# Project 8 _ MNIST Handwritten Digit Regcognition using Neural Network

Ishita Ghosh (ighosh4)

**Part 1. (Problem 17.1)**

All implementations were in PyTorch 1.5.0.. I downloaded tutorial code from the following website:
MNIST Handwritten Digit Recognition in PyTorch

It is a very simple architecture containing two convolutional layers followed by two feedforward layers, so it serves as a good baseline model. I have trained it using stochastic gradient descent for 3 epochs with the following hyper-parameters:

1. Batch_size_train (Minibatch size) = 64
2. learning_rate = 0.01
3. momentum = 0.5

The source code is in the Appendix.

With this baseline model I have achieved a relatively good test accuracy of 96.54%.

**Part 2 (Problem 17.2)**

As instructed, I have implemented the architecture from Sect. 17.2.1 in the textbook.Used the following hyperparameters:

1. Batch_size_train (Minibatch size) = 64
2. learning_rate = 0.01
3. momentum = 0.5
4. Dropout = 0.5 and 0.1

Performed experiments where we trained multiple variations of the model and evaluated it on the test set. The relative performances are given in the table below:

| **Momentum (Yes/No)** | **Dropout (0.5/0.1/No)** | **Accuracy (%)** |
|---|---|---|
| No | No | 97.93 |
| Yes | No | **98.42** |

| | | |
|---|---|---|
| No | 0.1 | 97.65 |
| No | 0.5 | 96.10 |
| Yes | 0.1 | 97.94 |
| Yes | 0.5 | 97.42 |

It is seen that adding momentum improves the accuracy. But adding dropout does not always make the model better. With zero momentum, adding dropout increases accuracy slightly from 97.93 to 97.94. But with momentum added, dropout actually decreases performance.

Adding Improvements:

To further improve the model, I added a batch-normalization layer after every convolutional layer. The result of adding these layers is given below:

| **Dropout (Yes/No)** | **Accuracy (%)** |
|---|---|
| No | 98.76 |
| Yes | **98.94** |

Used momentum=0.5 for these experiments. Since it wasn't clear whether adding dropout necessarily improves performance or not, I tried with two settings, one with no dropout and one where dropout=0.1. I observed that for this improved model adding dropout actually does improve performance. With batch-norm, momentum and dropout we get **98.94%** accuracy which is the best performance out of all the models we tested.

## Appendix:

### Part 1 (Problem 17.1): Source Code

```python
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt


# Source code from https://nextjournal.com/gkoehler/pytorch-mnist
class TutorialNet(nn.Module):
    def __init__(self):
        super(TutorialNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x, 1)

def train(epoch):
    network.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = network(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
            epoch, batch_idx * len(data), len(train_loader.dataset),
            100. * batch_idx / len(train_loader), loss.item()))
            train_losses.append(loss.item())
            train_counter.append(
            (batch_idx*64) + ((epoch-1)*len(train_loader.dataset)))
            torch.save(network.state_dict(), 'results/model_17_1.pth')
            torch.save(optimizer.state_dict(),
'results/optimizer_17_1.pth')


def test():
    network.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            output = network(data)
            test_loss += F.nll_loss(output, target, reduction="sum").item()
            pred = output.data.max(1, keepdim=True)[1]
            correct += pred.eq(target.data.view_as(pred)).sum()
        test_loss /= len(test_loader.dataset)
        test_losses.append(test_loss)
        print('\nTest set: Avg. loss: {:.4f}, Accuracy: {}/{}
({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))


if __name__ =="__main__":
    n_epochs = 3
    batch_size_train = 64
    batch_size_test = 1000
    learning_rate = 0.01
    momentum = 0.5
    log_interval = 10

    random_seed = 1
    torch.backends.cudnn.enabled = False
    torch.manual_seed(random_seed)

    train_loader = torch.utils.data.DataLoader(
      torchvision.datasets.MNIST('data/', train=True, download=True,
                                 transform=torchvision.transforms.Compose([
                                   torchvision.transforms.ToTensor(),
                                   torchvision.transforms.Normalize(
                                     (0.1307,), (0.3081,))
                                 ])),
      batch_size=batch_size_train, shuffle=True)

    test_loader = torch.utils.data.DataLoader(
      torchvision.datasets.MNIST('data/', train=False, download=True,
                                 transform=torchvision.transforms.Compose([
                                   torchvision.transforms.ToTensor(),
                                   torchvision.transforms.Normalize(
                                     (0.1307,), (0.3081,))
                                 ])),
                                 batch_size=batch_size_test, shuffle=True)
    network = TutorialNet()
    optimizer = optim.SGD(network.parameters(), lr=learning_rate,
                          momentum=momentum)
    train_losses = []
    train_counter = []
    test_losses = []
    test_counter = [i*len(train_loader.dataset) for i in range(n_epochs +
1)]

    test()
    for epoch in range(1, n_epochs + 1):
        train(epoch)
        test()

    fig = plt.figure()
    plt.plot(train_counter, train_losses, color='blue')
    plt.scatter(test_counter, test_losses, color='red')
    plt.legend(['Train Loss', 'Test Loss'], loc='upper right')
    plt.xlabel('number of training examples seen')
    plt.ylabel('negative log likelihood loss')
    plt.savefig("performance_17_1.png")
```

