Lab Course: Distributed Data Analytics

Exercise Sheet 7

In [31]:
```
import numpy as np
import pandas as pd
import torch
from torch.utils.data import DataLoader, Dataset
import torchvision
import matplotlib.pyplot as plt
from torchvision import datasets, transforms
from torch import nn, optim
import torch.nn.functional as F
from torch.utils.tensorboard import SummaryWriter
import torch.utils.data as data_utils
```

In [32]:
```
def fiftyprcnt(train_dataset):
    indices = torch.arange(int(0.5*len(train_dataset)))
    train_50pcnt = data_utils.Subset(train_dataset, indices)
    return train_50pcnt

def dim_calc(width,kernel_size):
    w,k,p,s=width,kernel_size,0,1
    conv_op=(w-k+2*p)/s+1
    w=conv_op/2
    conv_op=(w-k+2*p)/s+1
    conv_op=(conv_op-k+2*p)/s+1
    w=conv_op/2
    return int(w)
```

Network Analysis: Image Classification

Approach: 1) **Model creation**: Created the model "base" as per specifications gvien in the excercise. The output of the network is without applying softmax as the cross entropy loss funtion already contains it.otmax is applied while calulating accuracy. Kenel size is chosen as 3. THe number of input and output channels in convolution layers and number of neurons are chosen randomly keeping in mind of complexity of the model. The number o neurons after flattening of the dimension of the feature caluted using a predefined funtion named "dim_calc". This function uses predefined formulas to calculate the width of the output features at each convolution layer and returns the width of the feature output of pool2 layer.

2) To use only **50 percent of the training dataset**,"data_utils.Subset(train_dataset, indices)" is used with "indices" as a list of numbers in the range of (50 percent the total length of the actuual train data.

3) Baseline image classification without any data augmentation or normalization is performed along with other configurations below

In [3]:
```
class base(nn.Module):
    def __init__(self,in_ch,width,kernel):
        super(base, self).__init__()

        self.conv1 = nn.Conv2d(in_ch, out_channels=32, kernel_size=kernel)
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(32,64,kernel)
        self.conv3 = nn.Conv2d(64,128,kernel)
        self.pool2 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(128*dim_calc(width,kernel)**2,100)
        self.fc2 = nn.Linear(100,50)
        self.fc3 = nn.Linear(50, 10)
        self.relu=nn.ReLU()

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.pool1(x)
        x = self.relu(self.conv2(x))
        x = self.relu(self.conv3(x))
        x = self.pool2(x)
        x = x.view(x.size(0),-1)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.relu(self.fc3(x))
        return x
```

Exercise 1: Normalization Effect (CNN)

Appraoch for data loading

1)**Data Augmentation**: the images are flipped using "RandomHorizontalFlip" & "RandomVerticalFlip",translated and scaled and translated using "RandomAffine(degrees=0,translate=(0.1, 0.3)".

2)**Normalization**: Each channel of the image is normalized by substracting the mean (μ) of each feature and a division by the standard deviation (σ). Referring to https://towardsdatascience.com/how-to-calculate-the-mean-and-standard-deviation-normalizing-datasets-in-pytorch-704bd7d05f4c on 21st June,2022.

3) All combinations of configurations are predefined for the training data using transforms.Compose(). For the test data, only normalization is added for the configuration "with normalization" and " wiht augmentation and normalization". For the configuration "with baseline" and "with augmentatiions", only basic transformations to tensor is used. To load these configurations when required, a function named "data-laoder" is created.

In [4]:
```
#https://www.programcreek.com/python/example/117699/torchvision.transforms.RandomAffine
augmetnations = transforms.Compose([transforms.RandomHorizontalFlip(p=0.5),transforms.RandomVerticalFlip(0.2),
                              transforms.RandomAffine(degrees=0,translate=(0.1, 0.3),scale=(1.1,1.2)),
                              transforms.ToTensor()])

augmentations_with_norm=transforms.Compose([
                              transforms.RandomHorizontalFlip(p=0.5),
                              transforms.RandomVerticalFlip(0.2),
                              transforms.RandomAffine(degrees=0,translate=(0.1, 0.3),scale=(1.1,1.2)),
                              transforms.ToTensor(),
                              transforms.Normalize(mean=[0.4914, 0.4822, 0.4465],std=[0.247, 0.243, 0.261])])

norms=transforms.Compose([transforms.ToTensor(),
                                            transforms.Normalize(mean=[0.4914, 0.4822, 0.4465],
                                                             std=[0.247, 0.243, 0.261])])



basic_transforms=transforms.Compose([transforms.ToTensor()])

test_trfms_with_norm=transforms.Compose([transforms.ToTensor(),transforms.Normalize(mean=[0.4914, 0.4822, 0.446
                                                             std=[0.247, 0.243, 0.261])])
```

In [5]:
```
config_l=["baseline","with augmetnations","with noramlization","augmentations_with_noramlization"]
def data_loader(conig):
    if config==config_l[0]:
        train_50pcnt = fiftyprcnt(datasets.CIFAR10(root='data', train=True,download=True, transform=basic_trans
        test_dataset = datasets.CIFAR10(root='data', train=False,download=True, transform=basic_transforms)
    elif config==config_l[1]:
        train_50pcnt = fiftyprcnt(datasets.CIFAR10(root='data', train=True,download=True, transform=augmetnatio
        test_dataset = datasets.CIFAR10(root='data', train=False,download=True, transform=basic_transforms)
    elif config==config_l[2]:
        train_50pcnt = fiftyprcnt(datasets.CIFAR10(root='data', train=True,download=True, transform=norms))
        test_dataset = datasets.CIFAR10(root='data', train=False,download=True, transform=test_trfms_with_norm)
    else:
        train_50pcnt = fiftyprcnt(datasets.CIFAR10(root='data', train=True,download=True, transform=augmentatio
        test_dataset = datasets.CIFAR10(root='data', train=False,download=True, transform=test_trfms_with_norm)
    return train_50pcnt,test_dataset
```

Learning with different configurations. 1) In learning, iterating through list of configrations **including baseline**, 50 percent train data and full test data are loaded and "torch.utils.data.DataLoader" is used to load the data in minibatches.

2) **Batch size** is chosen as **128**. **Adam optimizer** with learning rate **0.001** is chosen. The "cross entropy loss" is used to back propogate on to update the weights.

3) **Softmax** is applied on output of the model and is compared with actaul labels to get accuracies. Iterating through mininbatches of train and test data, tarin and test lossses and accuracies are taken at each epoch and written to tensorboard.

In [6]:
```
def learning(config,optim,lr,kernel):
    train_dataset,test_dataset=data_loader(config)
    trainloader_CIF = torch.utils.data.DataLoader(train_dataset, batch_size=128, shuffle=True,num_workers = 2)
    testloader_CIF = torch.utils.data.DataLoader(test_dataset, batch_size=128, shuffle=True)
    width,in_ch=32,3
    model = base(in_ch,width,kernel)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()
    train_loss_epoch_l,test_loss_epoch_l=[],[]
    writer = SummaryWriter(f"conf/coniguration={config}")
    print(f"for coniguration={config}")
    for j in range(40):
        # training
        model.train()
        train_loss_h,train_pred_l,test_pred_l,test_label,train_label=0,[],[],[],[]
        for images, labels in trainloader_CIF:
            optimizer.zero_grad()
            y_hat_train = model(images)
            prob=F.softmax(y_hat_train, dim=1)
            pred=[torch.argmax(j) for j in prob]
            train_loss = criterion(y_hat_train, labels)
            train_loss_h+=train_loss.item()*len(images)
            train_pred_l=train_pred_l+pred
            train_label=train_label+list(labels)
            train_loss.backward()
            optimizer.step()
```

```
        train_loss_epoch=np.round((train_loss_h/len(train_dataset)),4)
        train_loss_epoch_l.append(train_loss_epoch)
        train_acc=np.round((np.array(train_pred_l)==np.array(train_label)).mean(),4)
        # testing
        model.eval()
        with torch.no_grad():
            test_loss_h=0
            for images, labels in testloader_CIF:
                y_hat_test = model(images)
                prob=F.softmax(y_hat_test, dim=1)
                pred=[torch.argmax(j) for j in prob]
                test_loss = criterion(y_hat_test, labels)
                test_loss_h+=test_loss.item()*len(images)
                test_pred_l=test_pred_l+pred
                test_label=test_label+list(labels)
        test_acc=np.round((np.array(test_pred_l)==np.array(test_label))).mean()
        test_loss_epoch=np.round((test_loss_h/len(test_dataset)),4)
        test_loss_epoch_l.append(test_loss_epoch)
        writer.add_scalar('Loss_CIFAR10/train', train_loss_epoch, j)
        writer.add_scalar('Loss_CIFAR10/test', test_loss_epoch, j)
        writer.add_scalar('Accuracy_CIFAR10/train', train_acc, j)
        writer.add_scalar('Accuracy_CIFAR10/test', test_acc, j)
        print(f"Epoch {j} - train_loss : {train_loss_epoch},test loss : {test_loss_epoch},train_acc : {train_ac
    print("                                                                                  ")
```

