_losses, train_accs), (test_losses, test_accs) = trainer_with_normalization.fit(regulariser)
```

```
Model is using cuda
------EPOCH 1/10------
Training: LOSS: 1.6999912109910225 | ACCURACY: 0.425534119897959 14
Testing: LOSS: 1.5642434805631638 | ACCURACY: 0.456640625


Model is using cuda
------EPOCH 2/10------
Training: LOSS: 1.5235653726422056 | ACCURACY: 0.47338568239795914
Testing: LOSS: 1.5277052343225478 | ACCURACY: 0.471484375
```

CIFAR-10 with Normalization – After performing normalization we can see that the testing accuracy increases to nearly 52% as compared to baseline. Also, only normalization is done. Data augmentation is not carried out. In the training graph we see, data normalization gives the better performance as compared to baseline. The testing loss decreases from 1.6 to 1.4 as compared to baseline.



Both Augmentation and Normalization- Here both transforms are carried out on data.

Both Augmentation and Normalization

```
train_augmentations_all = transforms.Compose([transforms.RandomGrayscale(0.2),
                                              transforms.RandomHorizontalFlip(0.5),
                                              transforms.RandomVerticalFlip(0.2),
                                              transforms.RandomRotation(30),
                                              transforms.RandomAdjustSharpness(0.4),
                                              transforms.ToTensor(),
                                              transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
                                             ])

test_augmentations_all = transforms.Compose([transforms.ToTensor(),
                                             transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))]
                                           )
```

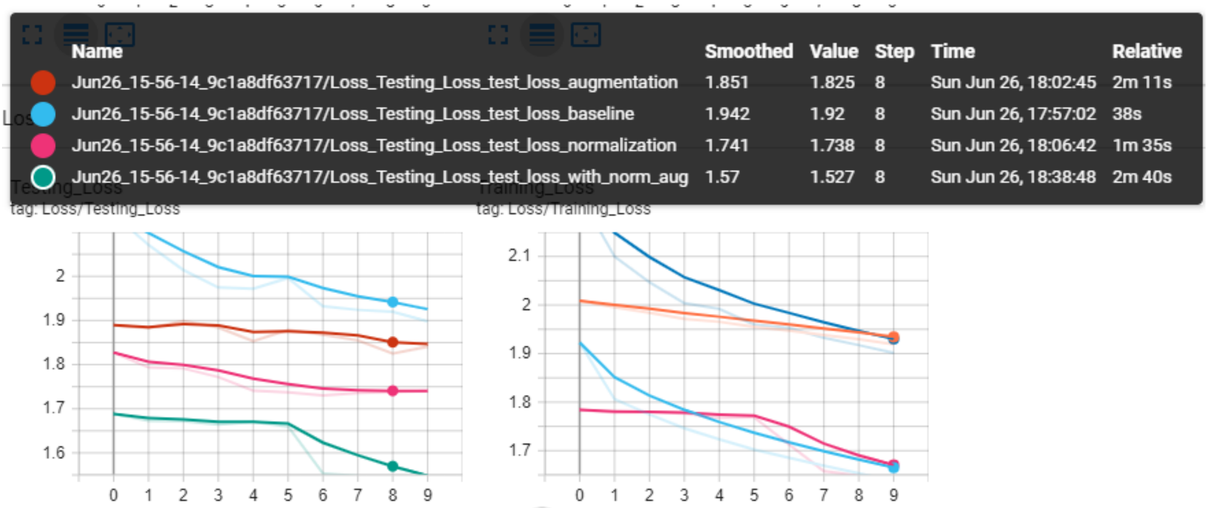
```
path = '/tmp'

train_ds_all = datasets.CIFAR10(root=path,
                            train=True,
                            transform=train_augmentations_all,
                            download=True
                           )

test_ds_all = datasets.CIFAR10(root=path,
                           train=False,
                           transform=test_augmentations_all
                           )

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

```
Files already downloaded and verified
```

```
[11] train_dataloader_all = torch.utils.data.DataLoader(train_ds_all, batch_size=256, shuffle=True, num_workers=2)
     test_dataloader_all = torch.utils.data.DataLoader(test_ds_all, batch_size=256, shuffle=True, num_workers=2)
```

```
# model_norm = Custom_Network().to(device)
trainer_with_all= Trainer(model, (train_dataloader_all, test_dataloader_all), device)
regulariser="None"
(train_losses, train_accs), (test_losses, test_accs) = trainer_with_all.fit(regulariser)
```

Graph results- The accuracy with normalization and augmentation is the highest as compared to baseline. Testing loss is the lowest for with_aug_norm as compared to the baseline at 1.5. This is true even for training. Thus, we can conclude augmentation along with normalization gives the best performance.



