DDA LAB

Simran Kaur

311443

Exercise 6

1 Exercise 1: PyTorch Network Analysis (10 points)

For this first task, you are required to implement LeNet (see Annex) for the following datasets for image classification:

- MNIST
- CIFAR10

Both datasets are available in the PyTorch Datasets library and you may use the library to service the model. **Note**: LeNet was originally written for MNIST dataset which has 1-Channel black and white images, so you would need to keep that in mind when you want to implement it for CIFAR10, which is a 3-Channel RGB image dataset. Furthermore, the image size for the two datasets is different, that is something to keep in mind as well when adapting LeNet for CIFAR10.

Some configurations for you to try when training MNIST and CIFAR are as follows:

- Learning rate [0.1, 0.01, 0.001]
- Optimizer [SGD, ADAM, RMSPROP]

Note that you do not necessarily need to provide results for all combinations of these configurations. A valid approach would be to fix the choice of the Optimizer and try different learning rates for that Optimizer only.

One thing that is essential to learn when training network models is using **Tensorboard with PyTorch**. Tensorboard enables us to visualize the network performance in an effective manner. Please report the Train/Test **accuracy** and Train/Test **loss** using Tensorboard. Refer to the Annex section for useful tips.

BONUS (3 points):
In addition to the Train/Test metrics, please also visualize the activation maps in the Tensorboard. Activation maps enable us to see what the network is seeing as our inputs traverse the network and are indicative of the receptive field of our CONV layers. Vary the kernel size in the CONV layers and show the difference it creates in our activation maps.
NOTE: When you modify the kernel size, it impacts the output size of the CONV layer and the input/output sizes of the layers following the change in kernel size would need to be adapted as well.
Try 2 different kernel sizes and report the impact they have on the activation maps.

Libraries

In [548…

```
import torch
from torch.nn import Module
from torch import nn
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Dataset
from torch.utils.tensorboard import SummaryWriter
```

```
import torch.nn.functional as F
import matplotlib.pyplot as plt
```

MNIST Dataset

In [338...
```
trainingSet = torchvision.datasets.MNIST(root = './data', train = True, transform =
testSet = torchvision.datasets.MNIST(root = './data', train = False, transform = tra
```

Input to our LeNet Model is 32 by 32 image whereas the images in MNIST are 28 by 28.

We will pad the images with zeros on all sides.

In [339...
```
def transformedImg(dataset):
    pad = nn.ZeroPad2d(2)
    transformedSet = []
    for i in range(len(dataset)):
        img, label = dataset[i]
        pad_img = pad(img)
        transformedSet.append((pad_img, label))
    return transformedSet
```

In [344...
```
train_set = transformedImg(trainingSet)
test_set = transformedImg(testSet)
```

In [345...
```
train_set[50][0].size()
```

Out[345... `torch.Size([1, 32, 32])`

In [346...
```
train_dataloader = DataLoader(train_set, batch_size = 256, shuffle=True)
test_dataloader = DataLoader(test_set, batch_size = 256, shuffle=True)
```

LeNet Architecture

The LeNet Architecture takes in an image of size 3232 *with 1 channel and converts it into an image of size 2828* by applying a kernel of size 5*5 and the rest layers are shown as in the architecture. The final output is a tensor of size 10 where each value represents probabilities corresponding to each class.

In [331...
```
class LeNet(Module):
    def __init__(self, input_channels):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(input_channels, 6, kernel_size = 5)
        self.pool1 = nn.AvgPool2d(kernel_size = 2, stride = 2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size = 5)
        self.pool2 = nn.AvgPool2d(kernel_size = 2, stride = 2)
        self.conv3 = nn.Conv2d(16, 120, kernel_size = 5)
        self.fc1 = nn.Linear(120, 84)
        self.fc2 = nn.Linear(84, 10)
```

```
    def forward(self, x):
        y = self.conv1(x)
        y = F.relu(self.pool1(y))
        y = self.conv2(y)
        y = F.relu(self.pool2(y))
        y = F.relu(self.conv3(y))
        y = y.view(-1, 120*1*1)
        y = F.relu(self.fc1(y))
        y = self.fc2(y)
        return y
```

