

Target:

```
-- TO introduce accuracy we want the model to train on more difficult data. So we are introducing image augmentation in this step
```

Results:

```
-- Parameters: 9,907
-- Best Training Accuracy: 97.92
-- Best Test Accuracy: 99.38
```

Analysis:

```
-- The accuracy has improved compared to previous step
-- The training accuracy is increasing but test accuracy is fluctuating a little. The model is not able to find the minu=ima. So, we
```

▼ Import libraries

```
from __future__ import print_function
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
```

▼ Data Transformations (without normalization)

```
# Train Phase transformations
train_transforms = transforms.Compose([
    transforms.RandomRotation((-7.0, 7.0), fill=(1,)),
    transforms.ToTensor()
])

# Test Phase transformations
test_transforms = transforms.Compose([
    transforms.ToTensor()
])
```

▼ Dataset and Creating Train/Test Split (without normalization)

```
train = datasets.MNIST('./data', train=True, download=True, transform=train_transforms)
test = datasets.MNIST('./data', train=False, download=True, transform=test_transforms)
```

Downloading <http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz>

▼ Dataloader Arguments & Test/Train Dataloaders (without normalization)

```
SEED = 1

# CUDA?
cuda = torch.cuda.is_available()
print("CUDA Available?", cuda)

# For reproducibility
torch.manual_seed(SEED)

if cuda:
    torch.cuda.manual_seed(SEED)

# dataloader arguments - something you'll fetch these from cmdprmt
dataloader_args = dict(shuffle=True, batch_size=128, num_workers=4, pin_memory=True) if cuda else dict(shuffle=True, batch

# train dataloader
train_loader = torch.utils.data.DataLoader(train, **dataloader_args)

# test dataloader
test_loader = torch.utils.data.DataLoader(test, **dataloader_args)

CUDA Available? True
/usr/local/lib/python3.8/dist-packages/torch/utils/data/dataloader.py:554: UserWarning: This DataLoader will create 4
warnings.warn(_create_warning_msg(
```

▼ Data Transformations (with normalization)

```
# Train Phase transformations
train_transforms = transforms.Compose([
    transforms.RandomRotation((-7.0, 7.0), fill=(1,)),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

# Test Phase transformations
test_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
```

▼ Dataset and Creating Train/Test Split (with normalization)

```
train = datasets.MNIST('./data', train=True, download=True, transform=train_transforms)
test = datasets.MNIST('./data', train=False, download=True, transform=test_transforms)
```

▼ Dataloader Arguments & Test/Train Dataloaders (with normalization)

```
SEED = 1

# CUDA?
cuda = torch.cuda.is_available()
print("CUDA Available?", cuda)

# For reproducibility
torch.manual_seed(SEED)

if cuda:
    torch.cuda.manual_seed(SEED)

# dataloader arguments - something you'll fetch these from cmdprmt
dataloader_args = dict(shuffle=True, batch_size=128, num_workers=4, pin_memory=True) if cuda else dict(shuffle=True, batch

# train dataloader
train_loader = torch.utils.data.DataLoader(train, **dataloader_args)
```

```
# test dataloader
test_loader = torch.utils.data.DataLoader(test, **dataloader_args)

CUDA Available? True
```

▼ plot some images to see which image augmentation to use (with normalization)

We will plot some images to see which image augmentation technique we can use

```
dataiter = iter(train_loader)
images, labels = next(dataiter)

# Let's visualize some of the images
%matplotlib inline
import matplotlib.pyplot as plt

figure = plt.figure()
num_of_images = 60
for index in range(1, num_of_images + 1):
    plt.subplot(6, 10, index)
    plt.axis('off')
    plt.imshow(images[index].numpy().squeeze(), cmap='gray_r')
```