Exercise 2: Network Regularization (CNN)

Modifications are made to the base network. Here two dropout layers are added with a probability of 25%. Both these layers are added in the fully connected layers. These ensures that the network doesn't overfit while training. To ensure dropout is carried during training and not during testing self.model.train() and self.model.eval() functions are added during training. Final train accuracy is 41% and test accuracy is 50%. The model performs better on testing data.

```
class Net_dropout(nn.Module):
    def __init__(self, in_features=3, num_classes=10):
        super(Net_dropout, self).__init__()
        self.conv_block = nn.Sequential( nn.Conv2d(in_channels=in_features,out_channels=6,kernel_size=3,stride=1),
                                         nn.ReLU(),
                                         nn.MaxPool2d(2,2),
                                         nn.Conv2d(in_channels=6,out_channels=12,kernel_size=3,stride=1),
                                         nn.ReLU(),
                                         nn.Conv2d(in_channels=12,out_channels=16,kernel_size=3,stride=1),
                                         nn.ReLU(),
                                         nn.MaxPool2d(2,2)
                                        )
        self.linear_block = nn.Sequential( nn.Linear(16*5*5, 120),
                                           nn.ReLU(),
                                           nn.Dropout(0.25),
                                           nn.Linear(120,84),
                                           nn.ReLU(),
                                           nn.Dropout(0.25),
                                           nn.Linear(84,10),
                                           nn.ReLU()
                                         )

    def forward(self, x):
        x = self.conv_block(x)
        x = torch.flatten(x,1)
        x = self.linear_block(x)
        return x
```

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

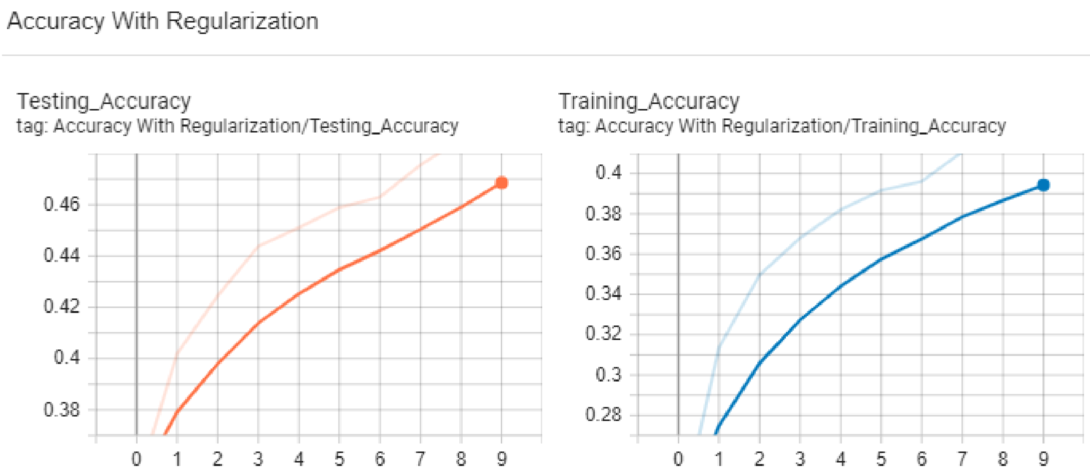
```
Net_dropout(
  (conv_block): Sequential(
    (0): Conv2d(3, 6, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(6, 12, kernel_size=(3, 3), stride=(1, 1))
    (4): ReLU()
    (5): Conv2d(12, 16, kernel_size=(3, 3), stride=(1, 1))
    (6): ReLU()
    (7): 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): ReLU()
    (2): Dropout(p=0.25, inplace=False)
    (3): Linear(in_features=120, out_features=84, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.25, inplace=False)
    (6): Linear(in_features=84, out_features=10, bias=True)
    (7): ReLU()
  )
)
```

