ÖZYEĞİN UNIVERSITY
FACULTY OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE

CS 454

2021 Fall

Homework4

Deep Neural Networks

By
Tuğcan Hoşer
S015561

Supervised By
Ethem Alpaydın

PART A

1) Intro

First of all, the data in the training.csv file and testing.csv in the project was separated according to their classes. There are 10 classes and each class has 100 data. Also used pytorch library.

2) HyperParameters

```
# torch parameters
SEED = 60             # reproducability
# NN Parameters
EPOCHS = 200          # number of epochs
LR = 0.01             # learning rate
MOMENTUM = 0.9        # momentum for the SGD optimizer (how much of the past gradients)
GAMMA = 0.1           # learning rate scheduler (how much to decrease learning rate)
BATCH_SIZE = 64       # number of images to load per iteration
d = 100               # number of input features
K = 10                # number of output features
H = None              # H=None for SLP else MLP
```

3) Networks

Three different networks were created. Only CNN and fully connected layer were used. In addition, parameters with different values were used.

a) Network1

CNN1 -> CNN2 -> FC1

CNN1 = In channel = 1

Out channel = 8

Kernel Size = 4

Stride = 2

Padding = 1

After convolution we have 5 x 5 feature maps this formula is

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1$$

n_{in}: number of input features
n_{out}: number of output features
k: convolution kernel size
p: convolution padding size
s: convolution stride size