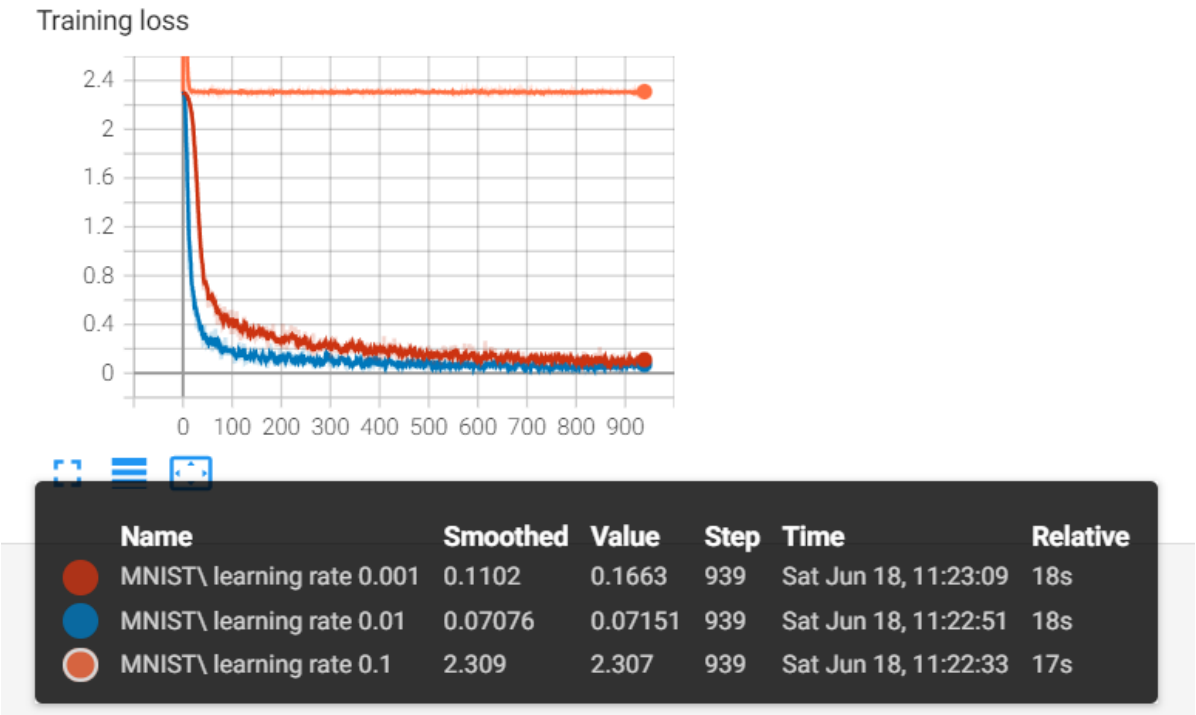
Training loss and Accuracy

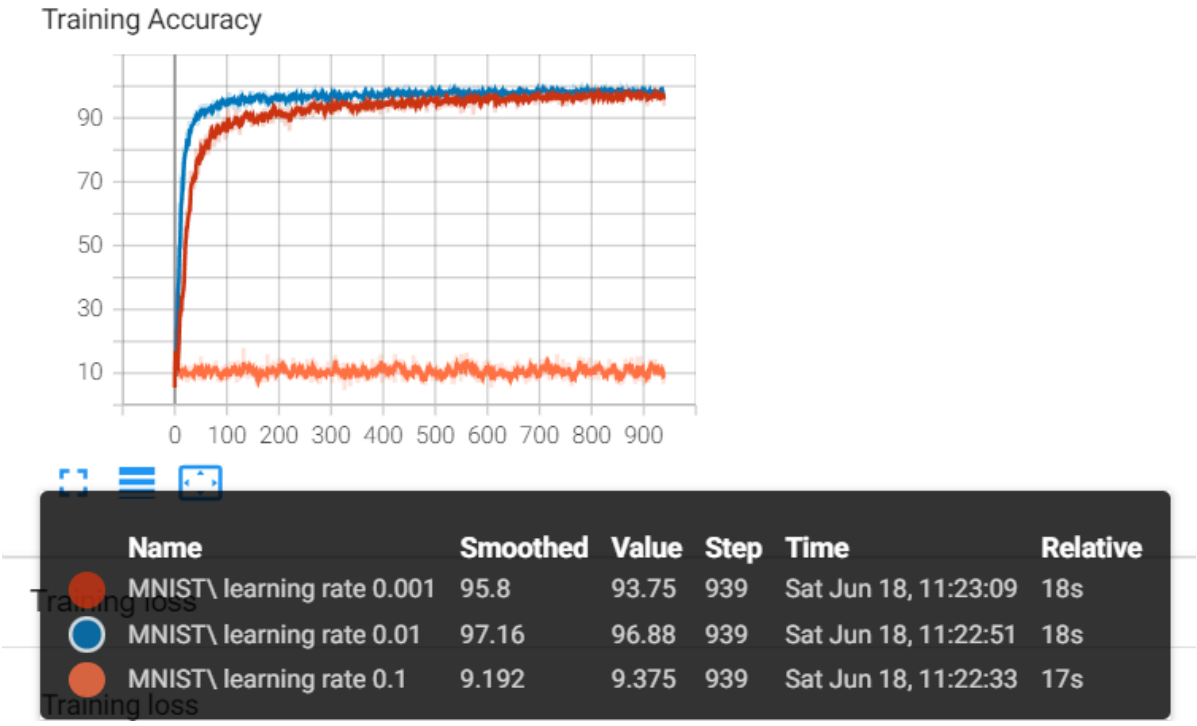
In [332…

```
n_epochs = 4
learningRate = [0.1, 0.01, 0.001]
for learning_rate in learningRate:
    step = 0
    print('------------------------')
    print(f'learning rate-------- {learning_rate}')
    net = LeNet(1)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(),lr = learning_rate)
    writer = SummaryWriter(f'runs/MNIST/ learning rate {learning_rate}')
    for epoch in range(n_epochs):
        correct_pred = 0
        sample_size = 0
        for i, batch in enumerate(train_dataloader):
            optimizer.zero_grad()
            yhat = net(batch[0])
            loss = criterion(yhat.squeeze(), batch[1].squeeze())
            pred = torch.max(yhat.data, 1)[1]
            sample_size = batch[1].size(0)
            correct_pred = (pred == batch[1]).sum().item()
            accuracy = (correct_pred/sample_size)*100
            loss.backward()
            optimizer.step()
            writer.add_scalar('Training loss', loss, global_step = step)
            writer.add_scalar('Training Accuracy', accuracy, global_step = step)
            step += 1
        print(f'Training loss after epoch {epoch} is {loss.item()}')
        print(f' Accuracy : {accuracy}')
        print('\n')
```

```
------------------------
learning rate-------- 0.1
Training loss after epoch 0 is 2.3043553829193115
 Accuracy : 10.416666666666668


Training loss after epoch 1 is 2.321345329284668
 Accuracy : 12.5


Training loss after epoch 2 is 2.295448064804077
 Accuracy : 11.458333333333332


Training loss after epoch 3 is 2.3074405193328857
 Accuracy : 9.375
```

```
------------------------
learning rate-------- 0.01
Training loss after epoch 0 is 0.15727464854717255
 Accuracy : 93.75


Training loss after epoch 1 is 0.07680729776620865
 Accuracy : 97.91666666666666


Training loss after epoch 2 is 0.14085617661476135
 Accuracy : 96.875


Training loss after epoch 3 is 0.07151161879301071
 Accuracy : 96.875


------------------------
learning rate-------- 0.001
Training loss after epoch 0 is 0.2940022647380829
 Accuracy : 90.625


Training loss after epoch 1 is 0.22409455478191376
 Accuracy : 94.79166666666666


Training loss after epoch 2 is 0.129289910197258
 Accuracy : 95.83333333333334


Training loss after epoch 3 is 0.16629204154014587
 Accuracy : 93.75
```





The training loss tends to decrease the most for the learning rate 0.01 and the accuracy is also highest corresponding to that learning rate.

Test loss and Accuracy

In [275…

```
with torch.no_grad():
    correct_pred = 0
    sample_size = 0
    for i, (img, label) in enumerate(test_dataloader):
        optimizer.zero_grad()
        output = net(img)
        loss = criterion(output.squeeze(), label.squeeze())
        pred = torch.max(output.data, 1)[1]
        sample_size += label.size(0)
        correct_pred += (pred == label).sum().item()
    accuracy = (correct_pred/sample_size)*100
    print(f'Test loss: {loss.item()}')
    print(f' Accuracy : {accuracy}')
```

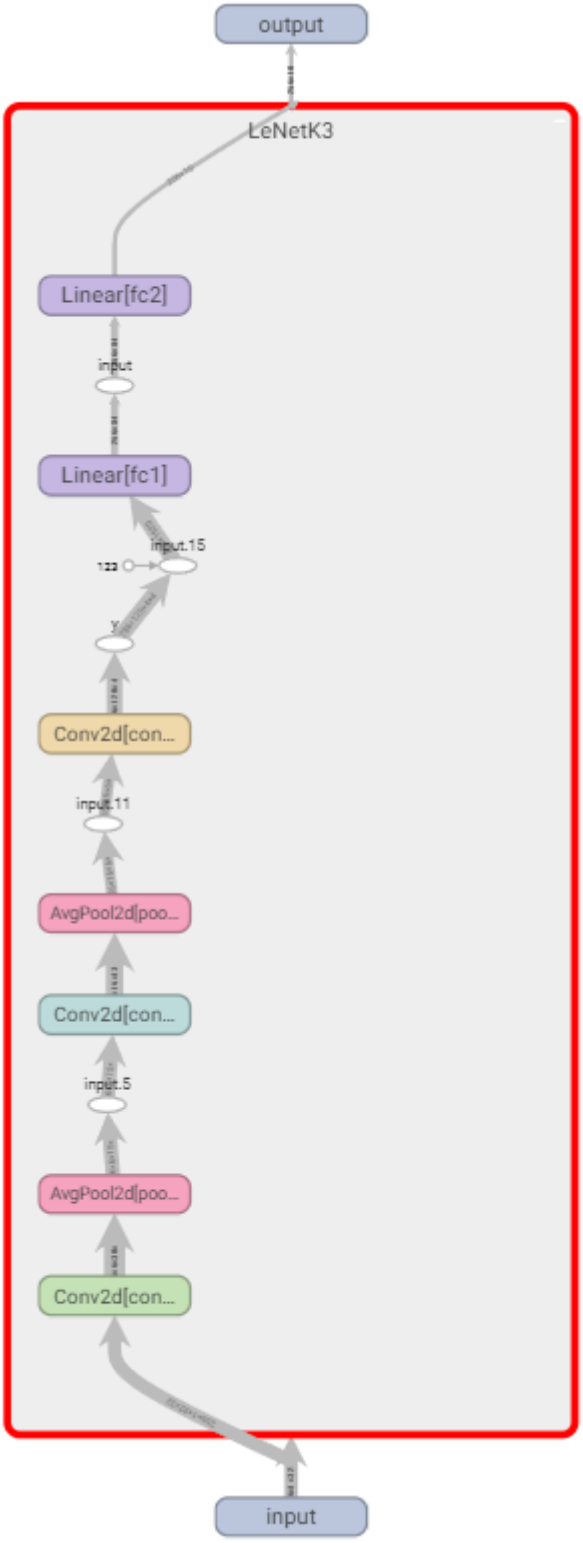
```
Test loss: 3.725289943190546e-08
 Accuracy : 98.79
```

