DIT407 Introduction to data science and AI
Assignment 6

Reynir Siik
reynir.siik@gmail.com

Franco Zambon
guszamfr@student.gu.se

2024-02-19

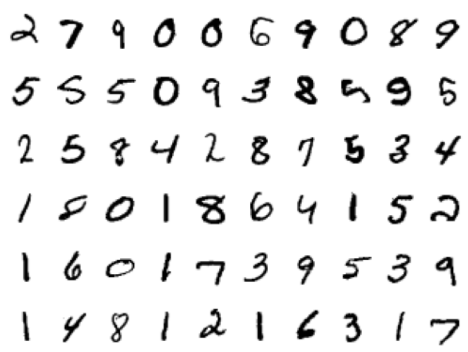
Abstract

This text is from the file `abstract.tex` and will be included in the abstract.

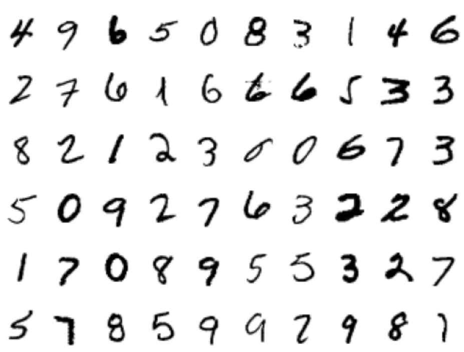
1 Problem 1:

The dataset is loaded with the specified tools, ref. [1].

The transforms function is used with a specifier for normalisation of the data in the range 0-1. The normalisation will use normal distribution with center 0.5 and $\sigma = 0.17$. This means that every (almost) value that falls within 3σ will fit into the specified range of 0-1. This loads a normalized dataset into Tensors as images wit 28 x 28 pixels. This was verified by printing the shape of the samples. See figure 1



(a) Images in the training set



(b) Images in the test set

Figure 1: Examples of the data set images

2 Problem 2: Single hidden layer

The neural network is initialised with the following parameters:

- input size = 784 (28X28 pixel of image)
- hidden layer = 284
- output = 10
- batch size = 2000
- learning rate = 0.01

When training the neural network the following table 3 shows the progress in learning.

Epoch	Accuracy
0	79.70
1	86.25
2	87.05
3	87.50
4	89.60
5	89.55
6	89.85
7	89.85
8	89.70
9	90.60
10	90.90
11	90.80
12	90.05
13	90.30
14	90.90
15	91.45

Table 1: Table showing accuracy after each epoch during training

3 Problem 3: Two hidden layers

3.1 Introduction

Essentially the same code is used for problem 3.
Line 21 is changed to 500
Line 22 is changed to 300
Line 93 is changed to 80
Line 78, 79, 89, 90 is uncommented.

For the optimizer, two parameters are introduced:
momentum = 0.9, and
weight decay =0.001
See A.2

3.2 Results

The accuracy after each epoch is presented in table 2

To reach an accuracy over 98% it took 80 epochs of training.

Epoch	Accuracy	Epoch	Accuracy	Epoch	Accuracy	Epoch	Accuracy
0	87.15	20	96.45	40	97.50	60	98.55
1	90.20	21	95.80	41	97.05	61	97.90
2	91.35	22	96.50	42	97.20	62	98.00
3	91.40	23	97.00	43	97.80	63	97.25
4	91.90	24	96.95	44	97.35	64	98.15
5	92.15	25	96.80	45	97.70	65	97.85
6	92.90	26	96.75	46	97.35	66	97.65
7	94.05	27	97.00	47	97.75	67	97.95
8	92.95	28	96.70	48	97.40	68	97.95
9	93.90	29	96.95	49	97.20	69	97.95
10	93.90	30	97.30	50	97.45	70	98.15
11	94.55	31	97.85	51	98.10	71	97.60
12	94.45	32	97.05	52	98.20	72	97.95
13	95.15	33	97.35	53	97.50	73	98.30
14	95.10	34	96.65	54	97.45	74	98.05
15	96.20	35	96.75	55	97.25	75	98.35
16	95.10	36	96.80	56	97.75	76	97.60
17	95.50	37	97.20	57	97.55	77	98.00
18	96.50	38	97.35	58	97.65	78	98.10
19	96.20	39	96.90	59	97.95	79	98.45

Table 2: Training progress for 2 hidden layer neural network

Noticable is that the two hidden layers reach an accuracy of 96.2 after 16 epochs, whereas the single layer, which of course also is smaller, only had reached 91.45.