$(10 – 4 + 2) / 2 + 1 = 5$

CNN2 = In channel = 8

Out channel = 16

Kernel Size = 4

Stride = 2

Padding = 1

After convolution we have 2 x 2 feature maps (5- 4 + 2) / 2 + 1 = 2

Fully connected layer from 16 * 2 * 2 = 64 input features to 10 hidden units

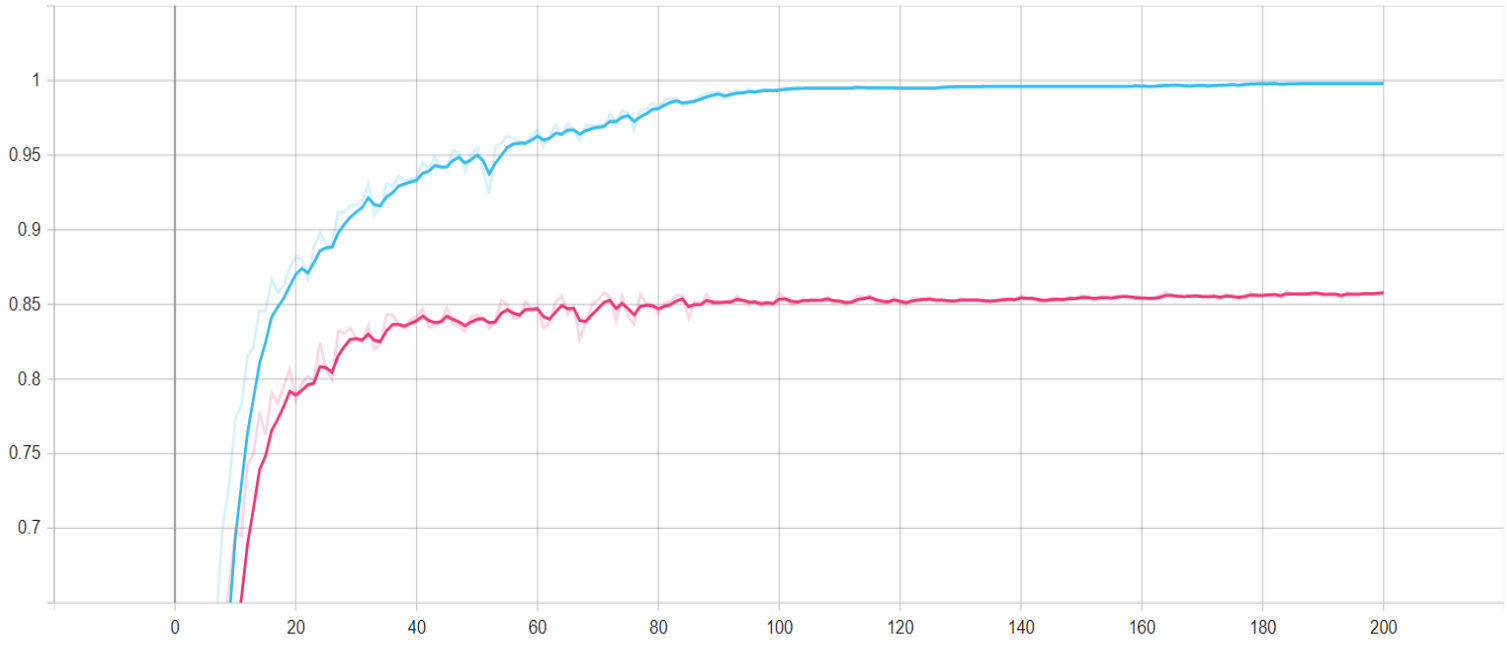
FC1 = In features = 16 * 2 * 2

Out features = 10

We have 10 class therefore, Out features must be 10.

Accuracy Graph

X axis is Epochs
Y axis is Accuracy
Blue represents Training accuracy
Red represents Test accuracy



b) Network2

CNN1 -> CNN2 -> FC1 -> FC2

CNN1 = In channel = 1

Out channel = 32

Kernel Size = 3

Stride = 2

Padding = 1

After convolution we have 5 x 5 feature maps (10 - 4 + 2) / 2 + 1 = 5

CNN2 = In channel = 32

Out channel = 64

Kernel Size = 3

Stride = 2

Padding = 1

After convolution we have 3 x 3 feature maps (5 – 3 + 2) / 2 + 1 = 3

First Fully connected layer from 64 * 3 * 3 = 576 input features to 144 hidden units