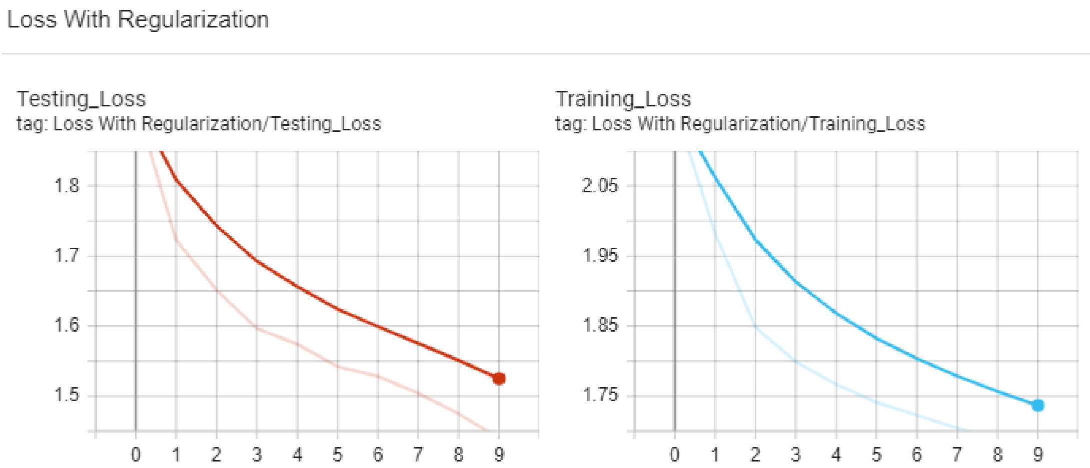
```
trainer = Trainer(model, (train_dataloader_all, test_dataloader_all), device)
regulariser="None"
(train_losses, train_accs), (test_losses, test_accs) = trainer.fit(regulariser)
```

```
Training: LOSS: 2.163170884458386 | ACCURACY: 0.22529496173469388
Testing: LOSS: 1.9181569248437882 | ACCURACY: 0.3501953125


Model is using cuda
------EPOCH 2/10------
Training: LOSS: 1.9826867598660138 | ACCURACY: 0.31368383290816326
Testing: LOSS: 1.7227968364953994 | ACCURACY: 0.40185546875


Model is using cuda
------EPOCH 3/10------
Training: LOSS: 1.8482156140463692 | ACCURACY: 0.34956951530612246
Testing: LOSS: 1.6514089226722717 | ACCURACY: 0.4244140625


Model is using cuda
------EPOCH 4/10------
Training: LOSS: 1.7993173690474764 | ACCURACY: 0.3680843431122449
Testing: LOSS: 1.5964259207248688 | ACCURACY: 0.44375
```

```
Model is using cuda
------EPOCH 6/10------
Training: LOSS: 1.741213980377937 | ACCURACY: 0.3917291135204082
Testing: LOSS: 1.5423934876918792 | ACCURACY: 0.45888671875


Model is using cuda
------EPOCH 7/10------
Training: LOSS: 1.7229671149837726 | ACCURACY: 0.3960658482142857
Testing: LOSS: 1.5285702735185622 | ACCURACY: 0.46298828125


Model is using cuda
------EPOCH 8/10------
Training: LOSS: 1.7043554278052584 | ACCURACY: 0.41077407525510207
Testing: LOSS: 1.5043107450008393 | ACCURACY: 0.47548828125


Model is using cuda
------EPOCH 9/10------
Training: LOSS: 1.687567409203977 | ACCURACY: 0.41250797193877553
Testing: LOSS: 1.4748970985412597 | ACCURACY: 0.48564453125


Model is using cuda
------EPOCH 10/10------
Training: LOSS: 1.6718419619968958 | ACCURACY: 0.4192163584183673
Testing: LOSS: 1.4402622938156129 | ACCURACY: 0.4998046875
```

Dropout- Here the final testing accuracy is 49% which is more compared to baseline model 42%. Due to regularisation the training accuracy is not much affected, but the overfitting problem is resolved. The training accuracy after dropout is 42%. The training loss is 1.44 as compared to baseline training loss which was 1.62.