Kernel size = (3*3)

In [347…

```
class LeNetK3(Module):
    def __init__(self, input_channels):
        super(LeNetK3, self).__init__()
        self.conv1 = nn.Conv2d(input_channels, 6, kernel_size = 3)
        self.pool1 = nn.AvgPool2d(kernel_size = 2, stride = 2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size = 3)
        self.pool2 = nn.AvgPool2d(kernel_size = 2, stride = 2)
        self.conv3 = nn.Conv2d(16, 120, kernel_size = 3)
        self.fc1 = nn.Linear(1920, 84)
        self.fc2 = nn.Linear(84, 10)
```

```
    def forward(self, x):
        y = self.conv1(x)
        y = F.relu(self.pool1(y))
        y = self.conv2(y)
        y = F.relu(self.pool2(y))
        y = F.relu(self.conv3(y))
        y = y.view(-1, 120*4*4)
        y = F.relu(self.fc1(y))
        y = self.fc2(y)
        return y

netK3 = LeNetK3(input_channels=1)

optimizer= torch.optim.Adam(netK3.parameters(),lr=0.01)

writer = SummaryWriter(f'runs/Kernel3/ Kernel size = 3')

image, label = next(iter(train_dataloader))
writer.add_graph(netK3, image)
```

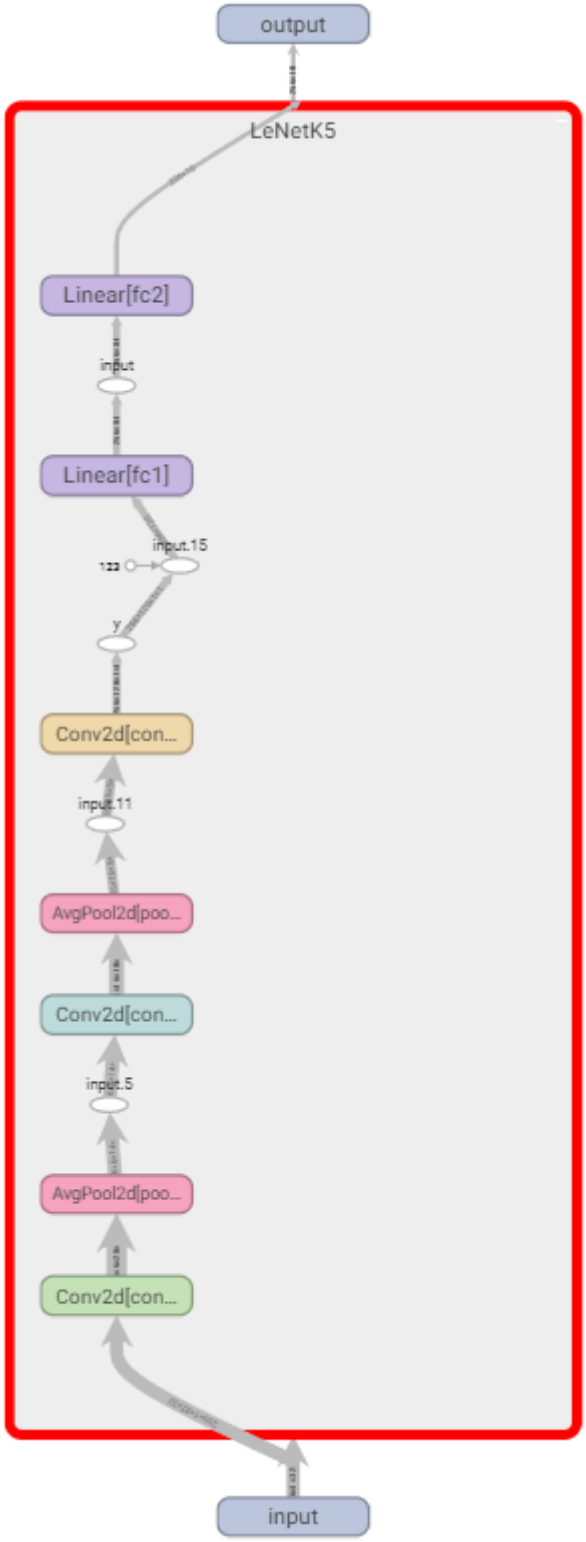


Kernel Size = (5*5)

In [348...

```
class LeNetK5(Module):
    def __init__(self, input_channels):
        super(LeNetK5, self).__init__()
        self.conv1 = nn.Conv2d(input_channels, 6, kernel_size = 5)
        self.pool1 = nn.AvgPool2d(kernel_size = 2, stride = 2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size = 5)
        self.pool2 = nn.AvgPool2d(kernel_size = 2, stride = 2)
        self.conv3 = nn.Conv2d(16, 120, kernel_size = 5)
        self.fc1 = nn.Linear(120, 84)
        self.fc2 = nn.Linear(84, 10)
```

```
    def forward(self, x):
        y = self.conv1(x)
        y = F.relu(self.pool1(y))
        y = self.conv2(y)
        y = F.relu(self.pool2(y))
        y = F.relu(self.conv3(y))
        y = y.view(-1, 120*1*1)
        y = F.relu(self.fc1(y))
        y = self.fc2(y)
        return y

netK5 = LeNetK5(input_channels=1)

optimizer= torch.optim.Adam(netK5.parameters(),lr=0.01)

writer = SummaryWriter(f'runs/Kernel5/ Kernel size = 5')

image, label = next(iter(train_dataloader))
writer.add_graph(netK5, image)
```



CIFAR10 Dataset

In [150...