FC1 = In features = 64 * 3 * 3

Out features = 144

Second Fully connected layer from 144 input features to 10 hidden units

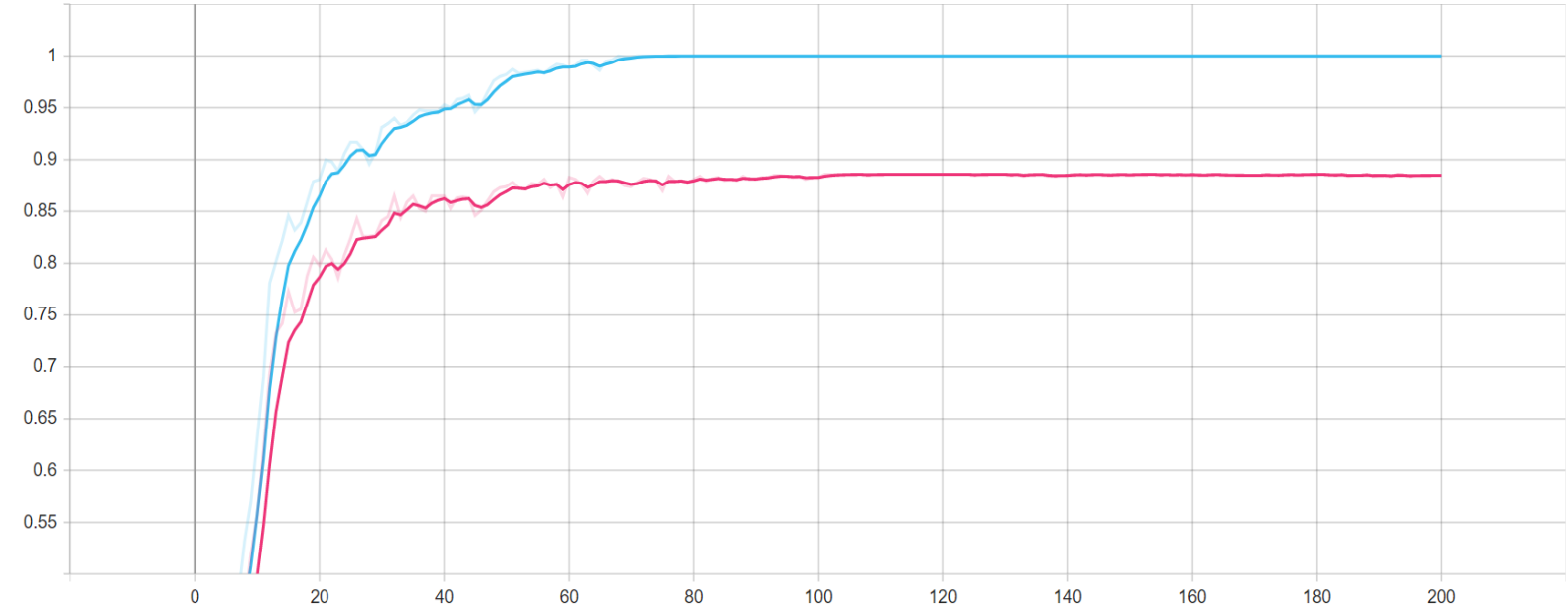
FC2 = In features = 144

Out features = 10

We have 10 class therefore, Out features must be 10.

Accuracy Graph

X axis Epochs
Y axis Accuracy
Blue represents Training accuracy
Red represents Test accuracy



c) Network3

CNN1 -> FC1 -> FC2