Bonus Question

Only training loss computation function changes for L1-Reg. Here Custom_Network() is used without dropout. Here, we are calculating a sum of the absolute values of all of the weights. We sum up all the weights and we multiply them by a value called lambda_L1 which is you have to tell it how big of an effect you want the L1 to have lambda. It tries to shrink the error as much as possible if you're adding the sum of the weights onto that error it's going to shrink those weights because that's just an additive property of the weights so it tries to shrink the weights down.

$$\boxed{Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|}$$

Fig: L1 Loss Source: https://androidkt.com/how-to-add-l1-l2-regularization-in-pytorch-loss-function/

```python
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)
    if regulariser=='L1': ## this is for L1 Reg
      reg_loss = sum(p.abs().sum()for p in model.parameters())
      loss +=lambda_l1*reg_loss
    elif regulariser=='L2': ## this is for L2 Reg
      reg_loss = sum(p.pow(2.0).sum()for p in model.parameters())
      loss +=lambda_l2*reg_loss
    else:
      loss = self.loss_fn(output, y)
    loss.backward()
    self.optim.step()
    running_loss += loss.item()
    running_acc += self.accuracy(output,y)
```

```
[39] # For L1 Regularisation
     trainer = Trainer(model, (train_dataloader_all, test_dataloader_all), device)
     regulariser='L1'
     (train_losses, train_accs), (test_losses, test_accs) = trainer.fit(regulariser)
```

```
Training: LOSS: 3.162023984656042 | ACCURACY: 0.41946348852040816
Testing: LOSS: 1.4132280141115188 | ACCURACY: 0.489453125


Model is using cuda
------EPOCH 2/10------
Training: LOSS: 2.365024064268385 | ACCURACY: 0.40707110969387755
Testing: LOSS: 1.444857546687126 | ACCURACY: 0.48564453125


Model is using cuda
------EPOCH 3/10------
Training: LOSS: 2.243377219657509 | ACCURACY: 0.39493781887755103
Testing: LOSS: 1.4606525719165802 | ACCURACY: 0.46953125


Model is using cuda
------EPOCH 4/10------
Training: LOSS: 2.1877212767698326 | ACCURACY: 0.3940808354591837
Testing: LOSS: 1.4607624381780624 | ACCURACY: 0.47216796875


Model is using cuda
```

L1- Regularization- According to the graph for training and testing, L1 Regularization does not give high accuracy or lower losses. This is not that beneficial compared to dropout. The testing loss with dropout is lower than testing loss with L1. Hence for this network parameters dropout gives better performance. Final accuracy after L1 Reg is 48% on test set. L2 regularization is the sum of squares of all weights in the model.

L2- Regularization- L2 performs better than L1. For loss computation dropout still gives the lowest training loss and compared to L1 which means it gives best results avoiding overfitting. L2 gives the best performance over testing loss. Final accuracy after L2 Reg is 55% on test set and 46% on train set which is higher than L1 and dropout. L2 Reg gives the lowest testing loss of 1.2 as compared to L1 and dropout which have 1.4.

$$\boxed{Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2}$$

Fig: L1 Loss Source: https://androidkt.com/how-to-add-l1-l2-regularization-in-pytorch-loss-function/