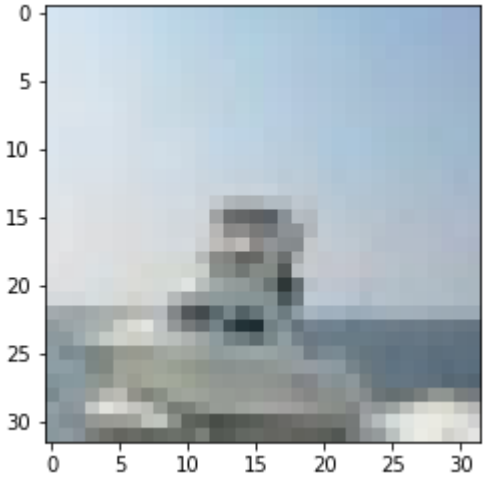
```
CifarTrain = torchvision.datasets.CIFAR10(root = './data', train = True, transform =
CifarTest = torchvision.datasets.CIFAR10(root = './data', train = False, transform =
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data\cifar-
10-python.tar.gz

Extracting ./data\cifar-10-python.tar.gz to ./data
```

Plotting a sample image from CIFAR10

In [295…
```
sample = CifarTrain[100][0]
plt.imshow(sample.permute(1, 2, 0))
```

Out[295…
```
<matplotlib.image.AxesImage at 0x218937f2790>
```



In [170…
```
classes = CifarTrain.classes
classes
```

Out[170…
```
['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']
```

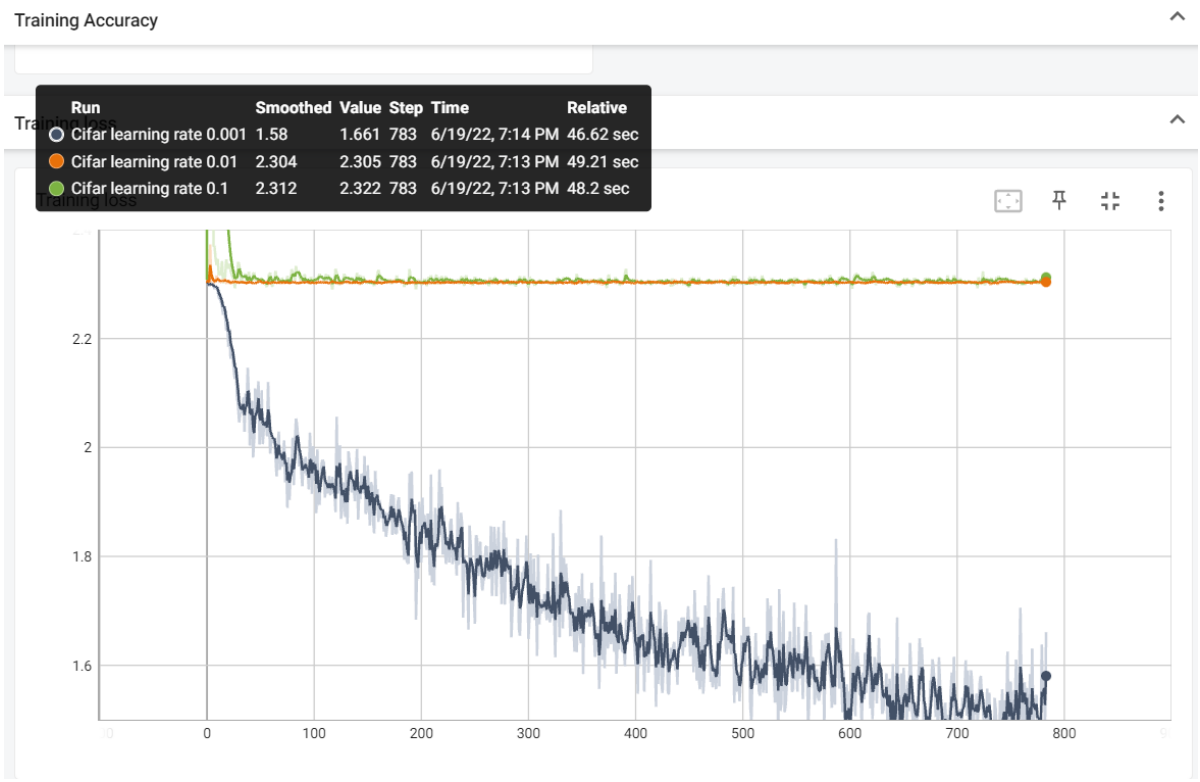
In [747…
```
train_loader = DataLoader(CifarTrain, batch_size = 256, shuffle=True)
test_loader = DataLoader(CifarTest, batch_size = 256, shuffle = True)
```