CNN1 = In channel = 1

Out channel = 32

Kernel Size = 3

Stride = 2

Padding = 1

After convolution we have 5 x 5 feature maps (10 - 4 + 2) / 2 + 1 = 5

First Fully connected layer from 32 * 5 * 5 = 800 input features to 100 hidden units

FC1 = In features = 32 * 5 * 5
Out features = 100

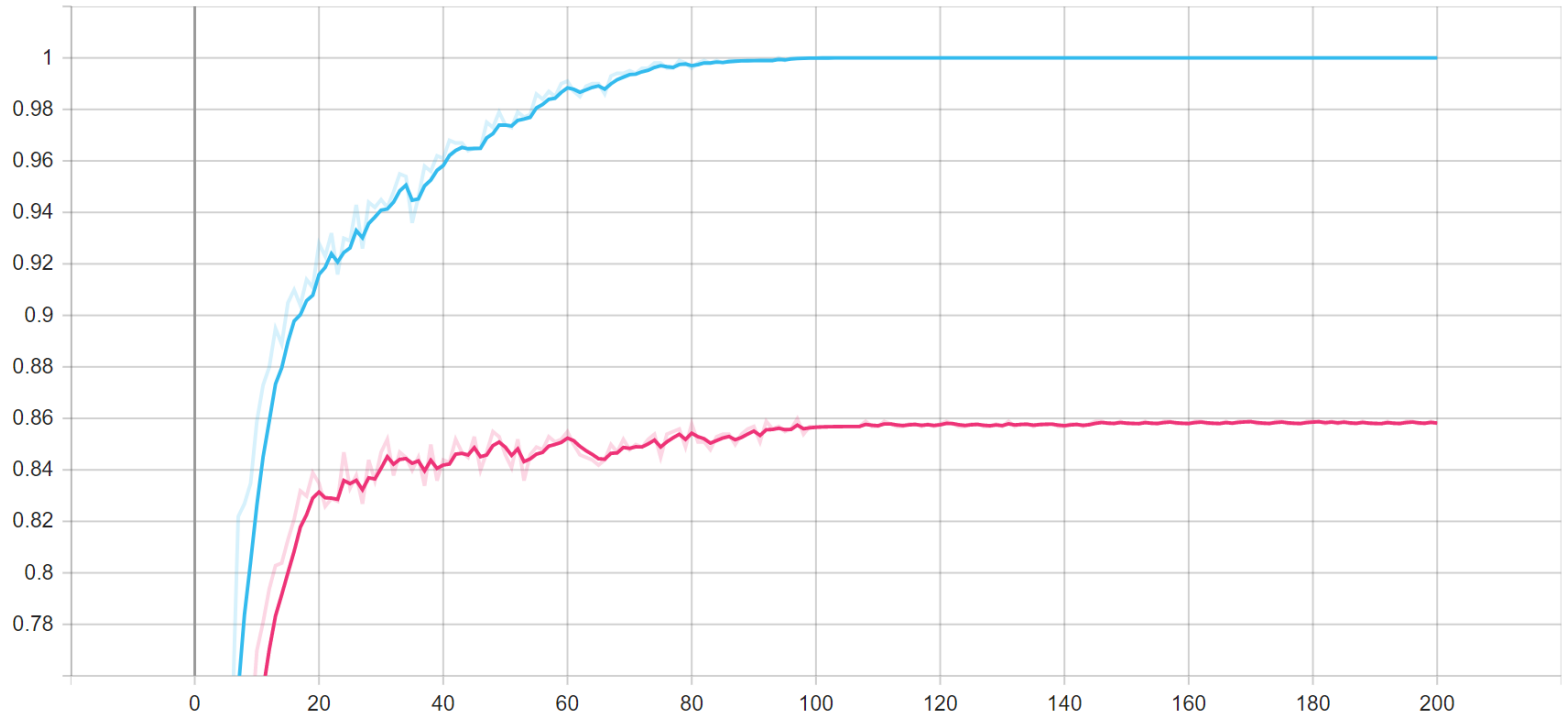
Second Fully connected layer from 100 input features to 10 hidden units