In [ ]:

```
optim,kernel,lr="Adam",3,0.001
torch.manual_seed(4)
for config in config_l:
    learning(config,optim,lr,kernel)
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

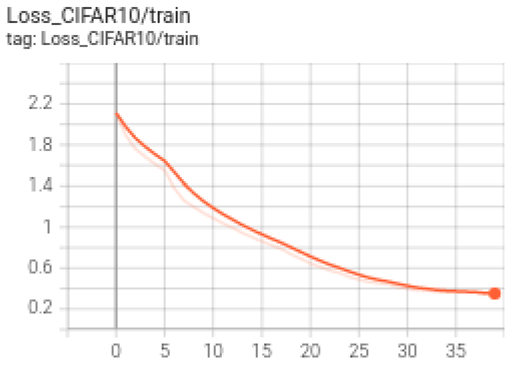
In [ ]:

In [ ]:

In [ ]:

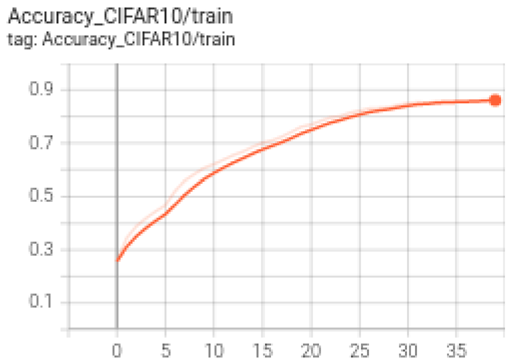
Baseline: (Loss/Accuracies vs Number of epochs) **Analysis**

Train loss

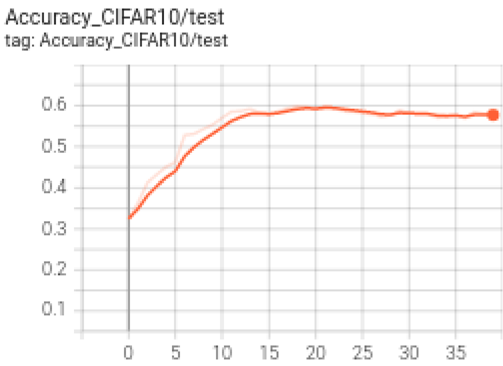


Test Loss

Train Accuracy



Test Accuracy

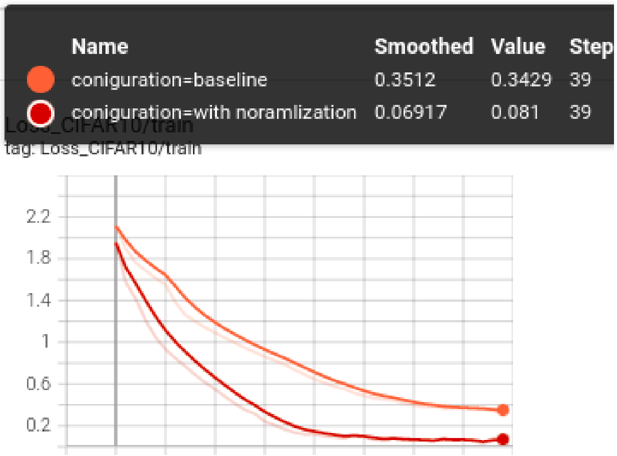


Comment: WIthout any augmentation and normalization and with only 50 percent of training data, the model is getting overfitted at small accuracy level.

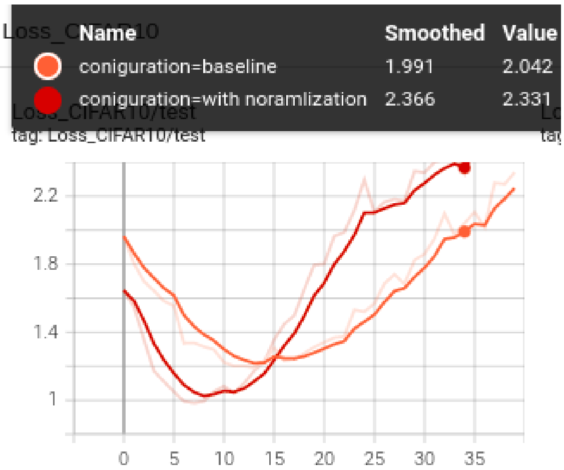
Ex1 Analysis:

Normalization vs **Baseline** (Loss/Accuracies vs Number of epochs)

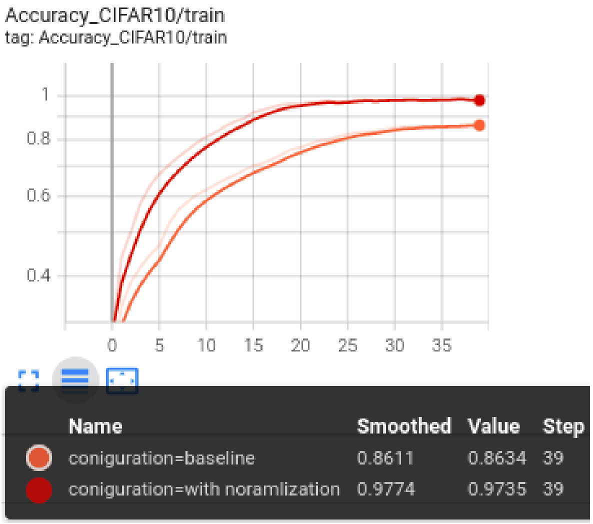
Train loss



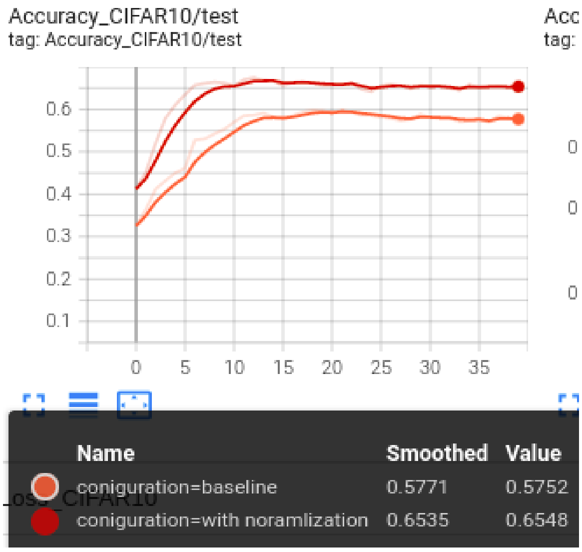
Test Loss



Train Accuracy



Test Accuracy

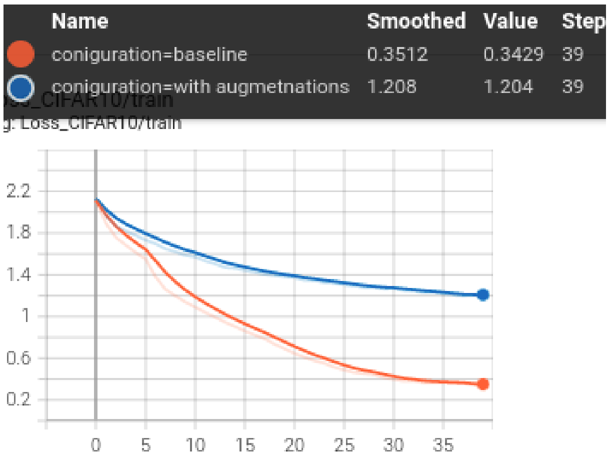


Comment: From the above graphs, it is observed that the train and test losses are lower than that of baseline without any normalization. Accuracy is also better with normalization. But after around 10 and 15 epochs, “normalization” and “baseline” models overfitted respectively indicating the need for regularization. But by the time o overfitting, model with normalization is performing better than that o baseline

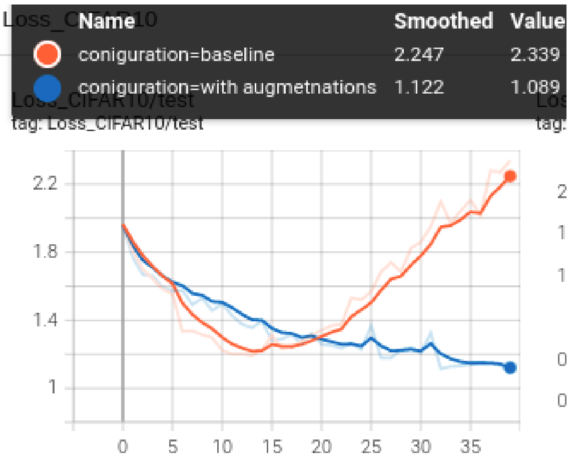
The normalization of the data i.e getting all the data in each channel on the same distribution or same scale is making the model stable by maintaining the contribution of every feature and also making the model learn small weights. This is resulting in reducing the loss and improving the accuracy even before overfitting than that of the baseline model without normalization.

With data augmentations vs **Baseline** (Loss/Accuracies vs Number of epochs)

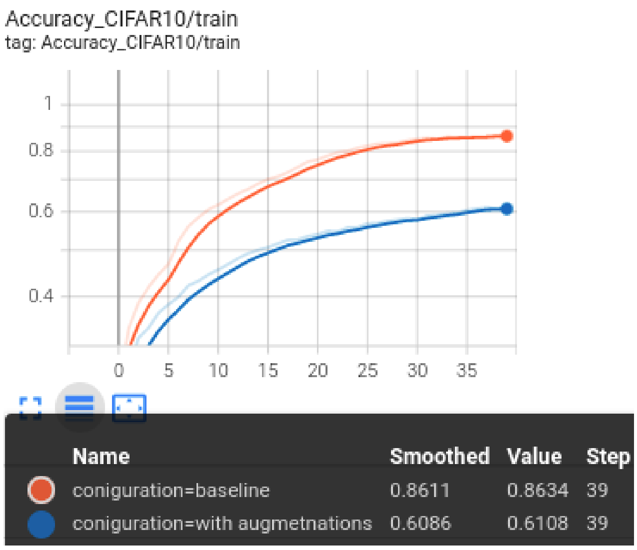
Train loss



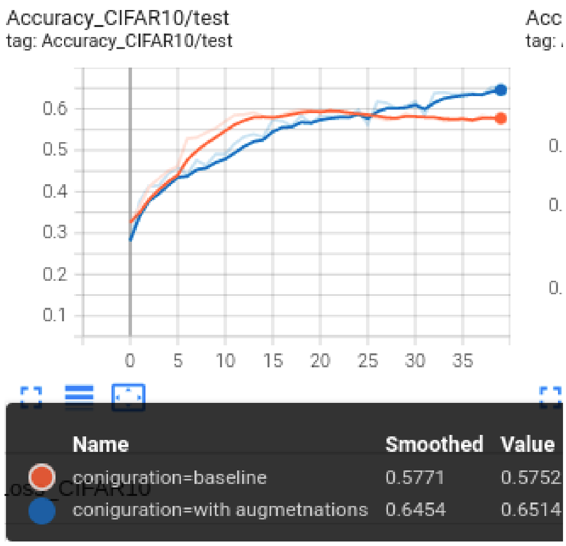
Test Loss



Train Accuracy



Test Accuracy

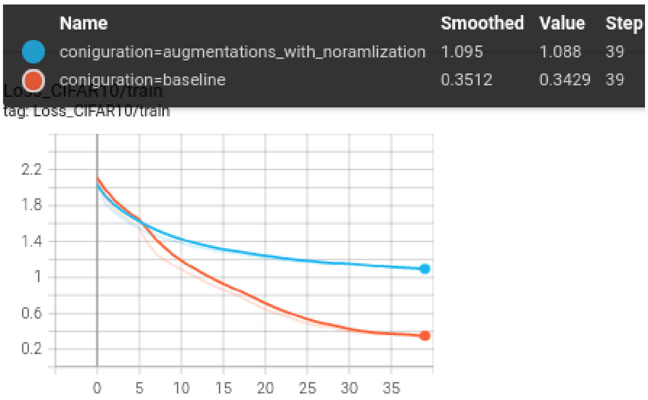


Comment: From the above graphs, it is observed that the train and test losses are lower than that of baseline with data augmentation. The loss curves are taking more iterations for converging than that of baseline. Accuracy is also better with data augmentation. We can see that data augmentation technique is also preventing the model from overfitting. This is because data augmentation increases the size of the data that is increasing the number of images present in the dataset.

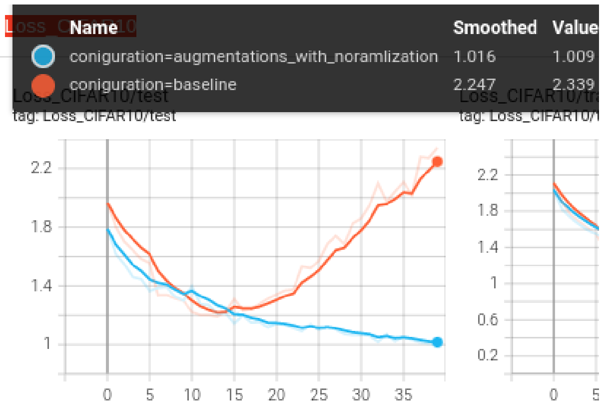
Although we have taken 50 percent of the data, the data augmentation technique provides a lot of variations of each image by rotating, scaling, and translating. This makes the model learn features of the same image class under diverse image transforms. This results in good generalization of the model for new images thereby reducing overfitting. The test curve of the model with data augmentation is not that smooth enough requiring better regularization.

With data augmentations and normalization vs **Baseline** (Loss/Accuracies vs Number of epochs)