Model

```
dropout_value = 0.05
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

    #input block
    self.convblock1 = nn.Sequential(nn.Conv2d(in_channels = 1, out_channels = 10, kernel_size = 3, padding = 1),
                                    nn.BatchNorm2d(10),
                                    nn.ReLU(),
                                    nn.Dropout(dropout_value)) #R_in = 1, C_in = 28, K = 3, P = 1, S = 1, J_in = 1, J_out

    #conv block 1
    self.convblock2 = nn.Sequential(nn.Conv2d(in_channels = 10, out_channels = 12, kernel_size = 3, padding = 1),
                                    nn.BatchNorm2d(12),
                                    nn.ReLU(),
                                    nn.Dropout(dropout_value)) #R_in = 3, C_in = 28, K = 3, P = 1, S = 1, J_in = 1, J_out

    #conv block 2
    self.convblock3 = nn.Sequential(nn.Conv2d(in_channels = 12, out_channels = 14, kernel_size = 3, padding = 1),
                                    nn.BatchNorm2d(14),
                                    nn.ReLU(),
                                    nn.Dropout(dropout_value)) #R_in = 5, C_in = 28, K = 3, P = 1, S = 1, J_in = 1, J_out

    #transition block1
    self.convblock4 = nn.Sequential(nn.Conv2d(in_channels = 14, out_channels = 16, kernel_size = 3, padding = 1),
                                    nn.BatchNorm2d(16),
                                    nn.ReLU(),
                                    nn.Dropout(dropout_value)) #R_in = 7, C_in = 28, K = 3, P = 1, S = 1, J_in = 1, J_out

    self.pool1 = nn.MaxPool2d(2, 2) #R_in = 9, C_in = 28, K = 2, P = 0, S = 2, J_in = 1, J_out = 2, R_out = R_in + (K-1)*J

    #conv block 3
    self.convblock5 = nn.Sequential(nn.Conv2d(in_channels = 16, out_channels = 12, kernel_size = 3, padding = 1),
                                    nn.BatchNorm2d(12),
                                    nn.ReLU(),
                                    nn.Dropout(dropout_value)) #R_in = 10, C_in = 14, K = 3, P = 1, S = 1, J_in = 2, J_out
```

```

#conv block 4
self.convblock6 = nn.Sequential(nn.Conv2d(in_channels = 12, out_channels = 12, kernel_size = 3, padding = 1),
                                nn.BatchNorm2d(12),
                                nn.ReLU(),
                                nn.Dropout(dropout_value)) #R_in = 14, C_in = 14, K = 3, P = 1, S = 1, J_in = 2, J_out

#conv block 4
self.convblock7 = nn.Sequential(nn.Conv2d(in_channels = 12, out_channels = 10, kernel_size = 3, padding = 1),
                                nn.BatchNorm2d(10),
                                nn.ReLU(),
                                nn.Dropout(dropout_value)) #R_in = 18, C_in = 14, K = 3, P = 1, S = 1, J_in = 2, J_out

#gap layer
self.gap = nn.Sequential(
    nn.AvgPool2d(kernel_size=4)) #R_in = 22, C_in = 14, K = 4, P = 1, S = 1, J_in = 2, J_out = 2, R_out = R_in +

#output block
self.convblock8 = nn.Sequential(nn.Conv2d(in_channels = 10, out_channels = 10, kernel_size = 3, padding = 0), nn.Dropout

def forward(self, x):
    x = self.convblock1(x)
    x = self.convblock2(x)
    x = self.convblock3(x)
    x = self.convblock4(x)
    x = self.pool1(x)
    x = self.convblock5(x)
    x = self.convblock6(x)
    x = self.convblock7(x)
    x = self.gap(x)
    x = self.convblock8(x)
    x = x.view(-1, 10)
    return F.log_softmax(x, dim=-1)

```

Model parameters

```

!pip install torchsummary
from torchsummary import summary

```

```

use_cuda = torch.cuda.is_available()
device = torch.device("cuda" if use_cuda else "cpu")

```

```

model = Net().to(device)
summary(model, input_size = (1, 28, 28))

```

Looking in indexes: <https://pypi.org/simple>, <https://us-python.pkg.dev/colab-wheels/public/simple/>
Requirement already satisfied: torchsummary in /usr/local/lib/python3.8/dist-packages (1.5.1)