4 Problem 4: Convolutional neural network

4.1 Introduction

We implemented a simple convolutional neural network architecture for classifying MNIST digits. The network consisted of two convolutional layers followed by two fully connected layers, and it was trained using SGD with cross-entropy loss.
The training and validation loops were used to monitor the training progress and evaluate the model's performance on the test dataset.

4.2 The process and the results

First of all we defined the neural network architecture in the ConvNet class. We created 2 convolutional layers with a 3x3 kernel size and a padding of 1. We then created 2 fully connected lauyers, one with 128 neurons, the second with 10.
ReLU activation functions were applied after each convolutional layer and the first fully connected layer.
Max pooling layers with a kernel size of 2x2 and a stride of 2 are applied after each convolutional layer to reduce the spatial dimensions.
As before, we used ReLU, SGD, and cross entropy loss, and, to prevent overfitting, chose 0.01 as value for weight loss. We also chose a learning rate of 0.01. We then trained the

network for 40 epochs which seams to be enough since we reached our goal of 99% accuracy. The accuracy obtained is in table 3.

Epoch	Accuracy
0	92.82
1	96.04
2	97.20
3	97.61
4	97.58
5	98.28
6	98.38
7	98.34
8	98.45
9	98.65
10	98.71
11	98.70
12	98.66
13	98.77
14	98.76
15	98.80
16	98.77
17	98.81
18	98.93
19	98.92
20	98.90
21	98.91
22	98.96
23	98.91
24	99.01
25	98.99
26	99.07
27	99.02
28	99.05
29	98.98
30	99.08
31	99.06
32	98.95
33	99.08
34	99.01
35	99.06
36	99.02
37	98.98
38	98.97
39	99.00

Table 3: Table showing accuracy after each epoch during training

References

[1] Y. LeCun, C. Cortes, and C. J. Burges, *The MNIST database of handwritten digits*, Retrieved 2024-01-02, 1998. [Online]. Available: http://yann.lecun.com/exdb/mnist/.

A Source code

This is the complete listing of the source code.

A.1 Problem 1 and 2

Code for splitting the datasets into training and test sets and training a linear model.

```
# Reynir Siik, Franco Zambon
# DIT407 lp3 2024-03-04
# Assignment 6
# Problem 1, 2
# Neural network
#
## ######################################################################
import torch
from torch.utils.data import DataLoader
import torchvision
import matplotlib.pyplot as plt
import torch.nn as nn
from torchvision.transforms import transforms
## ######################################################################
filelocation_in = r"Uppgift06\\00_Uppg_Data\\" # folder to get data from
filelocation_out = r"Uppgift06\\test_data\\" # folder to write data to
## ######################################################################

## initializing parameter
input_size=784   #28X28 pixel of image
hidden_size1=284 #size of 1st hidden layer(if used)
hidden_size=284  #size of last hidden layer
output =10       #output layer
batch_size=2000
lr_rate=0.01
########################################################################
## This transform should make the data normalized in the range 0-1 with
## a probability of 0.97 (of the top of my head)
transform = transforms.Compose([transforms.ToTensor(),
                              transforms.Normalize((0.5,), (0.17,)),
                              ])

# Download test data from open datasets and normalize data in image
# Store the data localy so we don't have to download it everytime
# Training data
train_dataset = torchvision.datasets.MNIST(
    root=filelocation_in,
    train=True,
    download=True,
    transform=transform,
)

# Test data
test_data = torchvision.datasets.MNIST(
    root=filelocation_in,
    train=False,
    download=True,
    transform=transform,
)