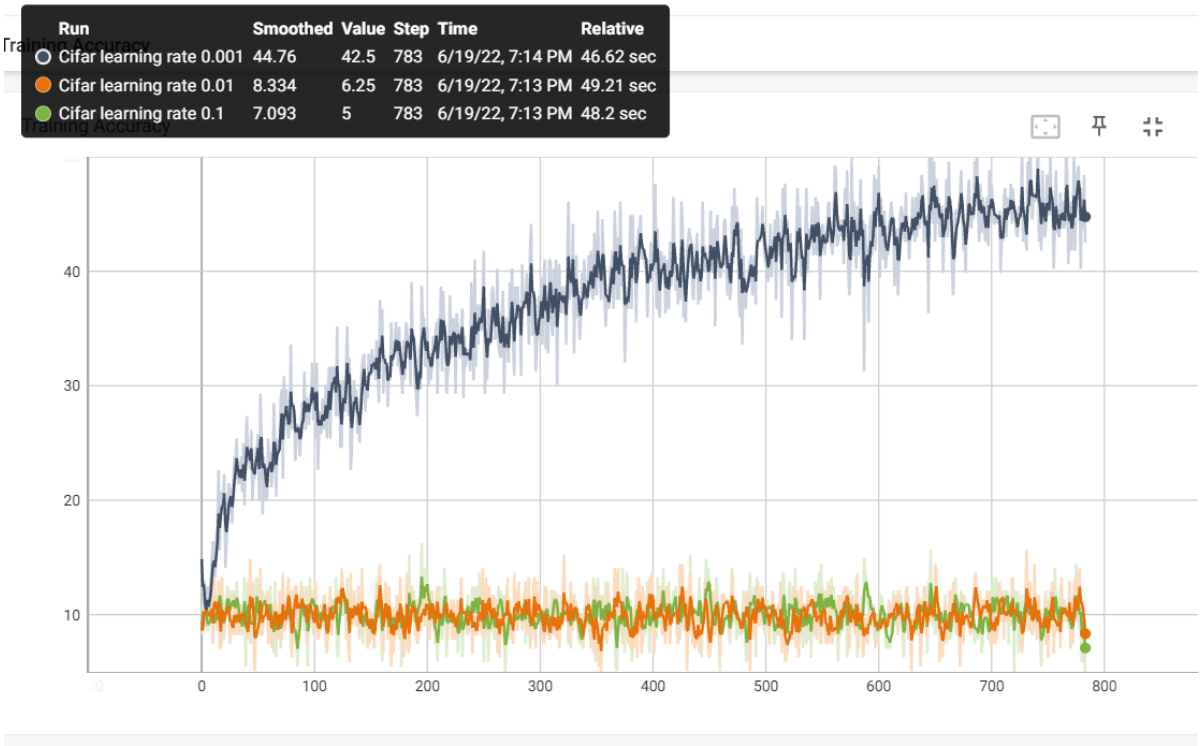
Training Loss and Accuracy

In [750…
```
n_epochs = 4
learningRate = [0.1, 0.01, 0.001]
for learning_rate in learningRate:
    step = 0
    print('------------------------')
    print(f'learning rate-------- {learning_rate}')
    netC = LeNet(3)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(netC.parameters(),lr = learning_rate)
    writer = SummaryWriter(f'runs/Cifar learning rate {learning_rate}')
    for epoch in range(n_epochs):
        correct_pred = 0
        sample_size = 0
        for i, batch in enumerate(train_loader):
            optimizer.zero_grad()
            yhat = netC(batch[0])
            loss = criterion(yhat.squeeze(), batch[1].squeeze())
            pred = torch.max(yhat.data, 1)[1]
            sample_size = batch[1].size(0)
            correct_pred = (pred == batch[1]).sum().item()
```

```
            accuracy = (correct_pred/sample_size)*100
            loss.backward()
            optimizer.step()
            writer.add_scalar('Training loss', loss, global_step = step)
            writer.add_scalar('Training Accuracy', accuracy, global_step = step)
            step += 1
        print(f'Training loss after epoch {epoch} is {loss.item()}')
        print(f' Accuracy : {accuracy}')
        print('\n')
```

```
------------------------
learning rate-------- 0.1
Training loss after epoch 0 is 2.2902486324310303
 Accuracy : 16.25


Training loss after epoch 1 is 2.328268051147461
 Accuracy : 7.5


Training loss after epoch 2 is 2.2971885204315186
 Accuracy : 15.0


Training loss after epoch 3 is 2.3218483924865723
 Accuracy : 5.0


------------------------
learning rate-------- 0.01
Training loss after epoch 0 is 2.3011512756347656
 Accuracy : 11.25


Training loss after epoch 1 is 2.3026463985443115
 Accuracy : 12.5


Training loss after epoch 2 is 2.302802085876465
 Accuracy : 7.5


Training loss after epoch 3 is 2.3047280311584473
 Accuracy : 6.25


------------------------
learning rate-------- 0.001
Training loss after epoch 0 is 1.6839525699615479
 Accuracy : 38.75


Training loss after epoch 1 is 1.6837297677993774
 Accuracy : 41.25


Training loss after epoch 2 is 1.832543134689331
 Accuracy : 31.25


Training loss after epoch 3 is 1.6606454849243164
 Accuracy : 42.5
```





Test and Loss Accuracy

In [751…

```
with torch.no_grad():
    correct_pred = 0
    sample_size = 0
    for i, (img, label) in enumerate(test_loader):
        optimizer.zero_grad()
        output = netC(img)
        loss = criterion(output.squeeze(), label.squeeze())
        pred = torch.max(output.data, 1)[1]
        sample_size += label.size(0)
        correct_pred += (pred == label).sum().item()
    accuracy = (correct_pred/sample_size)*100
```

```
    print(f'Test loss: {loss.item()}')
    print(f' Accuracy : {accuracy}')
```

```
Test loss: 1.6755192279815674
 Accuracy : 46.88
```

2 Exercise 2: Custom Task (10 points)

In Exercise 1 we implemented a well-known network for image classification. Building upon that learning, we will now extend the network to an image regression setting. Using the MNIST dataset, we will design a network that will consume (N, K, C, W, H) batches of images where N, K, C, W, H are batch size, number of images to sum, channels, width and height respectively. The output of the network will be the sum of the number in the K many images. For instance, if we have $K = 3$ and the three images contain the digits 1, 5 and 3, then the output of the network should be 9.
To setup this network, you need to write a custom dataset that will extend the basic dataset module to return a batch-shaped N, K, C, W, H data format instead of the typical N, C, W, H data shape. You should choose K many random images to populate the batch instances. You will need to adapt you network to consume this extended dataset dimension. Use the $x = x.\text{view}(N * K, C, W, H)$ trick discussed in the lab session. Your ground truth target during training will be the sum of the K many images. This can be calculated by summing the integer labels already associated with the images. The design for the network output (predictions) is left up to you, the only requirement here is that it should be a numeric value. Keeping in mind that we are no linger looking at a classification problem but rather a regression problem, our loss should also be modified to reflect that. **HINT**: Regular Maintenance Saves Energy.
For this task, you are to report:

- Train/Test Error on Tensorboard
- 10 sets of test images (each set containing K many images) and the output from your trained network (final value that your model outputs)

Please display the output from your trained network only **after** your model is done training, having a visual output will help you get in the habit of visually inspecting your network behavior.

For a dataset of size 60,000 with k = 3, the transformed dataset has size 20,000 with 3 images in each row. For this the length function is changed as below and the item function returns a set of 3 images and the sum of their labels.

In [703…

```
class MnistSet(Dataset):

    def __init__(self, k, transform = transforms.Compose([transforms.ToTensor()]), t
        super(MnistSet, self).__init__()
        self.k = k
        self.transform = transform
        self.train = train
        self.dataset = datasets.MNIST(root='./data',train = self.train, transform =

    def __len__(self):
        return len(self.dataset)//self.k

    def __getitem__(self, idx):
        k_img = []
        sum_labels = 0
        for i in range(self.k):
            img, target = self.dataset[idx*self.k + i]   # for a given index, k imag
            k_img.append(img)
            sum_labels += target
        return torch.stack(k_img, 0), sum_labels
```

Loading datasets

In [704…
```
import torchvision.datasets as datasets
transform = transforms.Compose([transforms.Resize((32,32)), transforms.ToTensor()])
train_dataset = MnistSet(k = 3, transform = transform, train = True)
Train_loader = torch.utils.data.DataLoader(train_dataset, batch_size = 64, shuffle=T
test_dataset = MnistSet(k = 3, transform = transform, train = False)
Test_loader = torch.utils.data.DataLoader(test_dataset, batch_size = 64, shuffle=Tru
```

Regression Model

The only change in the model architecture is that the model takes batch of size N with k images in each batch and hence this can be fit to our original architecture by expanding the size of the number of images held in one batch by multiplying N times k and then at the end the model retuns one value which will be the sum of these three images.

In [708…
```
class LeNetRegression(Module):
    def __init__(self, input_channels, K):
        super(LeNetRegression, self).__init__()
        self.conv1 = nn.Conv2d(input_channels, 6, kernel_size = 5)
        self.pool1 = nn.AvgPool2d(kernel_size = 2, stride = 2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size = 5)
        self.pool2 = nn.AvgPool2d(kernel_size = 2, stride = 2)
        self.conv3 = nn.Conv2d(16, 120, kernel_size = 5)
        self.fc1 = nn.Linear(120, 84)
        self.fc2 = nn.Linear(84, 1)

    def forward(self, x):
        N, K, C, W, H = x.size()
        x = x.view(N*K, C, W, H)
        y = self.conv1(x)
        y = F.relu(self.pool1(y))
        y = self.conv2(y)
        y = F.relu(self.pool2(y))
        y = F.relu(self.conv3(y))
        y = y.view(-1, 120*1*1)
        y = F.relu(self.fc1(y))
        y = self.fc2(y)
        return y
```