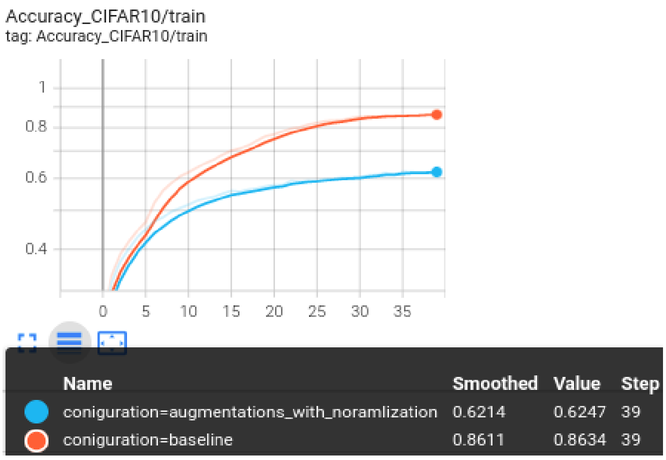
Train loss



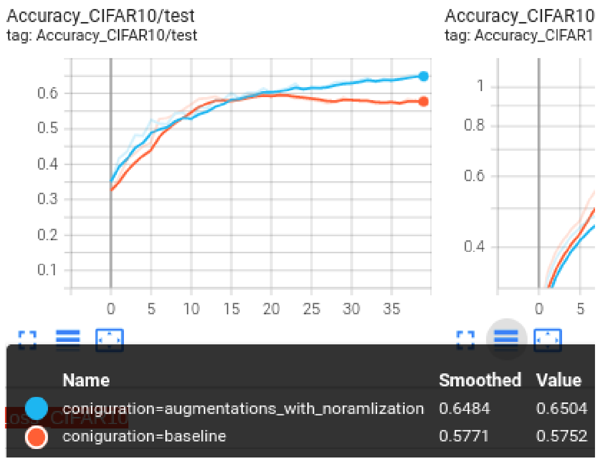
Test Loss



Train Accuracy



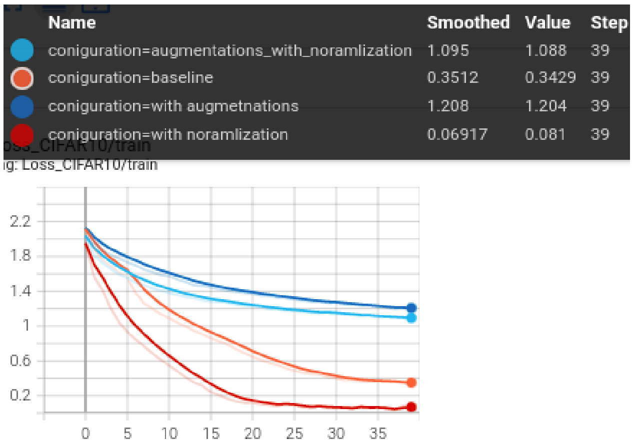
Test Accuracy



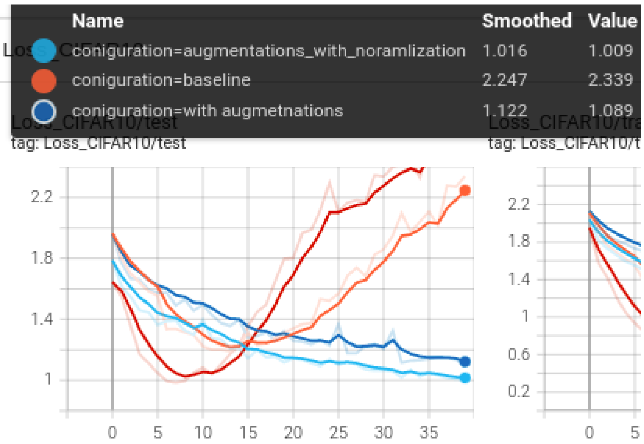
Comment: From the above graphs, it is observed that the train and test losses are lower than that of baseline with data augmentation and normalization. Accuracy is also better with data augmentation and normalization. Here also we can see the regularization of the model by data augmentation. The test curve of the model with data augmentation is not that smooth enough requiring better regularization. Convergence of the loss curve with data augmentations and normalization is still taking more epochs.

Comparing all configurations vs Baseline (Loss/Accuracies vs Number of epochs)

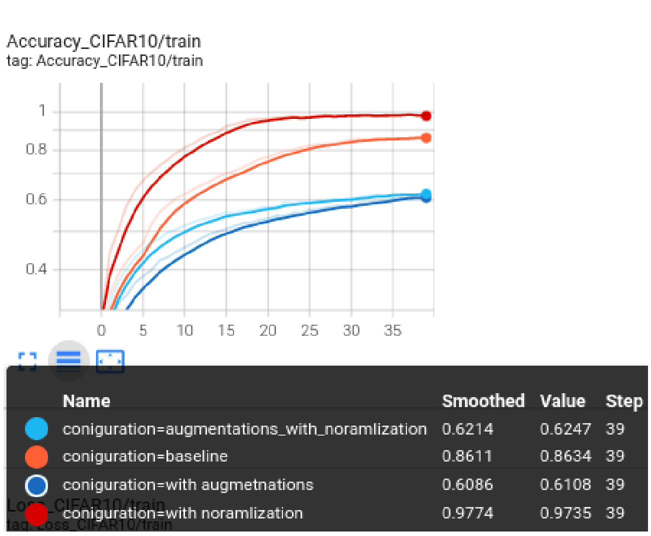
Train loss



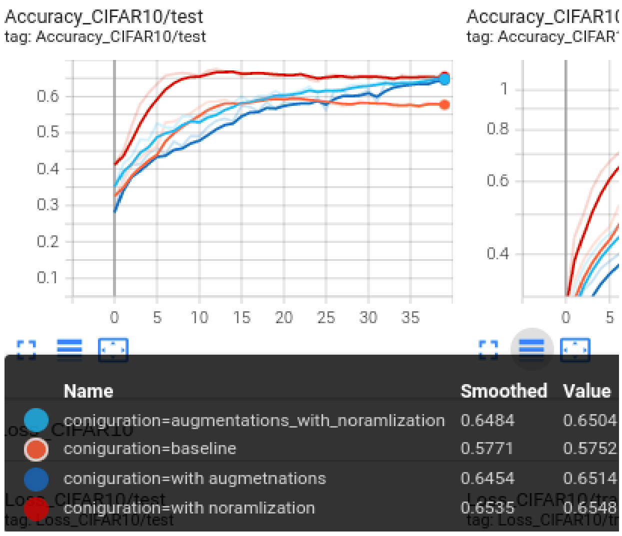
Test Loss



Train Accuracy



Test Accuracy



Comment: When all configurations are compared, the model with data augmentation and normalization is showing the best performance in terms of loss and accuracies reflecting the benefits of normalization and data augmentation as discussed above in each case.

Regarding the convergence of training loss, the model with normalization converges faster than any other. But the model is overfitting. This is rectified by the data augmentation. The regularization effect of data augmentation can be seen in the test curves of the models with “data augmentation” and “data augmentation with normalization”.

The roughness in the test curves shows the requirement for more effective regularization.

Exercise 2: Network Regularization (CNN)

Approach for regularization

Data 50 percent of data is taken with only baseline transformations i.e "toTensor()" as to have comparison with baseline.

Models A new model named "base_drop" is created. The dropout is added to in fully connected network of the original model "base" with p=0.25. Remining network kept same. For L1 and L2 regularization, the original model without dropout i.e "base" is used.

Learning For learning with different regularization techniques, a new function called "learning_regu" is created. This funtion checks for the name of regularization. If it is "dropout", it takes the model "base_drop". Inorder to handle droputs during testing, model.train() and model.eval() is used before testing and training. If it is l1 or l2, it takes original model "base". Then while calculating the losses,

for **l1 regularization**, loss is calcualted as below,

lamda=0.0001
l1_abs = sum(p.abs().sum() for p in model.parameters())
train_loss = train_loss + lamda * l1_abs

for **l2 regularization**, loss is calcualted as below

lamda=0.001
l2_norm = sum(p.pow(2.0).sum() for p in model.parameters())
train_loss = train_loss + lamda * l2_norm

In [8]:
```
#https://arxiv.org/pdf/1207.0580.pdf
# https://wandb.ai/authors/ayusht/reports/Implementing-Dropout-in-PyTorch-With-Example--VmlldzoxNTgwOTE
class base_drop(nn.Module):
    def __init__(self,in_ch,width,kernel):
        super(base_drop, self).__init__()