```

```
# Create data loaders. Shuffle the data to avoid training on data in a
# certain order
train_dataloader = DataLoader(
    train_dataset, batch_size=batch_size, shuffle=True
    )
test_dataloader = DataLoader(
    test_data, batch_size=batch_size, shuffle=True
    )
#########################################################################
## shape of the data that has been passed in batch
data=iter(train_dataloader)
samples,labels=next(data)

# Printing Images from dataset (quick and dirty)
# To be run twice with different setting to get both datasets
# 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(samples[index].numpy().squeeze(), cmap='gist_gray_r')
# plt.savefig(filelocation_out + 'A6_training_sample.pdf')

## Set up the neural network, modified between problems
Mnist_model = nn.Sequential(
    nn.Linear(input_size, hidden_size1),
    nn.ReLU(),
    # nn.Linear(hidden_size1, hidden_size),
    # nn.ReLU(),
    nn.Linear(hidden_size, output),
    nn.LogSoftmax(dim=1),
)

loss=nn.CrossEntropyLoss()
# Recomended parameters that have been found to work well after trying
# other values
optimizer=torch.optim.SGD(Mnist_model.parameters(),
                          lr=lr_rate,
                        #   momentum=0.9,
                        #   weight_decay=0.05
                          )

num_epochs = 16
print('Epoch␣,␣Accuracy')

for epoch in range(num_epochs):
    for i,(images, labels) in enumerate(train_dataloader):
        images = images.reshape(-1, 28*28)
        ## forward connection
        output = Mnist_model(images)
        Loss = loss(output, labels)
        ## calculating gradient
        optimizer.zero_grad()  # it will not cumulate the gradient result after every
        Loss.backward()        # it will do backward propagation
        optimizer.step()

        # if (i+1)%batch_size == 0:
        #     print(f"epoch={epoch+1}/{num_epochs}, step={i+1}/{len(train_dataloader)
```

```

## model accuracy on validation dataset
    with torch.no_grad():
        n_correct=0
        n_samples=0

        for images,labels in test_dataloader:
            images=images.reshape(-1,784)
            output=Mnist_model(images)
            labels=labels
            _,prediction=torch.max(output,1)
            n_samples=labels.shape[0]
            n_correct=(prediction==labels).sum().item()

        accuracy=(n_correct/n_samples)*100

    print(epoch, ', ', accuracy)
    # print('Epoch: ',epoch , ', Accuracy = ', accuracy)

## predict the result
predicted=[]
with torch.no_grad():
    n_correct=0
    n_samples=0
    for images,labels in test_dataloader:
        images=images.reshape(-1,784)
        output=Mnist_model(images)
        labels=labels
        _,prediction=torch.max(output,1)
        predicted.append(prediction)
print(prediction)
```