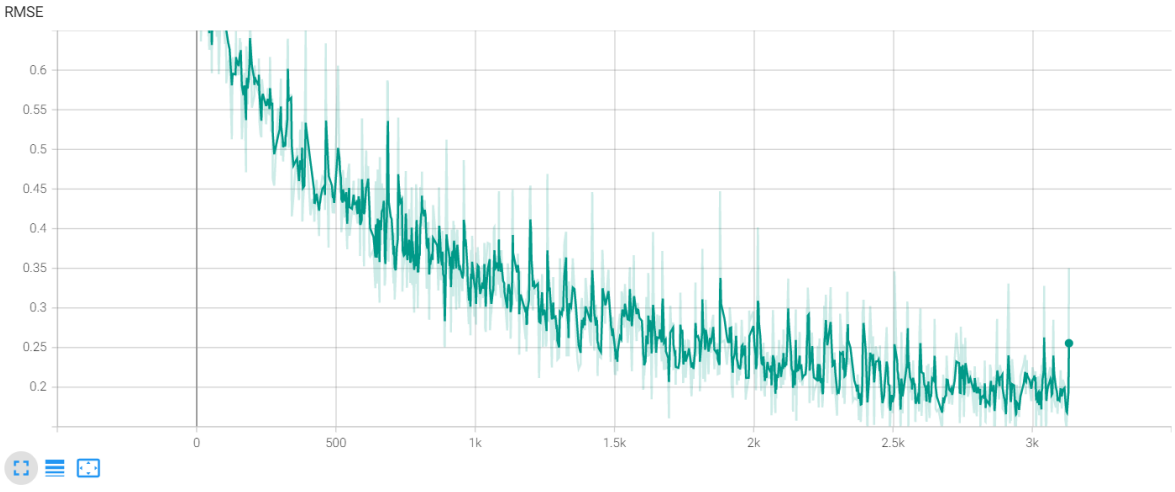
Training Loss

In [741…
```
n_epochs = 10
step = 0
writer = SummaryWriter()
netReg = LeNetRegression(1, 3)
criterion = nn.MSELoss()
optimizer = torch.optim.RMSprop(netReg.parameters(),lr = 0.001)
for epoch in range(n_epochs):
    rmse = 0
    for i, batch in enumerate(Train_loader):
        optimizer.zero_grad()
        yhat = netReg(batch[0])             # a flat vector of size N*k is received
        yhat = yhat.view(len(batch[0]), 3)       # reshape vector to sum over column
        output = torch.sum(yhat, dim=1)
        loss = criterion(output.float().squeeze(), batch[1].float().squeeze())
        rmse = torch.sqrt(torch.sum(loss)/len(batch[1]))
        writer.add_scalar('RMSE', rmse, global_step = step)
        loss.backward()
```

```
        optimizer.step()
        step += 1
    print(f'Training loss after epoch {epoch} is {rmse.item()}')
    print('\n')
writer.flush()
```

```
Training loss after epoch 0 is 0.7521438598632812


Training loss after epoch 1 is 0.6318392157554626


Training loss after epoch 2 is 0.5469512343406677


Training loss after epoch 3 is 0.49354010820388794


Training loss after epoch 4 is 0.4113883972167969


Training loss after epoch 5 is 0.287677139043808


Training loss after epoch 6 is 0.2942727208137512


Training loss after epoch 7 is 0.2318689227104187


Training loss after epoch 8 is 0.33353686332702637


Training loss after epoch 9 is 0.28384190797805786
```

For a total of around 300 set of 3 images each over 10 epochs training loss is shown below. The trend shows that as number of epochs increases, training loss decreases.

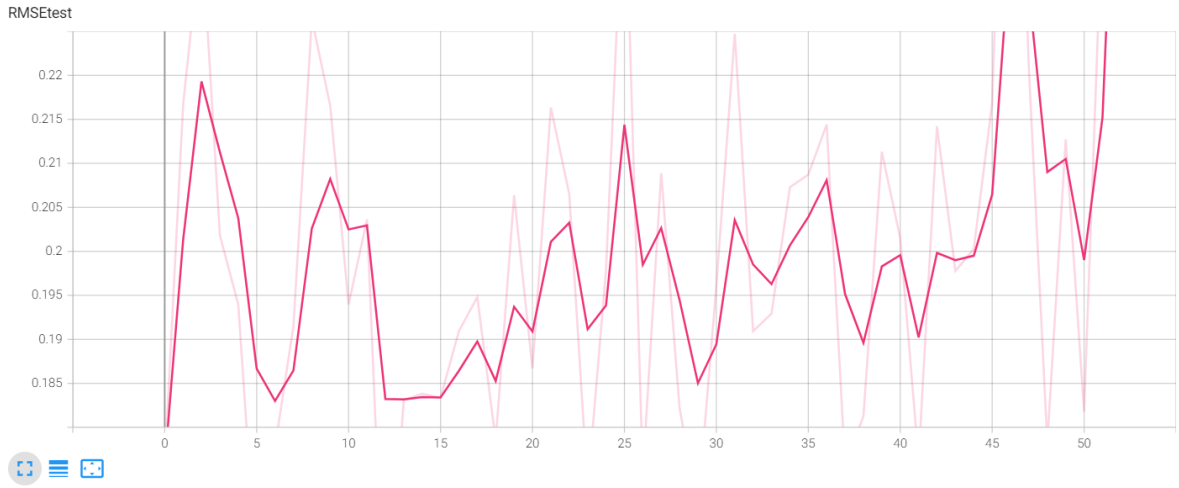


Test Loss

In [745…

```
writer = SummaryWriter()
criterion = nn.MSELoss()
```

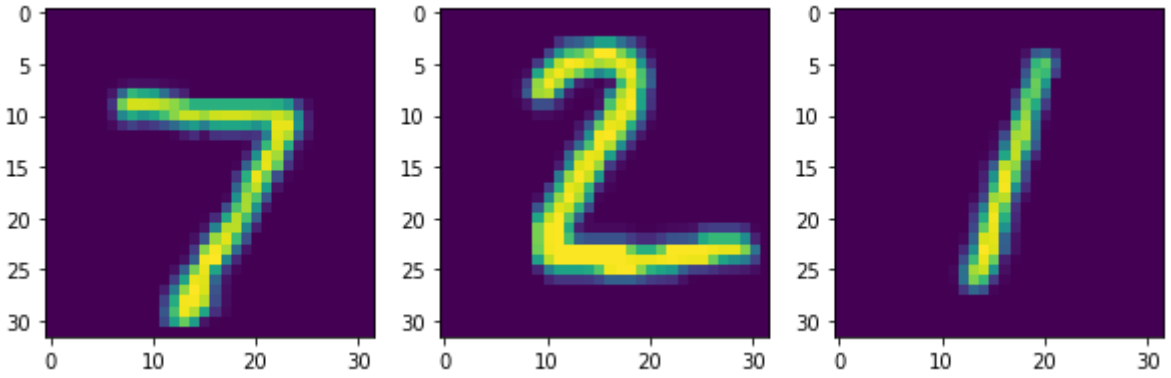
```
with torch.no_grad():
    step = 0
    for i, batch in enumerate(Test_loader):
        optimizer.zero_grad()
        yhat = netReg(batch[0])
        yhat = yhat.view(len(batch[0]), 3)
        output = torch.sum(yhat, dim=1)
        loss = criterion(output.float().squeeze(), batch[1].float().squeeze())
        rmse = torch.sqrt(torch.sum(loss)/len(batch[1]))
        writer.add_scalar('RMSEtest', rmse, global_step = step)
        step += 1
writer.flush()
```