        self.conv1 = nn.Conv2d(in_ch, out_channels=32, kernel_size=kernel)
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(32,64,kernel)
        self.conv3 = nn.Conv2d(64,128,kernel)
        self.pool2 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(128*dim_calc(width,kernel)**2,100)
        self.fc2 = nn.Linear(100,50)
        self.fc3 = nn.Linear(50, 10)
        self.relu=nn.ReLU()
        self.drop_fc=nn.Dropout(p=0.25)

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.pool1(x)
        x = self.relu(self.conv2(x))
        x = self.relu(self.conv3(x))
        x = self.pool2(x)
        x = x.view(x.size(0),-1)
        x = self.drop_fc(x)
        x = self.relu(self.fc1(x))
        x = self.drop_fc(x)
        x = self.relu(self.fc2(x))
        x = self.drop_fc(x)
        x = self.relu(self.fc3(x))
        return x
```

In [9]:
```
train_dataset = fiftyprcnt(datasets.CIFAR10(root='data', train=True,download=True, transform=basic_transforms))
test_dataset = datasets.CIFAR10(root='data', train=False,download=True, transform=basic_transforms)
trainloader_CIF = torch.utils.data.DataLoader(train_dataset, batch_size=128, shuffle=True,num_workers = 2)
testloader_CIF = torch.utils.data.DataLoader(test_dataset, batch_size=128, shuffle=True)
```

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

In [28]:
```
# https://androidkt.com/how-to-add-l1-l2-regularization-in-pytorch-loss-function/
def learning_regu(reg,optim,lr,kernel,lamda):
    width,in_ch=32,3
    if reg=="dropout":
        model = base_drop(in_ch,width,kernel)
    else:
        model = base(in_ch,width,kernel)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()
    train_loss_epoch_l,test_loss_epoch_l=[],[]
    writer = SummaryWriter(f"conf/regularization={reg}")
    print(f"for regularization={reg}")
    for j in range(40):
        # training
        model.train()
        train_loss_h,train_pred_l,test_pred_l,test_label,train_label=0,[],[],[],[]
        for images, labels in trainloader_CIF:
            optimizer.zero_grad()
```

```
            y_hat_train = model(images)
            prob=F.softmax(y_hat_train, dim=1)
            pred=[torch.argmax(j) for j in prob]
            train_loss = criterion(y_hat_train, labels)
            if reg=="l2":
                lamda=0.001
                l2_norm = sum(p.pow(2.0).sum() for p in model.parameters())
                train_loss = train_loss + lamda * l2_norm
            elif reg=="l1":
                lamda=0.0001
                l1_abs = sum(p.abs().sum() for p in model.parameters())
                train_loss = train_loss + lamda * l1_abs
            train_loss_h+=train_loss.item()*len(images)
            train_pred_l=train_pred_l+pred
            train_label=train_label+list(labels)
            train_loss.backward()
            optimizer.step()
        train_loss_epoch=np.round((train_loss_h/len(train_dataset)),4)
        train_loss_epoch_l.append(train_loss_epoch)
        train_acc=np.round((np.array(train_pred_l)==np.array(train_label)).mean(),4)
        # testing
        model.eval()
        with torch.no_grad():
            test_loss_h=0
            for images, labels in testloader_CIF:
                y_hat_test = model(images)
                prob=F.softmax(y_hat_test, dim=1)
                pred=[torch.argmax(j) for j in prob]
                test_loss = criterion(y_hat_test, labels)
                test_loss_h+=test_loss.item()*len(images)
                test_pred_l=test_pred_l+pred
                test_label=test_label+list(labels)
        test_acc=np.round((np.array(test_pred_l)==np.array(test_label))).mean()
        test_loss_epoch=np.round((test_loss_h/len(test_dataset)),4)
        test_loss_epoch_l.append(test_loss_epoch)
        writer.add_scalar('Loss_CIFAR10/train', train_loss_epoch, j)
        writer.add_scalar('Loss_CIFAR10/test', test_loss_epoch, j)
        writer.add_scalar('Accuracy_CIFAR10/train', train_acc, j)
        writer.add_scalar('Accuracy_CIFAR10/test', test_acc, j)
        print(f"Epoch {j} - train_loss : {train_loss_epoch},test loss : {test_loss_epoch},train_acc : {train_ac
    print("                                                                                         ")
```

In [ ]:
```
regu_l=["l2","l1","dropout"]
optim,kernel,lr,lamda="Adam",3,0.001,0.001
torch.manual_seed(4)
for reg in regu_l:
    learning_regu(reg,optim,lr,kernel,lamda)
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

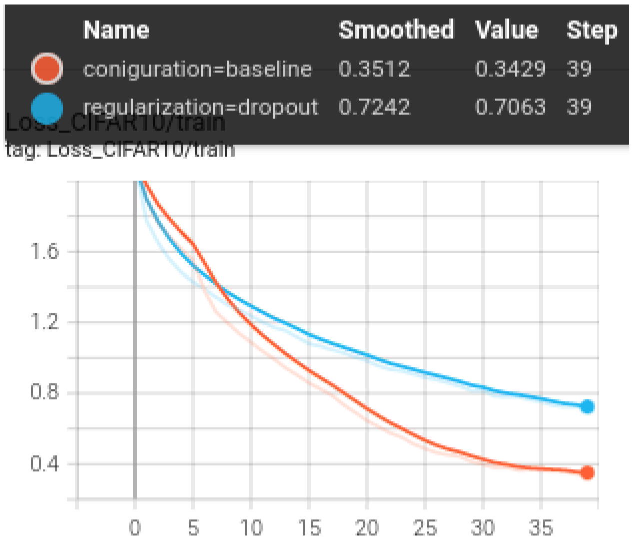
In [ ]:

In [ ]:

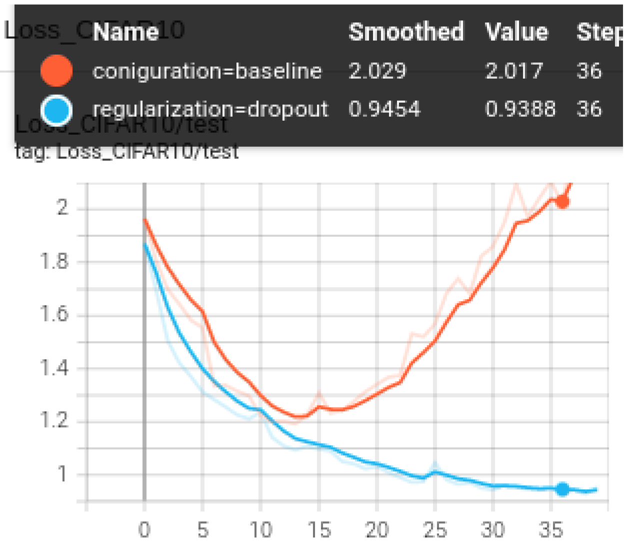
Exercise 2: Network Regularization Analysis

Drop out regularization vs Baseline: (Loss/Accuracies vs Number of epochs)