A.2 Problem 3

Code for two hidden layers.

```
# Reynir Siik, Franco Zambon
# DIT407 lp3 2024-03-04
# Assignment 6
# Problem 1, 2
# Neural network
#
## #####################################################################
import torch
from torch.utils.data import DataLoader
import torchvision
import matplotlib.pyplot as plt
import torch.nn as nn
from torchvision.transforms import transforms
## #####################################################################
filelocation_in = r"Uppgift06\\00_Uppg_Data\\" # folder to get data from
filelocation_out = r"Uppgift06\\test_data\\" # folder to write data to
## #####################################################################

## initializing parameter
input_size=784   # 28X28 pixel of image
hidden_size1=500 # size of 1st hidden layer(if used)
hidden_size=300  # size of last hidden layer
```

```
output =10        # output layer
batch_size=2000   # batch size
lr_rate=0.01      # learning rate
########################################################################
## This transform should make the data normalized in the range 0-1 with
## a probability of 0.97 (of the top of my head)
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.17,)),
                                ])

# Download test data from open datasets and normalize data in image
# Store the data localy so we don't have to download it everytime
# Training data
train_dataset = torchvision.datasets.MNIST(
    root=filelocation_in,
    train=True,
    download=True,
    transform=transform,
)

# Test data
test_data = torchvision.datasets.MNIST(
    root=filelocation_in,
    train=False,
    download=True,
    transform=transform,
)

# Create data loaders. Shuffle the data to avoid training on data in a
# certain order
train_dataloader = DataLoader(
    train_dataset, batch_size=batch_size, shuffle=True
    )
test_dataloader = DataLoader(
    test_data, batch_size=batch_size, shuffle=True
    )
########################################################################
## shape of the data that has been passed in batch
data=iter(train_dataloader)
samples,labels=next(data)

# Printing Images from dataset (quick and dirty)
# To be run twice with different setting to get both datasets
# 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(samples[index].numpy().squeeze(), cmap='gist_gray_r')
# plt.savefig(filelocation_out + 'A6_training_sample.pdf')

## Set up the neural network, modified between problems
Mnist_model = nn.Sequential(
    nn.Linear(input_size, hidden_size1),
    nn.ReLU(),
    nn.Linear(hidden_size1, hidden_size),
    nn.ReLU(),
    nn.Linear(hidden_size, output),
    nn.LogSoftmax(dim=1),
```

```
)

loss=nn.CrossEntropyLoss()
# Recomended parameters that have been found to work well after trying
# other values
optimizer=torch.optim.SGD(Mnist_model.parameters(),
                          lr=lr_rate,
                          momentum=0.9,
                          weight_decay=0.001
                          )

num_epochs = 80
print('Epoch␣,␣Accuracy')

for epoch in range(num_epochs):
    for i,(images, labels) in enumerate(train_dataloader):
        images = images.reshape(-1, 28*28)
        ## forward connection
        output = Mnist_model(images)
        Loss = loss(output, labels)
        ## calculating gradient
        optimizer.zero_grad()  # it will not cumulate the gradient result after every
        Loss.backward()        # it will do backward propagation
        optimizer.step()

        # if (i+1)%batch_size == 0:
        #     print(f"epoch={epoch+1}/{num_epochs}, step={i+1}/{len(train_dataloader


## model accuracy on validation dataset
    with torch.no_grad():
        n_correct=0
        n_samples=0

        for images,labels in test_dataloader:
            images=images.reshape(-1,784)
            output=Mnist_model(images)
            labels=labels
            _,prediction=torch.max(output,1)
            n_samples=labels.shape[0]
            n_correct=(prediction==labels).sum().item()

        accuracy=(n_correct/n_samples)*100

    print(f'{epoch}␣,␣{accuracy:.2f}')
    # print('Epoch: ',epoch , ', Accuracy = ', accuracy)

## predict the result
predicted=[]
with torch.no_grad():
    n_correct=0
    n_samples=0
    for images,labels in test_dataloader:
        images=images.reshape(-1,784)
        output=Mnist_model(images)
        labels=labels
        _,prediction=torch.max(output,1)
        predicted.append(prediction)
print(prediction)
```

A.3 Problem 4

Code for Non-linear relationship and multiple linear regression.

```
import torch.optim as optim
import torchvision.datasets as datasets

## PROBLEM 4

## Define the neural network architecture

class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, stride=
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(x, kernel_size=2, stride=2)
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, kernel_size=2, stride=2)
        x = x.view(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x


## Initialize the model and optimizer

model = ConvNet()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=0.01)



## Training and validation

num_epochs = 40
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for batch_idx, (data, targets) in enumerate(train_loader):
        optimizer.zero_grad()
        outputs = model(data)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(train_loader):.4f

    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for data, targets in test_loader:
            outputs = model(data)
```

```
            _, predicted = torch.max(outputs.data, 1)
            total += targets.size(0)
            correct += (predicted == targets).sum().item()
    print(f"Validation Accuracy: {(correct/total)*100:.2f}%")
```

```
# Reynir Siik, Franco Zambon
# DIT407 lp3 2024-02-11
# Assignment 4
# Problem 1, 2
# Life expectancy
# Non-linear relationship, Multiple linear regression
#########################################################################
```