Layer (type)	Output Shape	Param #
=====		
Conv2d-1	[-1, 10, 28, 28]	100
BatchNorm2d-2	[-1, 10, 28, 28]	20
ReLU-3	[-1, 10, 28, 28]	0
Dropout-4	[-1, 10, 28, 28]	0
Conv2d-5	[-1, 12, 28, 28]	1,092
BatchNorm2d-6	[-1, 12, 28, 28]	24
ReLU-7	[-1, 12, 28, 28]	0
Dropout-8	[-1, 12, 28, 28]	0
Conv2d-9	[-1, 14, 28, 28]	1,526
BatchNorm2d-10	[-1, 14, 28, 28]	28
ReLU-11	[-1, 14, 28, 28]	0
Dropout-12	[-1, 14, 28, 28]	0
Conv2d-13	[-1, 16, 28, 28]	2,032
BatchNorm2d-14	[-1, 16, 28, 28]	32
ReLU-15	[-1, 16, 28, 28]	0
Dropout-16	[-1, 16, 28, 28]	0
MaxPool2d-17	[-1, 16, 14, 14]	0
Conv2d-18	[-1, 12, 14, 14]	1,740
BatchNorm2d-19	[-1, 12, 14, 14]	24
ReLU-20	[-1, 12, 14, 14]	0
Dropout-21	[-1, 12, 14, 14]	0
Conv2d-22	[-1, 12, 14, 14]	1,308
BatchNorm2d-23	[-1, 12, 14, 14]	24
ReLU-24	[-1, 12, 14, 14]	0
Dropout-25	[-1, 12, 14, 14]	0
Conv2d-26	[-1, 10, 14, 14]	1,090
BatchNorm2d-27	[-1, 10, 14, 14]	20
ReLU-28	[-1, 10, 14, 14]	0
Dropout-29	[-1, 10, 14, 14]	0
AvgPool2d-30	[-1, 10, 3, 3]	0
Conv2d-31	[-1, 10, 1, 1]	910
Dropout-32	[-1, 10, 1, 1]	0
=====		

Total params: 9,970

```

Trainable params: 9,970
Non-trainable params: 0
-----
Input size (MB): 0.00
Forward/backward pass size (MB): 1.47
Params size (MB): 0.04
Estimated Total Size (MB): 1.51
-----

```

Training and Testing

```

from tqdm import tqdm

train_losses = []
test_losses = []
train_acc = []
test_acc = []

def train(model, device, train_loader, optimizer, epoch):
    model.train()
    pbar = tqdm(train_loader)
    correct = 0
    processed = 0
    for batch_idx, (data, target) in enumerate(pbar):
        # get samples
        data, target = data.to(device), target.to(device)

        # Init
        optimizer.zero_grad()
        # In PyTorch, we need to set the gradients to zero before starting to do backpropagation because PyTorch accumulates
        # Because of this, when you start your training loop, ideally you should zero out the gradients so that you do the par

        # Predict
        y_pred = model(data)

        # Calculate loss
        loss = F.nll_loss(y_pred, target)
        train_losses.append(loss)

        # Backpropagation
        loss.backward()
        optimizer.step()

        # Update pbar-tqdm

        pred = y_pred.argmax(dim=1, keepdim=True) # get the index of the max log-probability
        correct += pred.eq(target.view_as(pred)).sum().item()
        processed += len(data)

    pbar.set_description(desc= f'Loss={loss.item()} Batch_id={batch_idx} Accuracy={100*correct/processed:0.2f}%')
    train_acc.append(100*correct/processed)

def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item() # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True) # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    test_losses.append(test_loss)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%) \n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))

    test_acc.append(100. * correct / len(test_loader.dataset))

from torch.optim.lr_scheduler import StepLR

model = Net().to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
# scheduler = StepLR(optimizer, step_size=6, gamma=0.1)