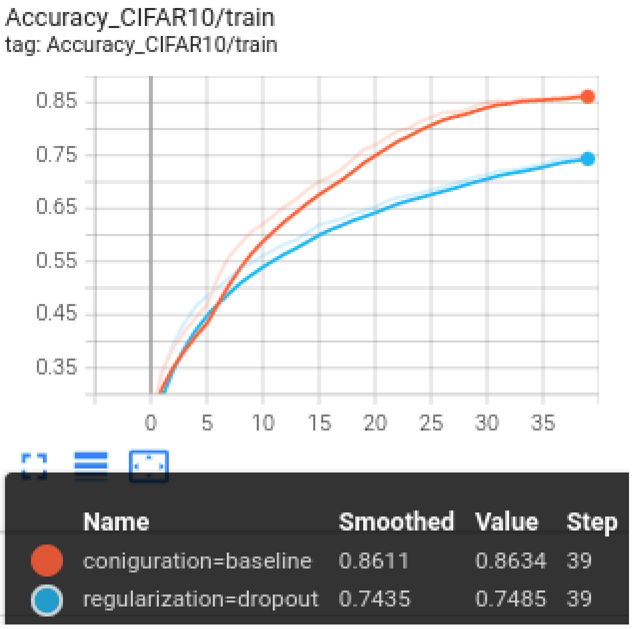
Train loss



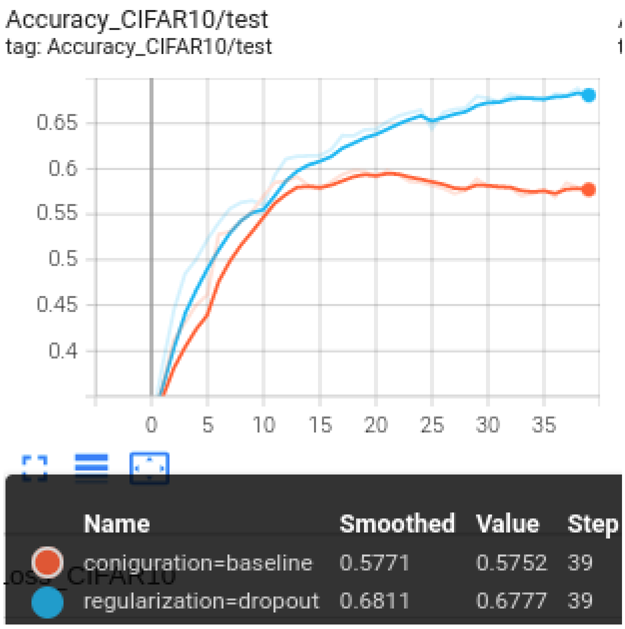
Test Loss



Train Accuracy



Test Accuracy



Comment:

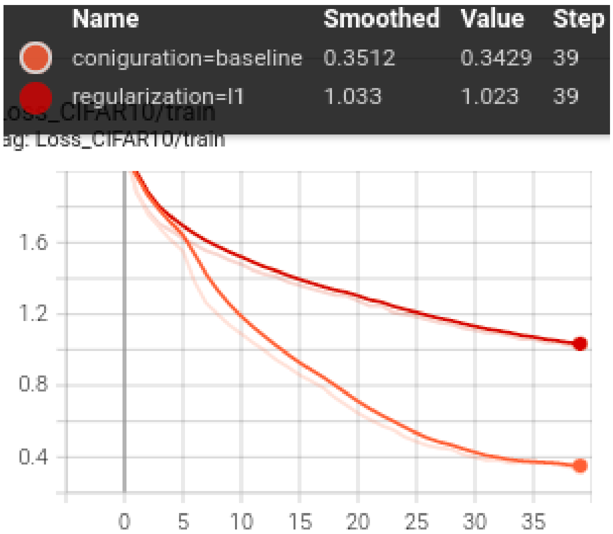
From the above graphs, it is observed that, adding the dropout layer to the baseline model successfully regularized. The test loss curves continuously decreased with very less and small ups and downs. Although the baseline train accuracy is high, the model with “dropout” as a regularizer scored the highest test accuracy and least test loss for the taken number of epochs. The train and test loss curves of the “dropout” model are still moving downwards after 39 epochs. We can expect better test accuracy if iterations are increased. This shows the effectiveness of “dropout” in regularizing the model.

This effectiveness is due to, the random dropout of neurons is making the network less dependent and less sensitive to the specific weights of neurons. With this, the model will learn the more robust features so that the model can easily predict any unseen data leading to good generalization.

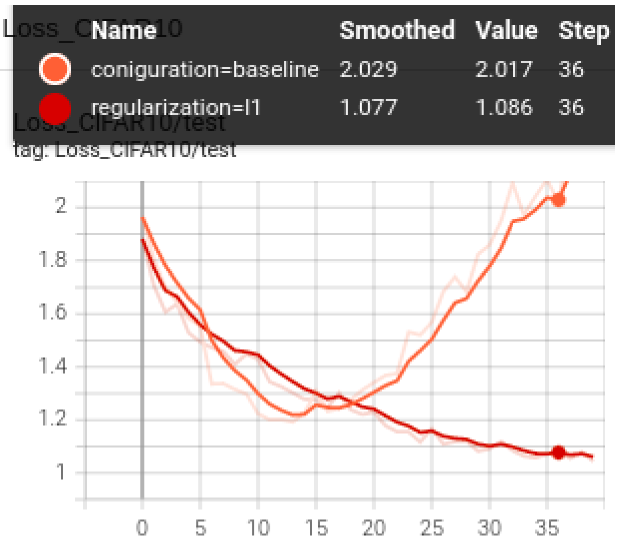
Bonus:

L1 regularization vs Baseline: (Loss/Accuracies vs Number of epochs)

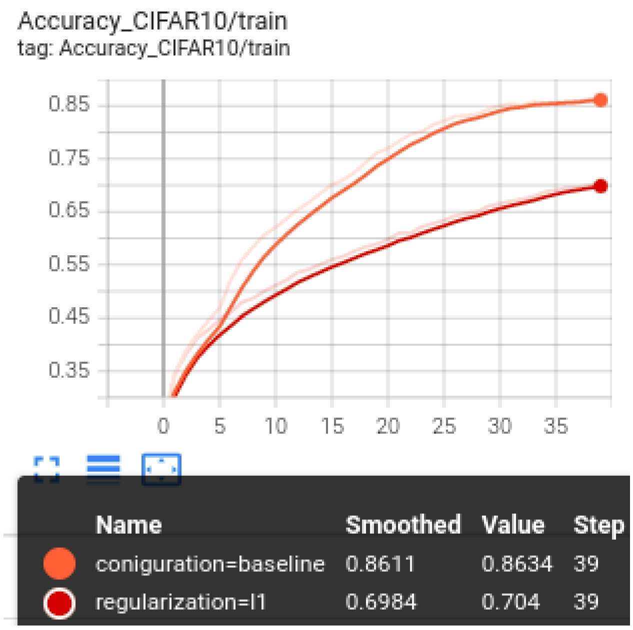
Train loss



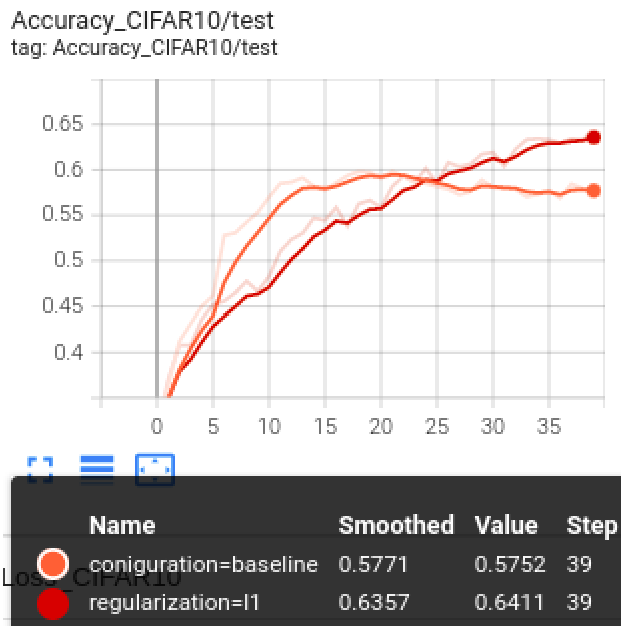
Test Loss



Train Accuracy



Test Accuracy



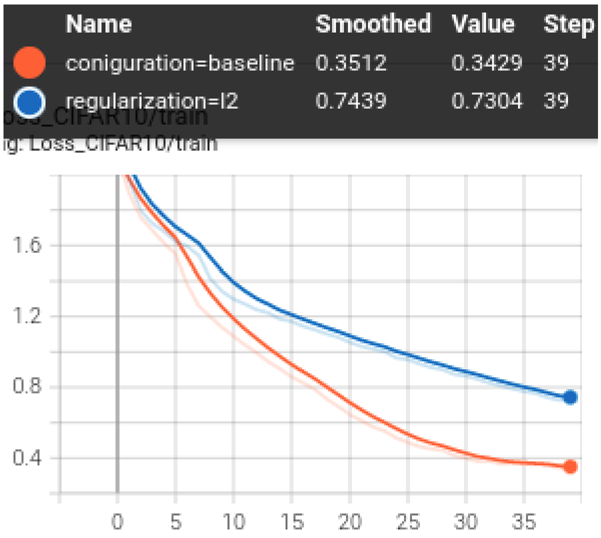
Comment:

From the above graphs, it is observed that, the L1 model successfully regularized. The test loss curves continuously decreased with very small ups and downs. Although the baseline train accuracy is high, the model with “L1” regularizer scored the highest test accuracy and least test loss for the taken number of epochs. The train and test loss curves of the “L1” model are still moving downwards after 39 epochs. We can expect better test accuracy if iterations are increased. This shows the effectiveness of “L1” regularizer in regularizing the model. The L1 reugalrizer here is reducing the complexity of the model by forcing the weights o irrelevant features to zero thereby regularizing the model.