**Part 2 (Problem 17.2) Source Code:**

```
import matplotlib.pyplot as plt
import numpy as np
```

```python
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import argparse
from tqdm import tqdm, trange


class Net(nn.Module):
    def __init__(self, args):
        super(Net, self).__init__()
        self.use_dropout = args.use_dropout
        self.use_improved = args.use_improved
        self.dropout_p = args.dropout_p

        if self.use_improved:
            self.conv1 = self.conv_bn(in_channels=1, out_channels=20,
                                      kernel_size=5, stride=1, padding=0,
                                      dilation=1, groups=1, bias=True)
            self.conv2 = self.conv_bn(in_channels=20, out_channels=50,
                                      kernel_size=5, stride=1, padding=0,
                                      dilation=1, groups=1, bias=True)
            self.conv3 = self.conv_bn(in_channels=50, out_channels=500,
                                      kernel_size=4, stride=1, padding=0,
                                      dilation=1, groups=1, bias=True)
            self.conv4 = self.conv_bn(in_channels=500, out_channels=10,
                                      kernel_size=1, stride=1, padding=0,
                                      dilation=1, groups=1, bias=True)
        else:
            self.conv1 = nn.Conv2d(in_channels=1, out_channels=20,
                                   kernel_size=5, stride=1, padding=0,
                                   dilation=1, groups=1, bias=True)
            self.conv2 = nn.Conv2d(in_channels=20, out_channels=50,
                                   kernel_size=5, stride=1, padding=0,
                                   dilation=1, groups=1, bias=True)
            self.conv3 = nn.Conv2d(in_channels=50, out_channels=500,
                                   kernel_size=4, stride=1, padding=0,
                                   dilation=1, groups=1, bias=True)
            self.conv4 = nn.Conv2d(in_channels=500, out_channels=10,
                                   kernel_size=1, stride=1, padding=0,
                                   dilation=1, groups=1, bias=True)
        self.maxpool1 = nn.MaxPool2d(2)
        if self.use_dropout:
            self.conv1 = nn.Sequential(self.conv1,
nn.Dropout2d(self.dropout_p))
            self.conv2 = nn.Sequential(self.conv2,
nn.Dropout2d(self.dropout_p))
        self.maxpool2 = nn.MaxPool2d(2)


    def conv_bn(self, in_channels, out_channels, kernel_size, stride,
                padding, dilation, groups, bias):
        return nn.Sequential(nn.Conv2d(in_channels, out_channels,
kernel_size, stride,
                padding, dilation, groups, bias),
nn.BatchNorm2d(out_channels))


    def forward(self, x):
        x = self.conv1(x)
        x = self.maxpool1(x)
        x = self.conv2(x)
        x = self.maxpool2(x)
        x = self.conv3(x)
        x = F.relu(x)
        x = self.conv4(x)
        x = torch.squeeze(x, 2)
        x = torch.squeeze(x, 2)
        x = F.log_softmax(x, 1)
        return x

def train(epoch):
    network.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = network(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
            epoch, batch_idx * len(data), len(train_loader.dataset),
            100. * batch_idx / len(train_loader), loss.item()))
            train_losses.append(loss.item())
            train_counter.append(
            (batch_idx*64) + ((epoch-1)*len(train_loader.dataset)))
            torch.save(network.state_dict(), 'results/model_17_2.pth')
            torch.save(optimizer.state_dict(),
'results/optimizer_17_2.pth')


def test():
    network.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            output = network(data)
            test_loss += F.nll_loss(output, target, reduction="sum").item()
            pred = output.data.max(1, keepdim=True)[1]
            correct += pred.eq(target.data.view_as(pred)).sum()
        test_loss /= len(test_loader.dataset)
        test_losses.append(test_loss)
        print('\nTest set: Avg. loss: {:.4f}, Accuracy: {}/{}
({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))


def get_image(label):
    for batch_idx, (image, target) in enumerate(adv_loader):
        if label == target:
            return batch_idx, image
    return None, None


def save_image(img, label):
    fname = f"adversarial_images/{label}.png"
    plt.imsave(open(fname, "wb"), img.reshape(28,28),
cmap=plt.get_cmap('gray'), format="png")


def adversarial_example(x_target, label, eta=0.1, lam=0.05, steps=10000):
    # Create a random image to initialize gradient descent with
```

```python
    x = torch.tensor(np.random.normal(.5, .3, (1, 1, 28, 28)),
dtype=torch.float, requires_grad=True)
    # x = x_target
    # x = x.type(torch.FloatTensor)
    label = torch.tensor(label)
    label = torch.unsqueeze(label, 0)
    optimizer = optim.SGD(network.parameters(), lr=learning_rate,