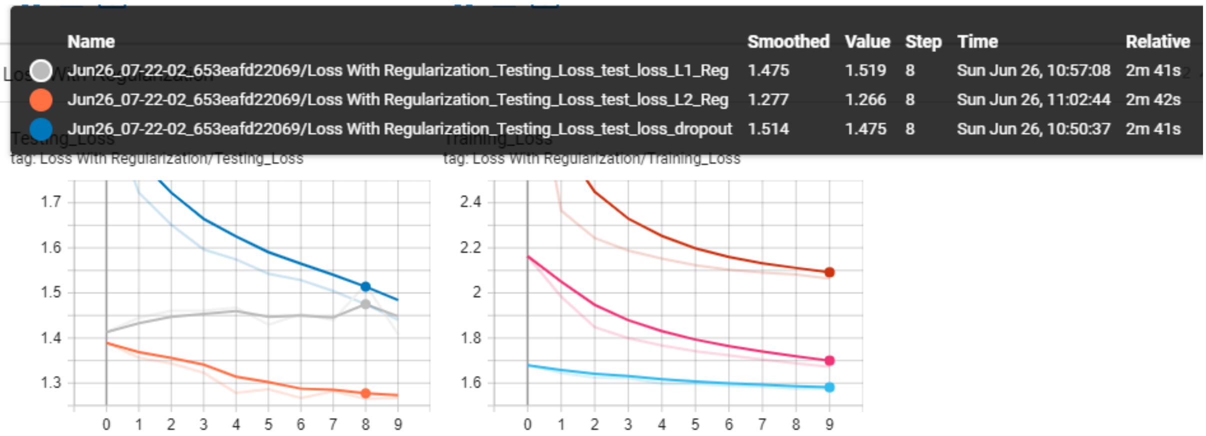
```
for x,y in self.train_loader:
    self.optim.zero_grad()
    x = x.to(self.device, dtype=torch.float)
    y = y.to(self.device, dtype=torch.long)
    output = self.model(x)
    loss = self.loss_fn(output, y)
    if regulariser=='L1': ## this is for L1 Reg
      reg_loss = sum(p.abs().sum()for p in model.parameters())
      loss +=lambda_l1*reg_loss
    elif regulariser=='L2': ## this is for L2 Reg
      reg_loss = sum(p.pow(2.0).sum()for p in model.parameters())
      loss +=lambda_l2*reg_loss
    else:
      loss = self.loss_fn(output, y)
    loss.backward()
    self.optim.step()
    running_loss += loss.item()
    running_acc += self.accuracy(output,y)
```

```
# For L2 Regularisation
trainer = Trainer(model, (train_dataloader_all, test_dataloader_all), device)
regulariser='L2'
(train_losses, train_accs), (test_losses, test_accs) = trainer.fit(regulariser)
```

```
Model is using cuda
------EPOCH 1/10------
Training: LOSS: 1.6794119799623684 | ACCURACY: 0.4116828762755102
Testing: LOSS: 1.3893619000911712 | ACCURACY: 0.48701171875


Model is using cuda
------EPOCH 2/10------
Training: LOSS: 1.6447452768987538 | ACCURACY: 0.4275749362244898
Testing: LOSS: 1.3564866095781327 | ACCURACY: 0.50654296875


Model is using cuda
------EPOCH 3/10------
Training: LOSS: 1.624359213576025 | ACCURACY: 0.4353037308673469
Testing: LOSS: 1.3439359962940216 | ACCURACY: 0.51005859375


Model is using cuda
------EPOCH 4/10------
Training: LOSS: 1.6210942615051658 | ACCURACY: 0.4424545599489796
Testing: LOSS: 1.3233083873987197 | ACCURACY: 0.5234375
```

Training and Testing Accuracy

Training and Testing Loss with Network Regularization



Exercise 3: Optimizers (CNN)

Testing is carried out with two optimizers and two learning rates. All configurations are checked.

```
learning_rate_array=[0.001,0.00001]
optimiser_arr=["Adam","SGD"]
```

Here the training function is modified to take different parameters. Any other changes to network are not done.

```
class Trainer_op:
    def __init__(self, model, dataloaders, device,learning_rate,optimiser):

        self.config = {
            'lr':learning_rate,
            'epochs': 10
        }

        self.model = model
        self.train_loader, self.test_loader = dataloaders
        self.loss_fn = nn.CrossEntropyLoss()
        if optimiser=='Adam':
          self.optim = torch.optim.Adam(self.model.parameters(), lr = self.config['lr'])
        else:
          self.optim = torch.optim.SGD(self.model.parameters(), lr = self.config['lr'])
        self.device = device
```

```
trainer = Trainer_op(model, (train_dataloader_all, test_dataloader_all), device,learning_rate_array[0],optimiser_arr[0])
(train_losses, train_accs), (test_losses, test_accs) = trainer.fit(regulariser,learning_rate_array[0],optimiser_arr[0])
```

```
Model is using cuda
------EPOCH for Adam Optimiser with Learning Rate 0.001 1/10------
Training: LOSS: 1.568271853485886 | ACCURACY: 0.4672632334183674
Testing: LOSS: 1.2520844906568527 | ACCURACY: 0.55107421875


Model is using cuda
------EPOCH for Adam Optimiser with Learning Rate 0.001 2/10------
Training: LOSS: 1.5593552267064854 | ACCURACY: 0.47099808673469384
Testing: LOSS: 1.2394691467285157 | ACCURACY: 0.5546875
```

```
trainer = Trainer_op(model, (train_dataloader_all, test_dataloader_all), device,learning_rate_array[0],optimiser_arr[1])
(train_losses, train_accs), (test_losses, test_accs) = trainer.fit(regulariser,learning_rate_array[0],optimiser_arr[1])
```

```
Model is using cuda
------EPOCH for SGD Optimiser with Learning Rate 0.001 1/10------
Training: LOSS: 1.5004704764911108 | ACCURACY: 0.4995854591836735
Testing: LOSS: 1.1690899759531022 | ACCURACY: 0.57939453125


Model is using cuda
------EPOCH for SGD Optimiser with Learning Rate 0.001 2/10------
Training: LOSS: 1.49948372707075 | ACCURACY: 0.4980508609693878
Testing: LOSS: 1.1637003898620606 | ACCURACY: 0.58486328125


Model is using cuda
------EPOCH for SGD Optimiser with Learning Rate 0.001 3/10------
Training: LOSS: 1.5016942261433115 | ACCURACY: 0.4974968112244898
```

```
trainer = Trainer_op(model, (train_dataloader_all, test_dataloader_all), device,learning_rate_array[1],optimiser_arr[0])
(train_losses, train_accs), (test_losses, test_accs) = trainer.fit(regulariser,learning_rate_array[1],optimiser_arr[0])
```

```
Training: LOSS: 1.5247197607342078 | ACCURACY: 0.48878348214285716
Testing: LOSS: 1.1826829224824906 | ACCURACY: 0.5767578125


Model is using cuda
------EPOCH for Adam Optimiser with Learning Rate 1e-05 2/10------
Training: LOSS: 1.5099038116785946 | ACCURACY: 0.49233896683673467
Testing: LOSS: 1.1768426418304443 | ACCURACY: 0.573046875


Model is using cuda
------EPOCH for Adam Optimiser with Learning Rate 1e-05 3/10------
Training: LOSS: 1.5068278209287294 | ACCURACY: 0.4962691326530612
Testing: LOSS: 1.1732120990753174 | ACCURACY: 0.5775390625
```