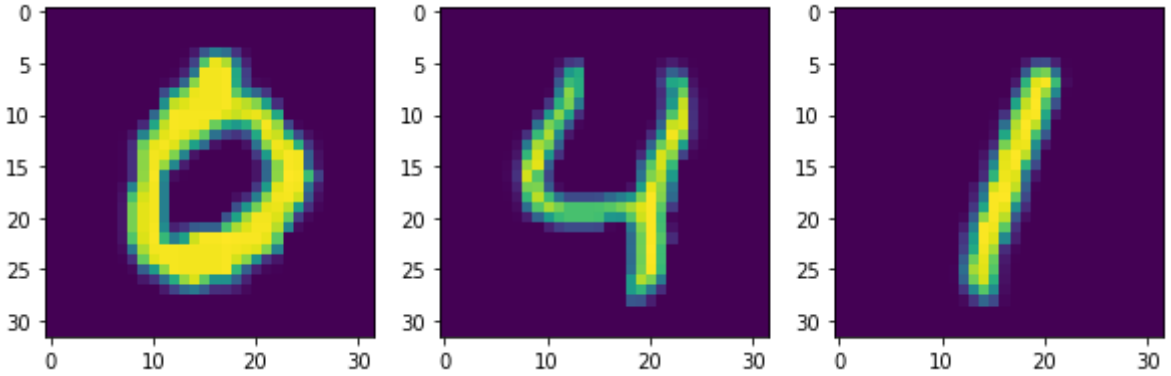
10 sets of test images (each set containing K many images) and the output from your trained network (final value that your model outputs)

In [744…

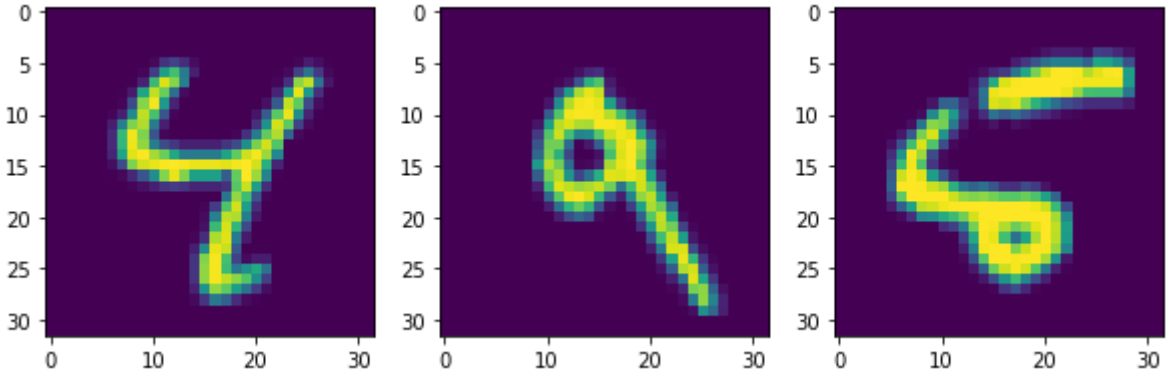
```
with torch.no_grad():
    for i in range(10):
        images, lable = test_dataset[i]
        fig = plt.figure(figsize=(10, 8))
        for i in range(len(images)):
            image = images[i]
            # for ploting changing position of channel : (h,w,c)
            tensor_image = image.view(image.shape[1], image.shape[2], image.shape[0])
            fig.add_subplot(1, len(images), i+1)
            plt.imshow(tensor_image)
        yhat = netReg(images.unsqueeze(0))
        output = torch.sum(yhat, dim=0)
        plt.show()

        print('Dataset label:',lable)
        print(output.item())
```



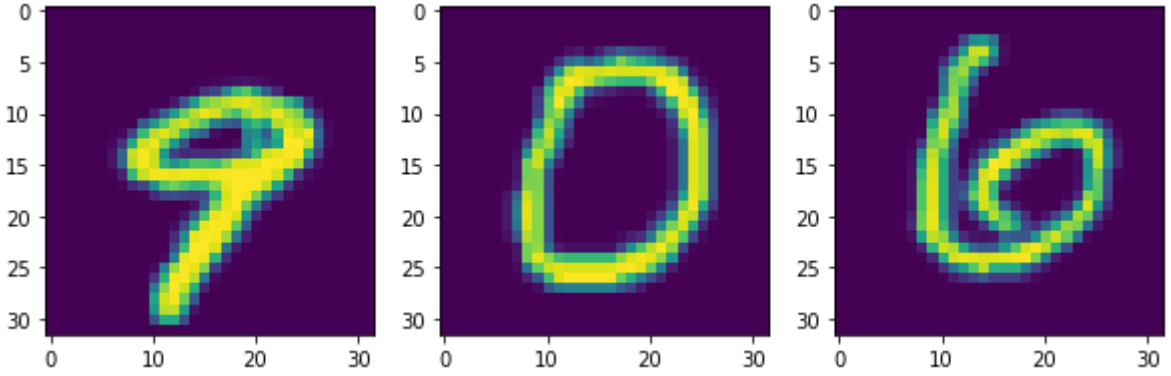
```
Dataset label: 10
10.759950637817383
```



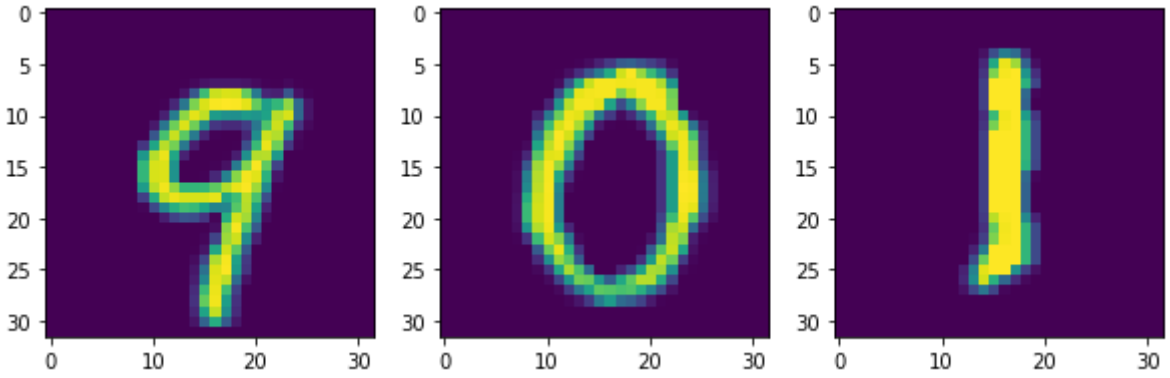
```
Dataset label: 5
6.164813995361328
```