                          momentum=momentum)
    # Gradient descent on the input
    network.eval()
    for i in trange(steps):
        # Calculate the derivative
        optimizer.zero_grad()
        if x.grad is not None:
            x.grad.zero_()
        output = network(x)
        loss = F.nll_loss(output, label)
        x.retain_grad()
        loss.backward(retain_graph=True)
        tqdm.write(f"Loss = {loss}")
        d = x.grad
        # The GD update on x, with an added penalty
        # to the cost function
        # ONLY CHANGE IS RIGHT HERE!!!
        if d is not None:
            x = x - eta * (d + lam * (x - x_target))
    return x


def generate_adversarials():
    adversarial_dict = {}
    for i in range(10):
        target_idx, target_image = get_image(i)
        if i==9:
            label = 0
        else:
            label = i + 1
        print(f"Generating Adversarial Example for label = {label}")
        adversarial_image = adversarial_example(target_image, label)
        adversarial_dict[i] = adversarial_image
    return adversarial_dict
```

```
if __name__ =="__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-m", "--use_momentum", help="Whether to use
momentum in optimizer",
                        action="store_true")
    parser.add_argument("-d", "--use_dropout", help="Whether to use dropout
in model",
                        action="store_true")
    parser.add_argument("-i", "--use_improved", help="Whether to use
improved network architecture",
                        action="store_true")
    parser.add_argument("-e", "--n_epochs", help="Number of epochs",
                        type=int, default=3)
    parser.add_argument("-p", "--dropout_p", help="Dropout Probability if
using dropout",
                        type=float, default=0.5)
    parser.add_argument("-l", "--lr", help="Learning Rate",
                        type=float, default=0.01)
    parser.add_argument("-t", "--train_model", help="Whether to train new
model from scratch",
                        action="store_true")
    parser.add_argument("-a", "--adversarial", help="Generate Adversarial
Examples",
                        action="store_true")
    args = parser.parse_args()
    n_epochs = args.n_epochs
    batch_size_train = 64
    batch_size_test = 1000
    learning_rate = args.lr
    if args.use_momentum:
        momentum = 0.5
    else:
        momentum = 0.0
    log_interval = 10

    random_seed = 1
    torch.backends.cudnn.enabled = False
    torch.manual_seed(random_seed)

    train_loader = torch.utils.data.DataLoader(
      torchvision.datasets.MNIST('data/', train=True, download=True,
                                 transform=torchvision.transforms.Compose([
                                   torchvision.transforms.ToTensor(),
                                   torchvision.transforms.Normalize(
                                     (0.1307,), (0.3081,))
                                 ])),
      batch_size=batch_size_train, shuffle=True)

    test_loader = torch.utils.data.DataLoader(
      torchvision.datasets.MNIST('data/', train=False, download=True,
                                 transform=torchvision.transforms.Compose([
                                   torchvision.transforms.ToTensor(),
                                   torchvision.transforms.Normalize(
                                     (0.1307,), (0.3081,))
                                 ])),
                                 batch_size=batch_size_test, shuffle=True)

    adv_loader = torch.utils.data.DataLoader(
      torchvision.datasets.MNIST('data/', train=False, download=True,
                                 transform=torchvision.transforms.Compose([
                                   torchvision.transforms.ToTensor(),
                                   torchvision.transforms.Normalize(
                                     (0.1307,), (0.3081,))
                                 ])),
                                 batch_size=1, shuffle=False)
    network = Net(args)
    test_losses = []
    test_counter = [i*len(train_loader.dataset) for i in range(n_epochs +
1)]
    if args.train_model:
        optimizer = optim.SGD(network.parameters(), lr=learning_rate,
                              momentum=momentum)
        train_losses = []
        train_counter = []
        test()
        for epoch in range(1, n_epochs + 1):
            train(epoch)
            test()
    else:
        network.load_state_dict(torch.load('results/model_17_2.pth'))
        test()

    print(f"Momentum: {args.use_momentum}.\nDropout: {args.use_dropout},
{args.dropout_p}\nImproved Model: {args.use_improved}")
    if args.adversarial:
        adversarial_dict = generate_adversarials()
        for i, x in adversarial_dict.items():
            output = network(x)
            pred = output.data.max(1, keepdim=True)[1]
            save_image(x[0], pred)
            print(f"Correct Label = {i}. Predicted Label = {pred}")
```