FC2 = In features = 100
Out features = 10

We have 10 class therefore, Out features must be 10.

Accuracy Graph

X axis Epochs
Y axis Accuracy
Blue represents Training accuracy
Red represents Test accuracy



CODING PART

```
import torch
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import torch.nn as nn  # neural network module
import torch.nn.functional as F
import torch.optim as optim  # optimization module
import torch.optim.lr_scheduler as lr_scheduler
from torch.utils.data import Dataset

from torch.utils.tensorboard import SummaryWriter  # logging module
from torchvision.utils import make_grid, save_image
import numpy as np

print(torch.__version__)


## Custom Dataset loader

class CustomDataset(Dataset):
    """ Custom dataset for flattened 10x10 csv dataset """

    # Initialize data
    def __init__(self, fname, transform=None):
        self.xy = np.genfromtxt(fname, delimiter=',', skip_header=1,
dtype=np.uint8)
        self.transform = transform

    def __getitem__(self, index):
        x = self.xy[index, 1:].reshape(10, 10, 1)  # H W C
        y = self.xy[index, 0]
        y = torch.as_tensor(y, dtype=torch.long)
        if self.transform:
            x = self.transform(x)
        return x, y

    def __len__(self):
        return self.xy.shape[0]


## Parameters and Hyperparameters

# torch parameters
SEED = 60  # reproducability
# NN Parameters
EPOCHS = 200  # number of epochs
LR = 0.01  # learning rate
MOMENTUM = 0.9  # momentum for the SGD optimizer (how much of the past
gradients)
GAMMA = 0.1  # learning rate scheduler (how much to decrease learning rate)
BATCH_SIZE = 64  # number of images to load per iteration
d = 100  # number of input features
K = 10  # number of output features
H = None  # H=None for SLP else MLP
```

```
# transform input data type from ndarray to tensor values between 0,1
transform = transforms.Compose([
    transforms.ToTensor(),
])

# read the datasets
tr_dataset = CustomDataset('training.csv', transform=transform)
# prepare loader for the training dataset
train_loader = torch.utils.data.DataLoader(tr_dataset,
batch_size=BATCH_SIZE, shuffle=True)

test_dataset = CustomDataset('testing.csv', transform=transform)
test_loader = torch.utils.data.DataLoader(test_dataset,
batch_size=BATCH_SIZE, shuffle=False)

class Network1(nn.Module):
    def __init__(self):
        super(Network1, self).__init__()
        # MNIST image is 1x10x10 (CxHxW)
        # Pytorch convolution expects input data in the form BxCxHxW
        # B: Batch size
        # C: number of channels gray scale images have 1 channel
        # W: width of the image
        # H: height of the image

        # use 8 4x4 filters with padding
        # padding is set to 1 so that image W,H is not changed after

          convolution
        # stride is 2 so filters will move 2 pixels for next calculation
        # W after conv2d  [(W - Kernelw + 2*padding)/stride] + 1
        # after convolution we'll have Bx8 5x5 feature maps

          (10-4+2)/2 + 1 = 5
        self.conv1 = nn.Conv2d(in_channels=1,  # 1 channel because gray

                                                scaled image
                               out_channels=8, # apply 32 filters and get a

                                                feature map for each filter
                               kernel_size=4,  # filters are 4x4 weights
                               stride=2,  # halves the size of the image
                               padding=1)

        # after convolution we'll have Bx16 2x2

           feature maps (5-4+2)/2 + 1 = 2
        self.conv2 = nn.Conv2d(in_channels=8,
                               out_channels=16,
                               kernel_size=4,
                               stride=2,
                               padding=1)

        # first fully connected layer from 16*2*2=64 input features

           to 10 hidden units
        self.fc1 = nn.Linear(in_features=16 * 2 * 2,
                             out_features=10)
```

```
    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = torch.flatten(x, start_dim=1)  # flatten feature maps,

                                           Bx(C*H*W)
        x = self.fc1(x)
        return x


class Network2(nn.Module):
    def __init__(self):
        super(Network2, self).__init__()

        # use 32 3x3 filters with padding
        # padding is set to 1 so that image
          W,H is not changed after convolution
        # stride is 2 so filters will move 2 pixels for next calculation
        # W after conv2d  [(W - Kernelw + 2*padding)/stride] + 1
        # after convolution we'll have Bx32 5x5
          feature maps (10-3+2)/2 + 1 = 5
        self.conv1 = nn.Conv2d(in_channels=1,  # 1 channel because gray
                                               scaled image
                               out_channels=32,  # apply 32 filters and
                                       get a feature map for each filter
                               kernel_size=3,  # filters are 3x3 weights
                               stride=2,  # halves the size of the image
                               padding=1)

        # after convolution we'll have Bx64 3x3
          feature maps (5-3+2)/2 + 1 = 3
        self.conv2 = nn.Conv2d(in_channels=32,
                               out_channels=64,
                               kernel_size=3,
                               stride=2,