```

```

EPOCHS = 15
for epoch in range(EPOCHS):
    print("EPOCH:", epoch)
    train(model, device, train_loader, optimizer, epoch)
    # scheduler.step()
    test(model, device, test_loader)

    EPOCH: 0
    Loss=0.11991719156503677 Batch_id=468 Accuracy=87.59: 100%|██████████| 469/469 [00:21<00:00, 22.22it/s]

    Test set: Average loss: 0.0686, Accuracy: 9798/10000 (97.98%)

    EPOCH: 1
    Loss=0.05137534439563751 Batch_id=468 Accuracy=96.21: 100%|██████████| 469/469 [00:24<00:00, 19.32it/s]

    Test set: Average loss: 0.0550, Accuracy: 9827/10000 (98.27%)

    EPOCH: 2
    Loss=0.033745329827070236 Batch_id=468 Accuracy=96.75: 100%|██████████| 469/469 [00:18<00:00, 25.32it/s]

    Test set: Average loss: 0.0555, Accuracy: 9812/10000 (98.12%)

    EPOCH: 3
    Loss=0.0538853220641613 Batch_id=468 Accuracy=97.10: 100%|██████████| 469/469 [00:18<00:00, 24.84it/s]

    Test set: Average loss: 0.0295, Accuracy: 9900/10000 (99.00%)

    EPOCH: 4
    Loss=0.05264396592974663 Batch_id=468 Accuracy=97.30: 100%|██████████| 469/469 [00:20<00:00, 23.22it/s]

    Test set: Average loss: 0.0418, Accuracy: 9863/10000 (98.63%)

    EPOCH: 5
    Loss=0.058095116168260574 Batch_id=468 Accuracy=97.47: 100%|██████████| 469/469 [00:19<00:00, 24.25it/s]

    Test set: Average loss: 0.0270, Accuracy: 9906/10000 (99.06%)

    EPOCH: 6
    Loss=0.08282031863927841 Batch_id=468 Accuracy=97.47: 100%|██████████| 469/469 [00:19<00:00, 24.14it/s]

    Test set: Average loss: 0.0243, Accuracy: 9926/10000 (99.26%)

    EPOCH: 7
    Loss=0.04594587907195091 Batch_id=468 Accuracy=97.60: 100%|██████████| 469/469 [00:18<00:00, 25.28it/s]

    Test set: Average loss: 0.0236, Accuracy: 9918/10000 (99.18%)

    EPOCH: 8
    Loss=0.03975338116288185 Batch_id=468 Accuracy=97.78: 100%|██████████| 469/469 [00:18<00:00, 25.10it/s]

    Test set: Average loss: 0.0231, Accuracy: 9915/10000 (99.15%)

    EPOCH: 9
    Loss=0.07474153488874435 Batch_id=468 Accuracy=97.76: 100%|██████████| 469/469 [00:18<00:00, 25.05it/s]

    Test set: Average loss: 0.0217, Accuracy: 9921/10000 (99.21%)

    EPOCH: 10
    Loss=0.031006107106804848 Batch_id=468 Accuracy=97.71: 100%|██████████| 469/469 [00:18<00:00, 25.06it/s]

    Test set: Average loss: 0.0216, Accuracy: 9918/10000 (99.18%)

    EPOCH: 11
    Loss=0.09529513120651245 Batch_id=468 Accuracy=97.77: 100%|██████████| 469/469 [00:19<00:00, 23.69it/s]

train_losses = [i.item() for i in train_losses]

%matplotlib inline
import matplotlib.pyplot as plt
fig, axs = plt.subplots(2,2,figsize=(15,10))
axs[0, 0].plot(train_losses)
axs[0, 0].set_title("Training Loss")
axs[1, 0].plot(train_acc[4000:])
axs[1, 0].set_title("Training Accuracy")
axs[0, 1].plot(test_losses)
axs[0, 1].set_title("Test Loss")
axs[1, 1].plot(test_acc)
axs[1, 1].set_title("Test Accuracy")

```

Text(0.5, 1.0, 'Test Accuracy')