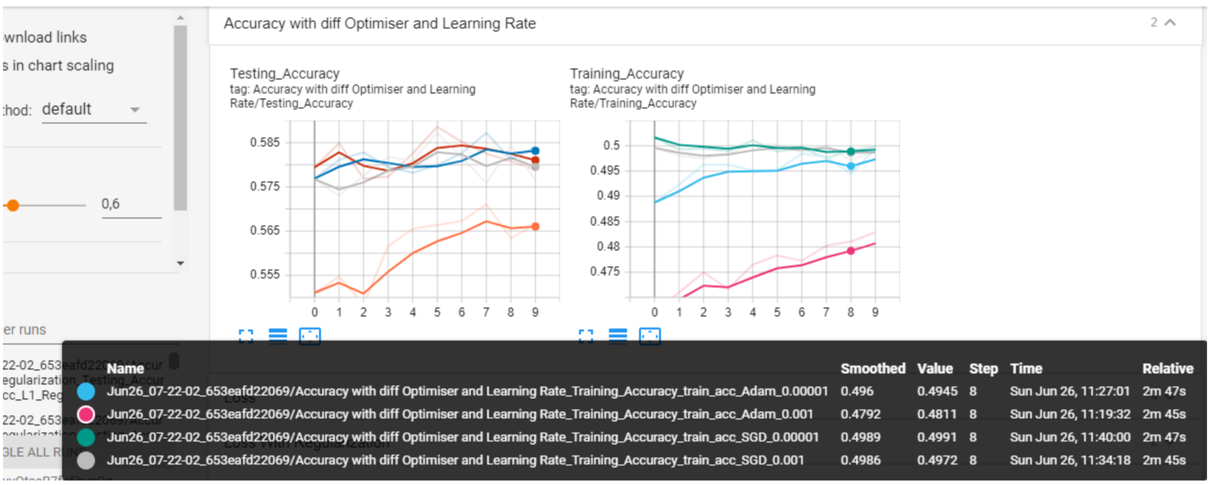
```
trainer = Trainer_op(model, (train_dataloader_all, test_dataloader_all), device,learning_rate_array[1],optimiser_arr[1])
(train_losses, train_accs), (test_losses, test_accs) = trainer.fit(regulariser,learning_rate_array[1],optimiser_arr[1])
```

```
Training: LOSS: 1.4917641634843788 | ACCURACY: 0.5016342474489796
Testing: LOSS: 1.1782564312219619 | ACCURACY: 0.57685546875


Model is using cuda
------EPOCH for SGD Optimiser with Learning Rate 1e-05 2/10------
Training: LOSS: 1.497227593344085 | ACCURACY: 0.4993064413265306
Testing: LOSS: 1.1678504675626755 | ACCURACY: 0.58115234375


Model is using cuda
------EPOCH for SGD Optimiser with Learning Rate 1e-05 3/10------
Training: LOSS: 1.4960670471191406 | ACCURACY: 0.4994419642857143
Testing: LOSS: 1.165156352519989 | ACCURACY: 0.5828125


Model is using cuda
------EPOCH for SGD Optimiser with Learning Rate 1e-05 4/10------
Training: LOSS: 1.4958208799362183 | ACCURACY: 0.4989078443877551
Testing: LOSS: 1.1712672680616378 | ACCURACY: 0.5794921875


Model is using cuda
------EPOCH for SGD Optimiser with Learning Rate 1e-05 5/10------
Training: LOSS: 1.4966949340032072 | ACCURACY: 0.5010841836734694
Testing: LOSS: 1.1694296061992646 | ACCURACY: 0.57822265625
```