                               padding=1)

        # first fully connected layer from 64*3*3 = 576
          input features to 144 hidden units
        self.fc1 = nn.Linear(in_features=64 * 3 * 3,
                             out_features=144)

        self.fc2 = nn.Linear(in_features=144,
                             out_features=10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = torch.flatten(x, start_dim=1)  # flatten feature maps,Bx(C*H*W)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```
class Network3(nn.Module):
    def __init__(self):
        super(Network3, self).__init__()

        # use 32 3x3 filters with padding
        # padding is set to 1 so that image W,H is not
          changed after convolution
        # stride is 2 so filters will move 2 pixels for next calculation
        # W after conv2d  [(W - Kernelw + 2*padding)/stride] + 1
        # after convolution we'll have Bx32 5x5
          feature maps (10-3+2)/2 + 1 = 5
        self.conv1 = nn.Conv2d(in_channels=1,  # 1 channel because
                                               gray scaled image
                               out_channels=32,  # apply 32 filters and get
                                                 a feature map for each filter
                               kernel_size=3,  # filters are 3x3 weights
                               stride=2,  # halves the size of the image
                               padding=1)

        # first fully connected layer from 32*5*5=800
          input features to 100 hidden units
        self.fc1 = nn.Linear(in_features=32 * 5 * 5,
                             out_features=100)

        # second fully connected layer from 100
          input features to 10 hidden units
        self.fc2 = nn.Linear(in_features=100,
                             out_features=10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = torch.flatten(x, start_dim=1)  # flatten feature maps,Bx(C*H*W)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

    # create the network


# check if CUDA is available
cuda = torch.cuda.is_available()
device = torch.device("cuda:0" if cuda else "cpu")

# if cuda is available move the network to gpu
net = Network1().to(device)
##net = Network2().to(device)
##net = Network3().to(device)


## Specify the loss function and the optimizer

# specify the loss to be used
# softmax is internally computed.
loss_fn = nn.CrossEntropyLoss()
# specify the optimizer to update the weights during backward pass
optimizer = optim.SGD(net.parameters(), lr=LR, momentum=MOMENTUM)
# change learning rate over time
scheduler = lr_scheduler.StepLR(optimizer, step_size=100, gamma=GAMMA)
```

```
def train_net():
    # put the network in training mode
    net.train()
    # keep record of the loss value
    epoch_loss = 0.0
    # use training data as batches
    for xt, rt in train_loader:
        # move training instances and corresponding
          labels into gpu if cuda is available
        xt, rt = xt.to(device), rt.to(device)
        # clear the previously accumulated gradients
        optimizer.zero_grad()
        # forward the network
        yt = net(xt)
        # calculate loss
        loss = loss_fn(yt, rt)
        # make a backward pass, calculate gradients
        loss.backward()
        # update weights
        optimizer.step()
        # accumulate loss
        epoch_loss += loss.item()
    return epoch_loss

## Define test function

def eval_net(loader):
    # put the network in evaluation mode
    net.eval()
    # keep record of the loss value
    total_loss = 0.0
    # number of correctly classified instances
    correct = 0
    # disable gradient tracking
    with torch.no_grad():
        for xt, rt in loader:
            # move training instances and corresponding
              labels into gpu if cuda is available
            xt, rt = xt.to(device), rt.to(device)
            # save_image(xt, f'images/sample_grid.png')  # save 8 images
            # x = 8/0
            # forward the network
            yt = net(xt)
            # calculate loss
            loss = loss_fn(yt, rt)
            # accumulate loss
            total_loss += loss.item()
            # get predicted classes
            pred = yt.argmax(dim=1)
            # accumulate correctly classified image counts
            correct += (pred == rt).sum().item()
            # correct += pred.eq(rt.view_as(pred)).sum().item()
    return correct / len(loader.dataset), total_loss
```

```
# initialize the logger instance
# by default creates run directory inside current folder
writer = SummaryWriter()
# train the network
for epoch in range(1, EPOCHS + 1):
    # train network for one epoch
    train_net()
    scheduler.step()
    # get accuracy and loss on the training dataset
    tr_ac, tr_loss = eval_net(train_loader)
    # get accuracy and loss on the test dataset
    tt_ac, tt_loss = eval_net(test_loader)
    # save stats
    writer.add_scalars("Loss", {"tr_loss": tr_loss, "tt_loss": tt_loss},
epoch)
    writer.add_scalars("Accuracy", {"tr_acc": tr_ac, "tt_acc": tt_ac},
epoch)

    if (epoch - 1) % 10 == 0:
        print("Epoch", epoch, "Tr Acc:", tr_ac, "Tt_Ac", tt_ac)

    writer.flush()
writer.close()


## Save the model

# save the network model
if H is None:
    torch.save(net.state_dict(), 'slp.pth')
else:
    torch.save(net.state_dict(), 'mlp.pth')

% load_ext
tensorboard
% tensorboard - -logdir
runs
# open http://localhost:6006/ to view the results
```