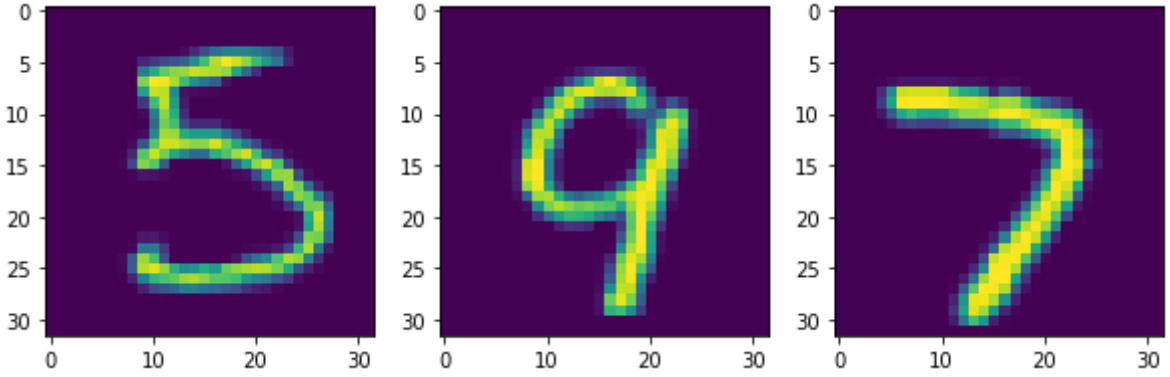
```
Dataset label: 18
16.521717071533203
```



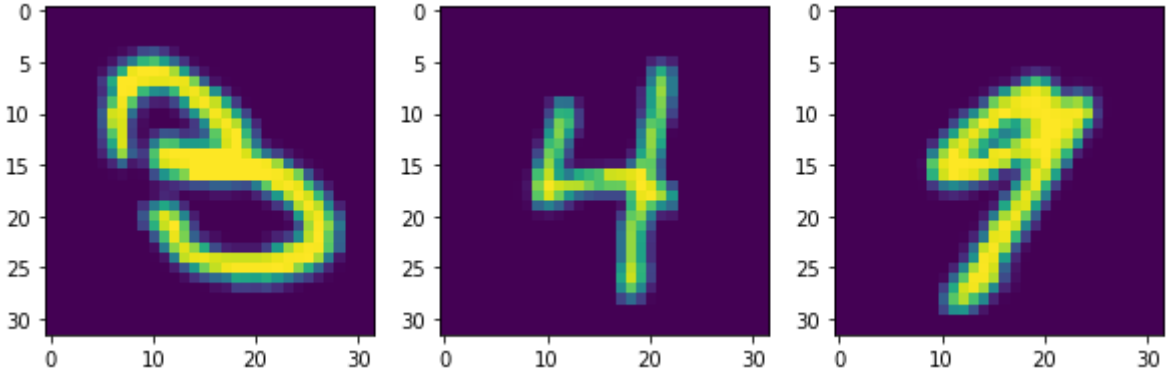
```
Dataset label: 15
15.444113731384277
```



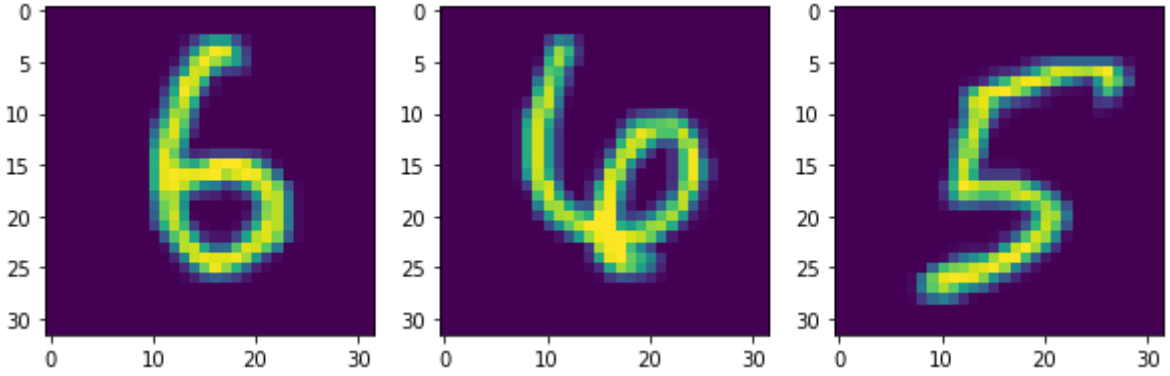
```
Dataset label: 10
10.443721771240234
```



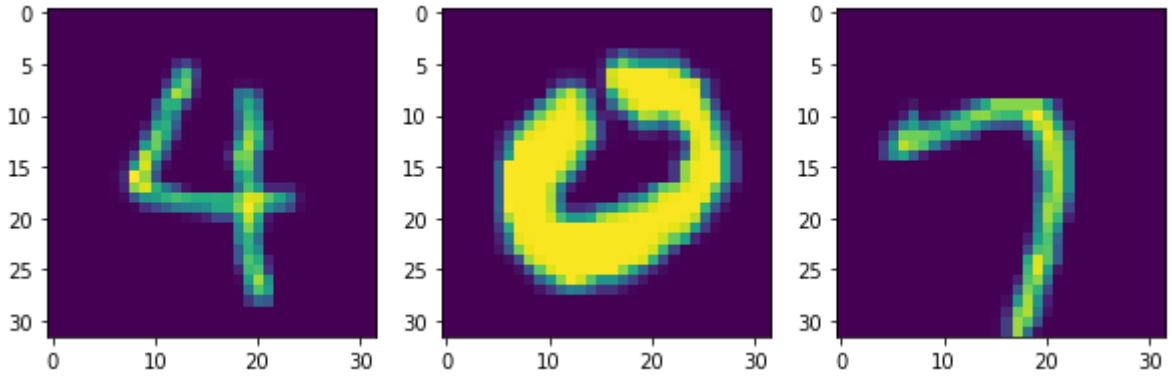
```
Dataset label: 21
21.81026268005371
```



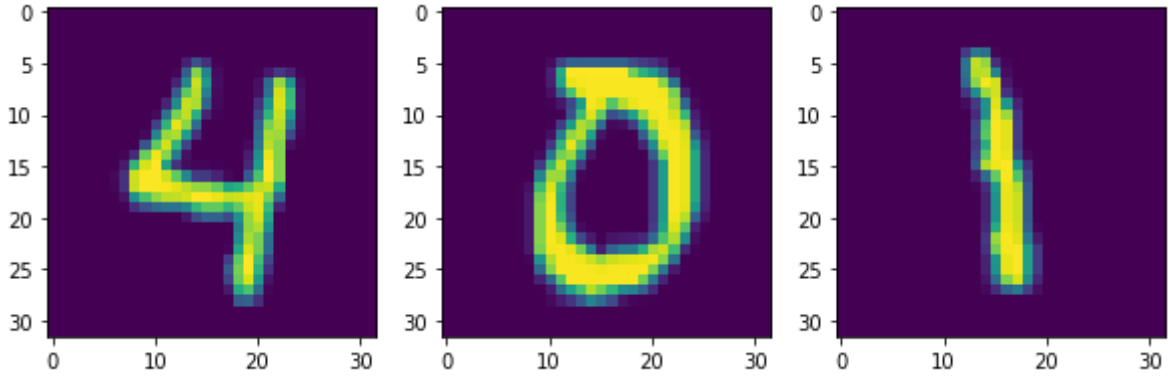
```
Dataset label: 16
18.712127685546875
```



```
Dataset label: 17
16.651506423950195
```



```
Dataset label: 11
12.887022018432617
```



```
Dataset label: 5
5.136078834533691
```