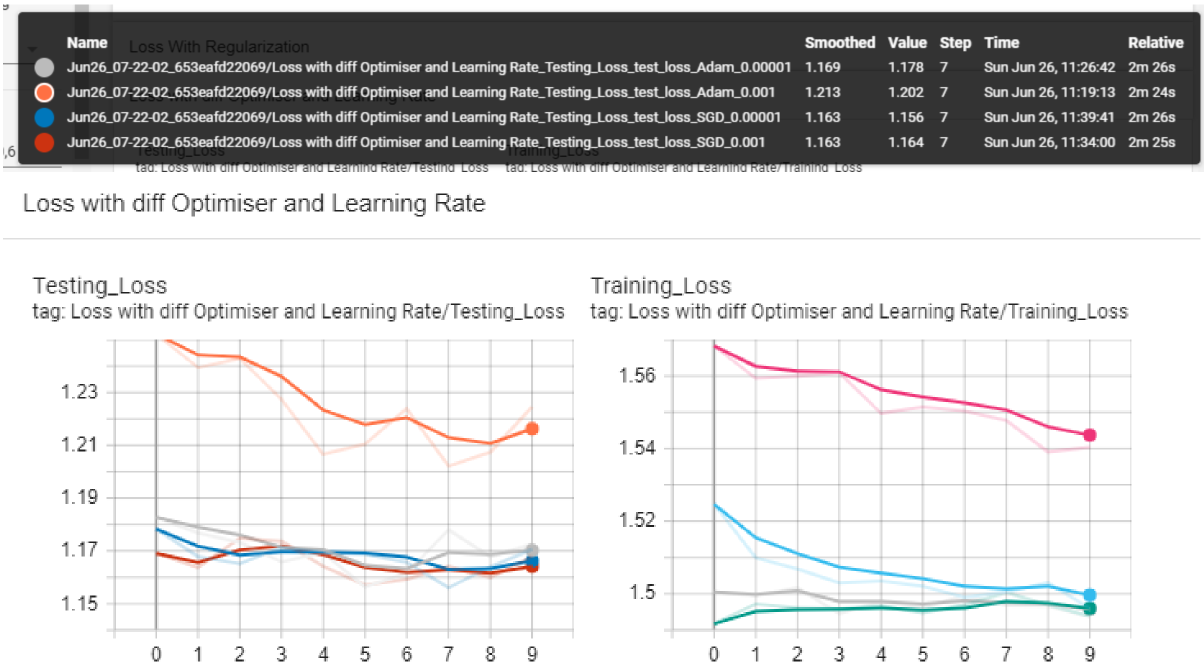
Graph for Different configurations:

Here after looking at the graph, we can conclude that SGD is more robust to learning rates. As the learning rate changes from 0.00001 to 0.001 testing accuracy drops from 58% to 56%. But in the case of SGD change in accuracy is from 58.3 to 58.1. For loss, for Adam Optimiser changing the learning rate affects the loss. When LR changes from 0.00001 to 0.001 the loss increases from 1.17 to 2.44. Whereas, in the case of SGD the change is very minor from 1.166 to 1.164. The same is the scenario for training accuracy and training loss. In conclusion for CIFAR-10 dataset, SGD is more robust to changes in learning rate compared to Adam Optimiser.

Accuracy vs Epoch Graph



Loss vs Epoch Graph



References:

[1] (Knowledge Transfer, 2021) https://androidkt.com/how-to-add-l1-l2-regularization-in-pytorch-loss-function/

[2] https://forums.pytorchlightning.ai/t/multiple-scalars-e-g-train-and-valid-loss-in-same-tensorboard-graph/751