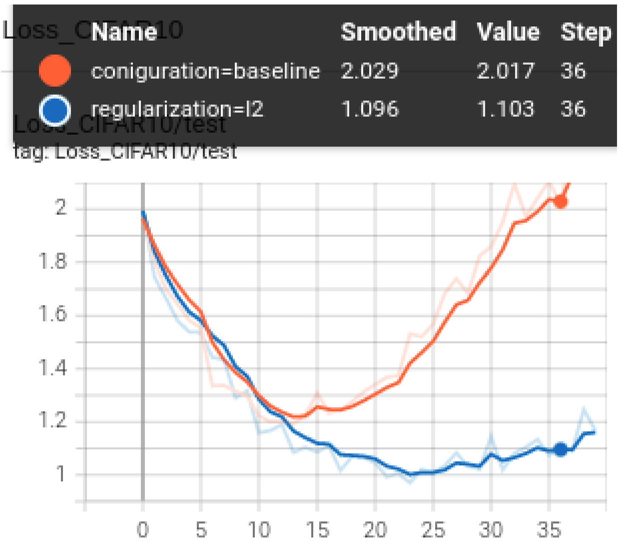
Bonus:

L2 regularization vs Baseline: (Loss/Accuracies vs Number of epochs)

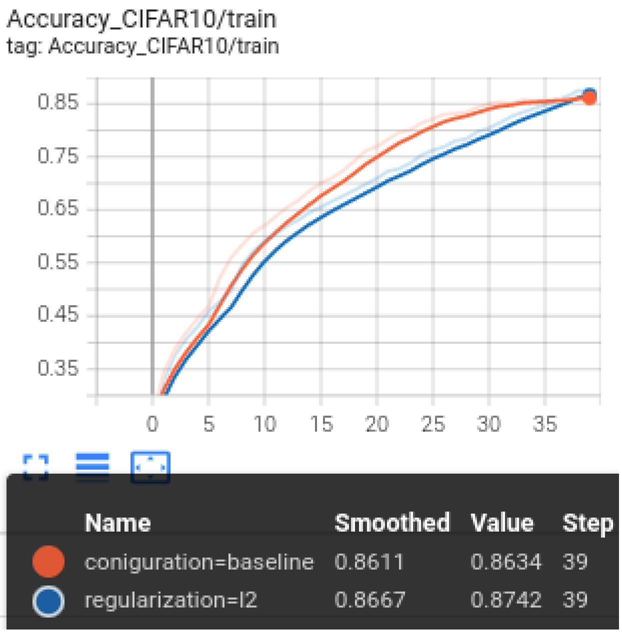
Train loss



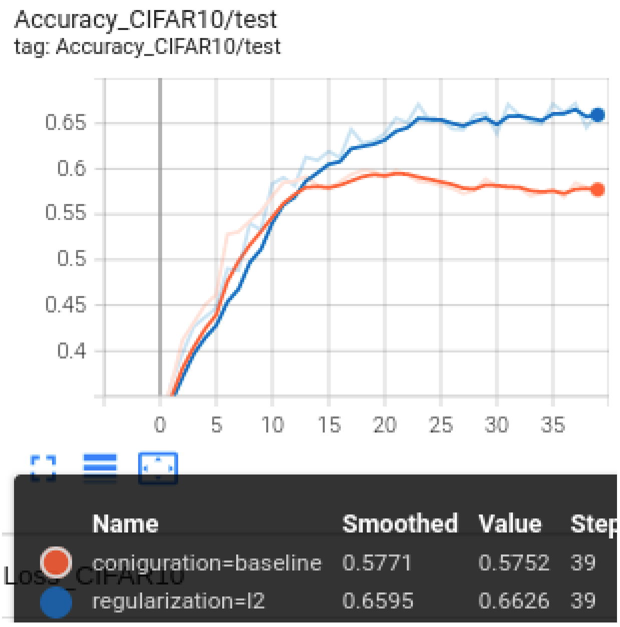
Test Loss



Train Accuracy



Test Accuracy



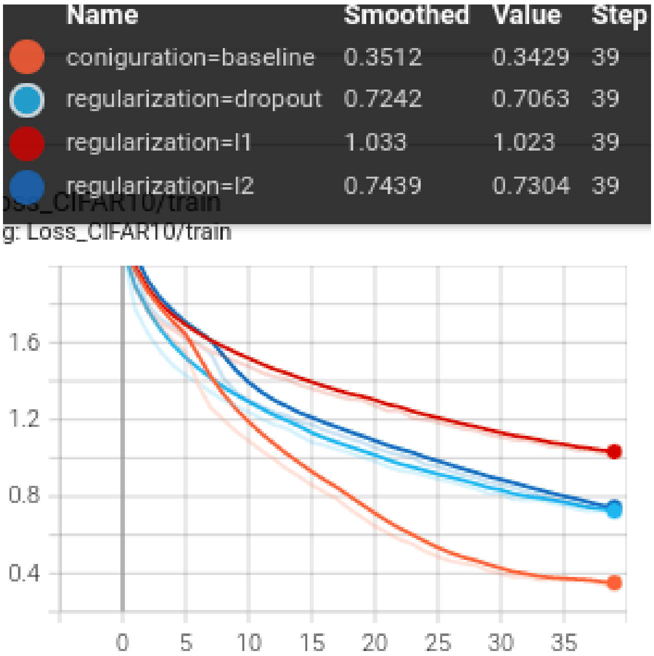
Comment:

From the above graphs, it is observed that, the L2 model regularized the model upto around 25 epochs. The test loss curves continuously decreased with very small ups and downs upto 25 epochs. At this 25th epoch, Although the baseline train accuracy is high, we can observe model with L2 regularizer scoring the highest test accuracy and least test loss. Ater 25th epoch, test loss curves started moving upwards showing the overfitting of the model.

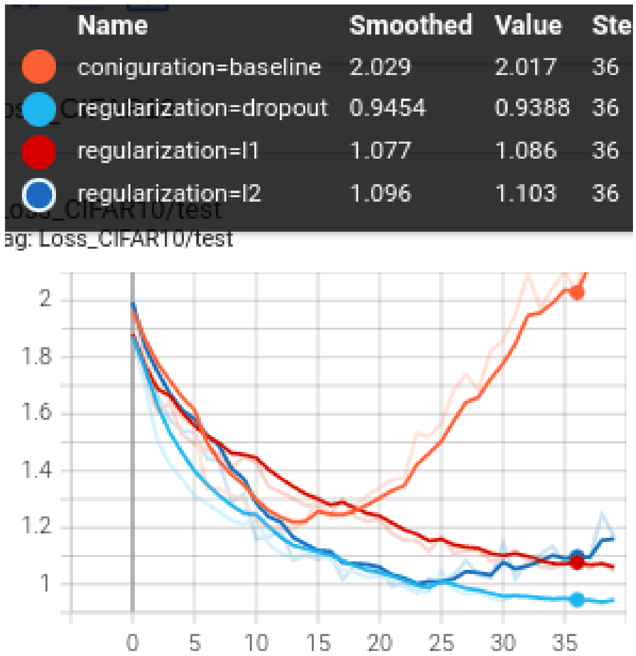
This shows that the penalty “lamda” =0.001 is not sufficient to regularize the model. The penalty factor is to be further increased to regularize the model.

Comparison with all regularizers vs Baseline: (Loss/Accuracies vs Number of epochs)

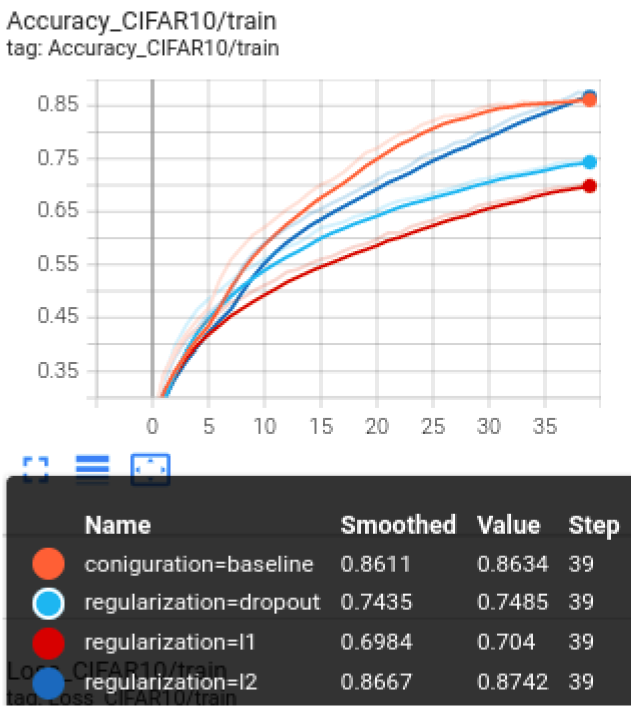
Train loss



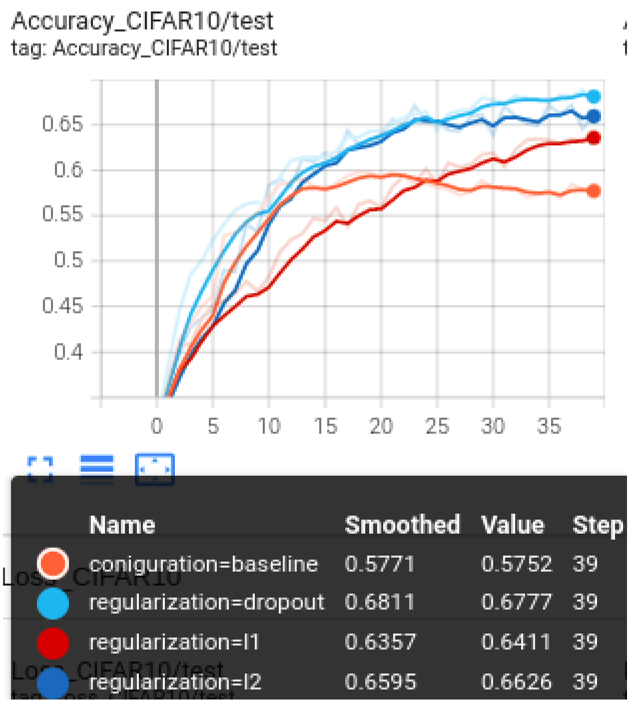
Test Loss



Train Accuracy



Test Accuracy



Comparison:

When all regularizers are compared, the model with “dropout” regularizer showed the best accuracies and least test losses than others.

Exercise 3: Optimizers (CNN)

Approach Data 50 percent of data is taken with only baseline transformations i.e "toTensor()" as to have comparison with baseline. Here baseline optimizer is also Adam with learning rate 0.001. It is alos a part of following excercise.

Models: the original model used for baseline i.e "base" is used.

Learning For learning with different optimizers i.e SGD and Adam, a new function called "learning_lr" is created. This funtion takes the name of the optimizer as argument and learns with different learning rates i.e [0.01,0.001,0.00001]. Rest of the learning methodology is same as that of the baseline model. The respective train/test losses and accuracies are recorded in tensprboard for each combination of optimzer and learning rate. The same are analyzed below.

In [15]:
```
def optim_sel(model,opt,lr):
    if opt=="Adam":
        optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    else:
        optimizer = torch.optim.SGD(model.parameters(), lr=lr)
    return optimizer
```

In [33]:
```
def learning_lr(optim,kernel):
    for lr in [0.01,0.001,0.00001]:
        width,in_ch=32,3
        model = base(in_ch,width,kernel)
        optimizer = optim_sel(model,optim,lr)
        criterion = nn.CrossEntropyLoss()
        train_loss_epoch_l,test_loss_epoch_l=[],[]
        writer = SummaryWriter(f"conf/optimizer={optim}_lr={lr}")
        print(f"for optimizer={optim}_lr={lr}")
        for j in range(40):
            # training
            model.train()
            train_loss_h,train_pred_l,test_pred_l,test_label,train_label=0,[],[],[],[]
            for images, labels in trainloader_CIF:
                optimizer.zero_grad()
                y_hat_train = model(images)
                prob=F.softmax(y_hat_train, dim=1)
                pred=[torch.argmax(j) for j in prob]
                train_loss = criterion(y_hat_train, labels)
                train_loss_h+=train_loss.item()*len(images)
                train_pred_l=train_pred_l+pred
                train_label=train_label+list(labels)
                train_loss.backward()
                optimizer.step()
            train_loss_epoch=np.round((train_loss_h/len(train_dataset)),4)
            train_loss_epoch_l.append(train_loss_epoch)
            train_acc=np.round((np.array(train_pred_l)==np.array(train_label)).mean(),4)
            # testing
            model.eval()
            with torch.no_grad():
                test_loss_h=0
                for images, labels in testloader_CIF:
                    y_hat_test = model(images)
                    prob=F.softmax(y_hat_test, dim=1)
                    pred=[torch.argmax(j) for j in prob]
                    test_loss = criterion(y_hat_test, labels)
                    test_loss_h+=test_loss.item()*len(images)
                    test_pred_l=test_pred_l+pred
                    test_label=test_label+list(labels)
            test_acc=np.round((np.array(test_pred_l)==np.array(test_label))).mean()
            test_loss_epoch=np.round((test_loss_h/len(test_dataset)),4)
            test_loss_epoch_l.append(test_loss_epoch)
            writer.add_scalar('Loss_CIFAR10/train', train_loss_epoch, j)
            writer.add_scalar('Loss_CIFAR10/test', test_loss_epoch, j)
            writer.add_scalar('Accuracy_CIFAR10/train', train_acc, j)
            writer.add_scalar('Accuracy_CIFAR10/test', test_acc, j)
            print(f"Epoch {j} - train_loss : {train_loss_epoch},test loss : {test_loss_epoch},train_acc : {trai
        print("                                                                                  ")
```

In [ ]:
```
torch.manual_seed(4)
for optim in ["SGD","Adam"]:
    learning_lr(optim,kernel)
```

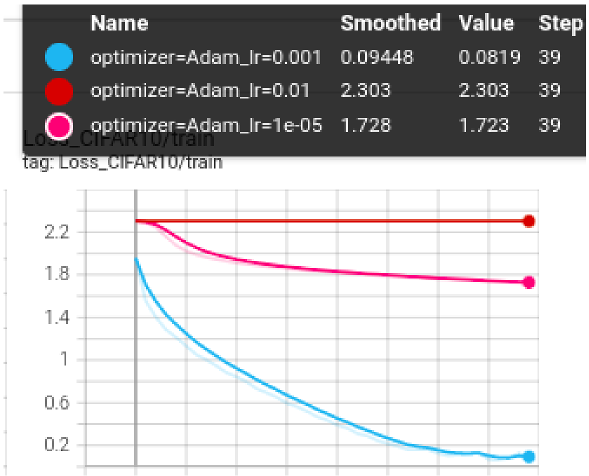
In [ ]:

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

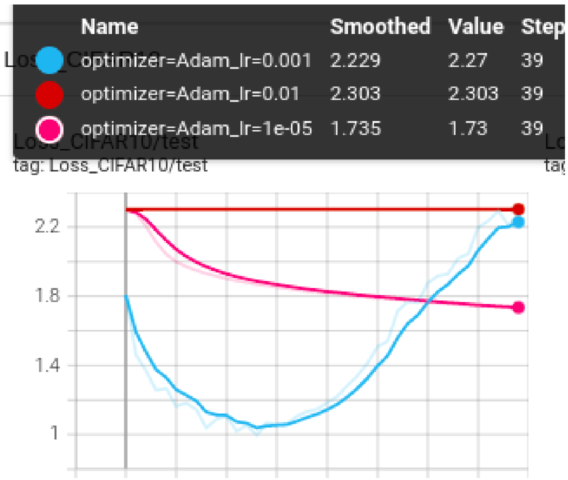
Exercise 3: Optimizers (CNN) Analysis

Adam optimizer vs lr=[0.01,0.001,0.00001]: (Loss/Accuracies vs Number of epochs)

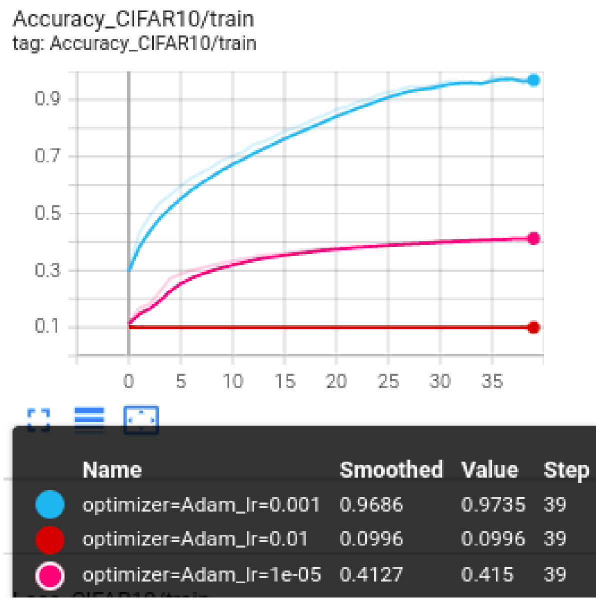
Train loss



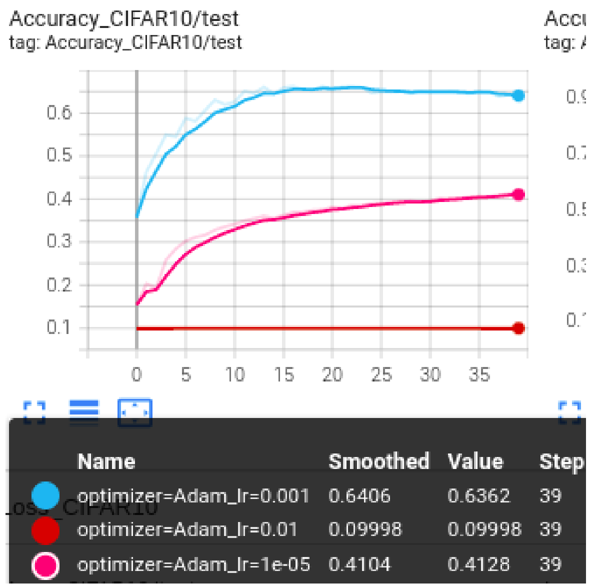
Test Loss



Train Accuracy



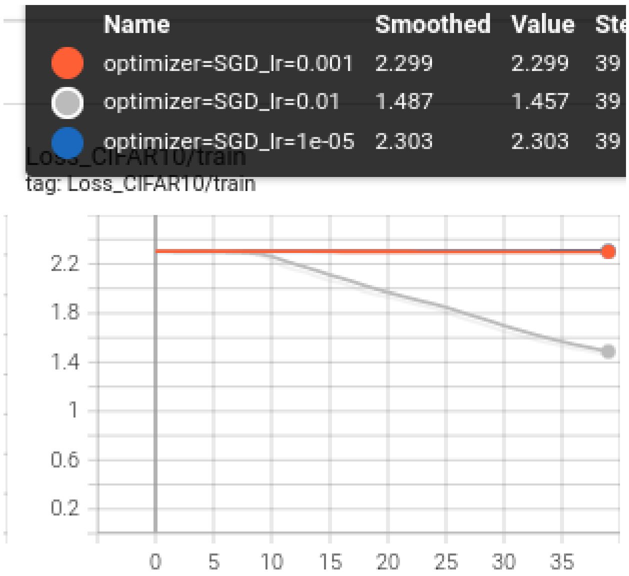
Test Accuracy



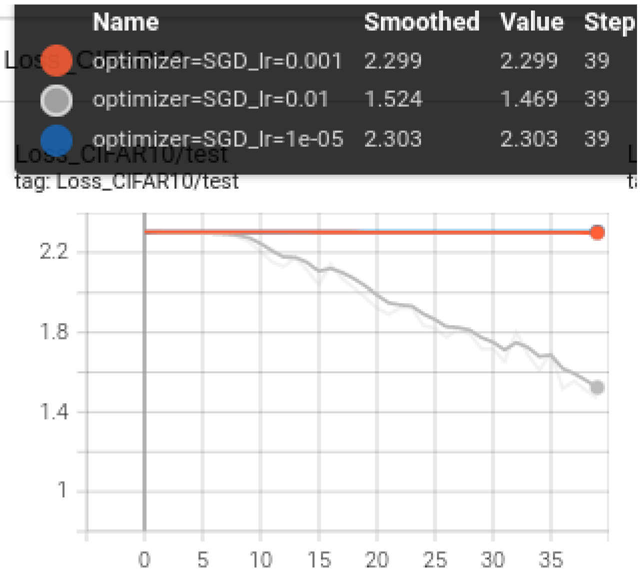
Comment: From the above curves it is observed that, adam optimizer is flexible with learning rates 0.001 and 0.00001 unless it is chosen so high as lr=0.01. With this 0.01 learning rate, the model is not learning anything. Although the model is overfitting with lr=0.001, it can be delta by regularizing. With lr=0.00001, the model is getting converged way faster at high train/test loss than with lr=0.001.

SGD optimizer vs **lr=[0.01,0.001,0.00001]:**(Loss/Accuracies vs Number of epochs)

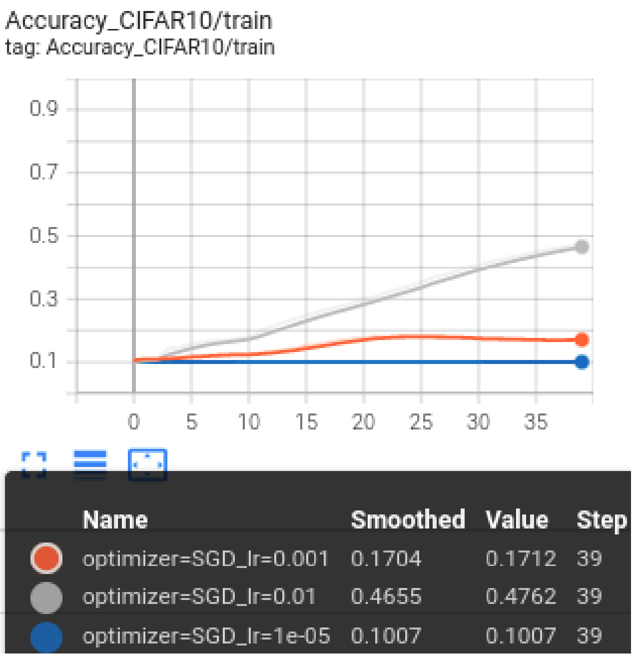
Train loss



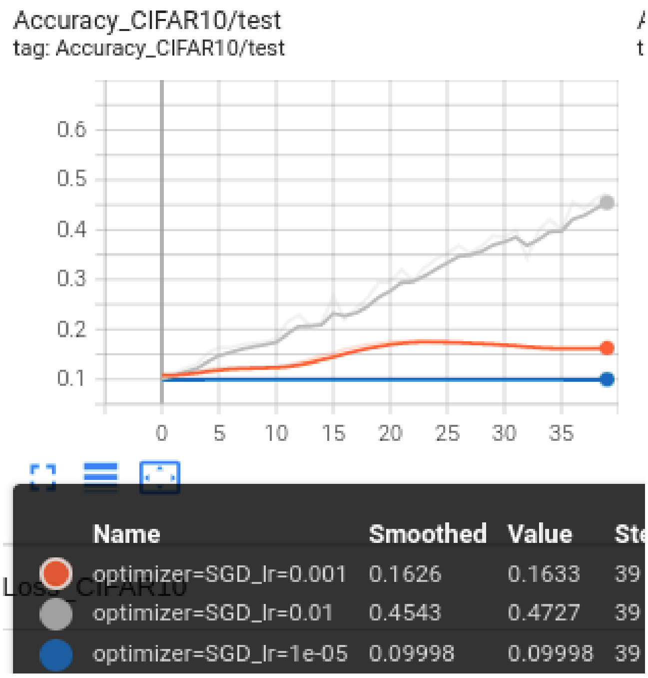
Test Loss



Train Accuracy



Test Accuracy



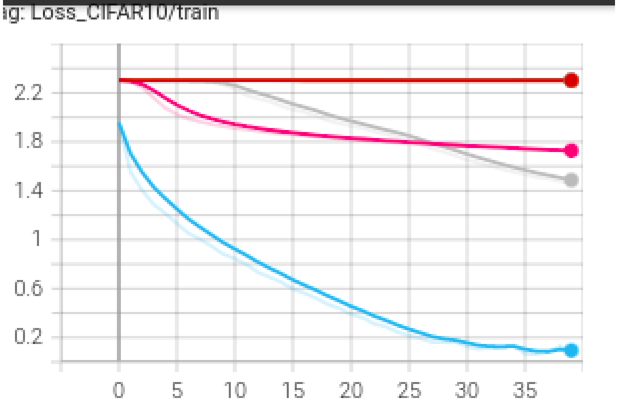
Comment:

From the above curves it is observed that, SGD optimizer is not flexible with all learning rates. It tried to learn only when lr=0.01. At other smaller learning rates, it failed to learn. It is understood that, for SGD, the initial learning rate is to be carefully chosen. Otherwise, the model will end up without learning anything.

Comparing SGD and Adam: (Loss/Accuracies vs Number of epochs)

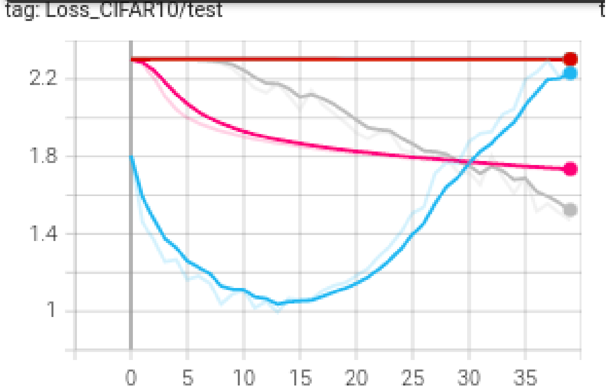
Train loss

Name	Smoothed	Value	Step
optimizer=Adam_lr=0.001	0.09448	0.0819	39
optimizer=Adam_lr=0.01	2.303	2.303	39
optimizer=Adam_lr=1e-05	1.728	1.723	39
optimizer=SGD_lr=0.001	2.299	2.299	39
optimizer=SGD_lr=0.01	1.487	1.457	39
optimizer=SGD_lr=1e-05	2.303	2.303	39

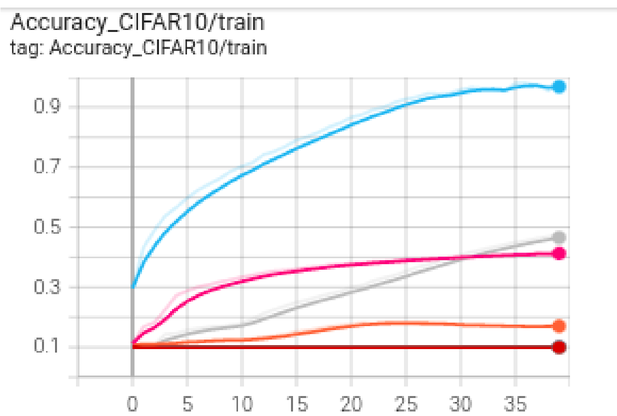


Test Loss

Name	Smoothed	Value	Step
optimizer=Adam_lr=0.001	2.229	2.27	39
optimizer=Adam_lr=0.01	2.303	2.303	39
optimizer=Adam_lr=1e-05	1.735	1.73	39
optimizer=SGD_lr=0.001	2.299	2.299	39
optimizer=SGD_lr=0.01	1.524	1.469	39
optimizer=SGD_lr=1e-05	2.303	2.303	39

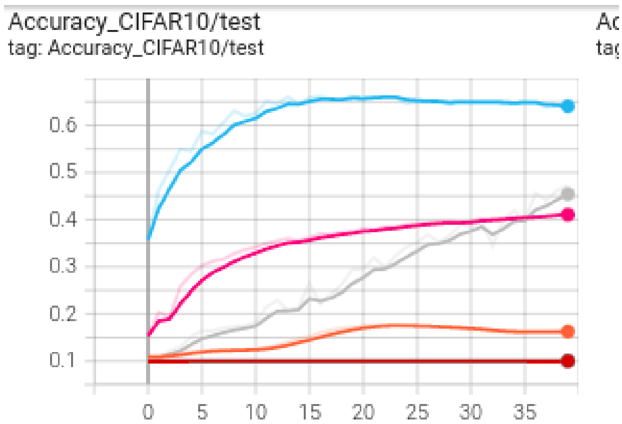


Train Accuracy



Name	Smoothed	Value	Step
optimizer=Adam_lr=0.001	0.9686	0.9735	39
optimizer=Adam_lr=0.01	0.0996	0.0996	39
optimizer=Adam_lr=1e-05	0.4127	0.415	39
optimizer=SGD_lr=0.001	0.1704	0.1712	39
optimizer=SGD_lr=0.01	0.4655	0.4762	39
optimizer=SGD_lr=1e-05	0.1007	0.1007	39

Test Accuracy



Name	Smoothed	Value	Step
optimizer=Adam_lr=0.001	0.6406	0.6362	39
optimizer=Adam_lr=0.01	0.09998	0.09998	39
optimizer=Adam_lr=1e-05	0.4104	0.4128	39
optimizer=SGD_lr=0.001	0.1626	0.1633	39
optimizer=SGD_lr=0.01	0.4543	0.4727	39
optimizer=SGD_lr=1e-05	0.09998	0.09998	39

Comment: Among SGD and Adam, for the taken learning rates, training loss decreased fast for ADAM with lr=0.001. But later the model is overfitted. This can be overcome by using a regularizer. Here in this example, for Adam,lr=0.001 will be good learning rate with some regularizer. And the SGD with lr=0.01 showed no overfitting and performed better than that of Adam optimizer with lr=0.0001. Overall, if overfitting can be avoided, the Adam optimizer with lr=0.001 can perform better than SGD.