DAT565/DIT407 Assignment 6

Connell Hagen
connellh@student.chalmers.se

Måns Redin
mansre@student.chalmers.se

2024-10-16

1 Problem 1: Pre-processing the Dataset

The dataset used in this project was a MNIST dataset. This pre-defined PyTorch dataset consists of 70 000 images of handwritten digits, with their respective labels. The dataset was split into two parts: a training set of size 60 000, and a test set of the remainder. Different neural network models will be discussed and were implemented with their test accuracy showing which were most effective. The images size was verified to be 28x28, with all pixels in the normalized greyscale.

2 Problem 2: Single Hidden Layer

We constructed a fully-connected, one layer neural network using ReLU as the activation function, Stochastic Gradient Descent as the optimization function, and Cross Entropy Loss as the loss function. We did not use any special parameters for this model beyond the default. Our single layer was 200 neurons large. The total accuracy came out to be 87.17%. The accuracy after each epoch can be viewed in table 1.

Table 1: The Single-Layer Model's Validation Accuracy with Respect to the Test Set for 10 Epochs

Epoch	Accuracy
1	36.12
2	39.82
3	43.81
4	44.45
5	45.17
6	45.99
7	47.54
8	49.32
9	50.76
10	51.88

3 Problem 3: Two Hidden Layers

We expanded our network by using a second hidden layer, next. For these layers, the first was 500 neurons large, and the second was 300 neurons large. We used ReLU, SGD, and Cross Entropy Loss, like the previous model. We also enabled L2 regularization by using the weight decay parameter with a value of 0.0001. We also set the learning rate to 0.01. The total The results can be viewed in table 2.

Table 2: The Dual-Layer Model's Validation Accuracy with Respect to the Test Set for 60 Epochs

Epoch	Accuracy	Epoch	Accuracy	Epoch	Accuracy
1	71.89	21	96.31	41	97.49
2	88.39	22	96.30	42	97.53
3	90.02	23	96.37	43	97.62
4	91.14	24	96.45	44	97.65
5	92.09	25	96.61	45	97.62
6	92.38	26	96.57	46	97.70
7	93.16	27	96.66	47	97.72
8	93.50	28	96.89	48	97.74
9	93.79	29	97.03	49	97.72
10	97.04	30	97.05	50	97.81
11	94.26	31	97.09	51	97.71
12	94.55	32	97.25	52	97.74
13	94.87	33	97.31	53	97.79
14	95.05	34	97.34	54	97.75
15	95.24	35	97.30	55	97.72
16	95.26	36	97.38	56	97.90
17	95.65	37	97.37	57	97.87
18	95.85	38	97.33	58	97.94
19	96.06	39	97.49	59	97.93
20	96.18	40	97.57	60	97.82

As can be seen in table 2, the accuracy increased vastly from the neural network with one hidden layer.

4 Problem 4: Convolutional Neural Network

In this part of the project, a Convolutional Neural Network (CNN) was implemented. CNNs are designed to detect essential features in early layers, and analyze each feature further in the deeper layers. The architecture was made of two convolutional layers, a pooling layer, and 2 fully-connected layers, making it fairly computationally heavy, certainly heavier than the recent trained models. It comes, however, with an increased accuracy for image recognition. We used a weight decay value of 0.00001, and a learning rate of 0.01 for this model. We reached an accuracy of 98.9% at best, but for the last 40 epochs the accuracy oscillated around 98.5-99%.

Table 3: The Convolutional Model's Validation Accuracy with Respect to the Test Set for 60 Epochs

Epoch	Accuracy	Epoch	Accuracy	Epoch	Accuracy
1	85.68	21	98.35	41	98.67
2	91.70	22	98.48	42	98.87
3	94.88	23	98.02	43	98.63
4	94.80	24	98.47	44	98.64
5	96.45	25	98.12	45	98.78
6	97.32	26	98.04	46	98.30
7	97.21	27	98.61	47	98.77
8	97.82	28	98.66	48	98.55
9	97.77	29	98.63	49	98.76
10	97.85	30	98.63	50	98.78
11	98.17	31	98.70	51	98.75
12	98.11	32	98.53	52	98.88
13	97.85	33	98.66	53	98.69
14	98.27	34	98.64	54	98.85
15	98.27	35	98.51	55	98.85
16	98.24	36	98.75	56	98.82
17	98.40	37	98.75	57	98.89
18	98.39	38	98.84	58	98.89
19	97.92	39	98.55	59	98.80
20	98.38	40	98.64	60	98.85

A Code

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import torch
import torch.nn.functional as F
from PIL import Image
from torch import nn
from torch.utils.data import DataLoader
from torch.nn import CrossEntropyLoss
from torch.optim import SGD
from torchvision import datasets
from torchvision.transforms import ToTensor
from tqdm import tqdm

training_data = datasets.MNIST(
    root="./data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.MNIST(
    root="./data",
    train=False,
    download=True,
    transform=ToTensor()
)

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)


train_features, train_labels = next(iter(train_dataloader))
img = train_features[0].squeeze()
label = train_labels[0]
plt.imshow(img, cmap="gray")
plt.show()

test_features, test_labels = next(iter(test_dataloader))
img = test_features[0].squeeze()
label = test_labels[0]
plt.imshow(img, cmap="gray")
```

```python
plt.show()


L1_NEURONS = 200

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

        self.fc1 = nn.Linear(28 * 28, L1_NEURONS)
        self.fc2 = nn.Linear(L1_NEURONS, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x));
        x = F.relu(self.fc2(x))

        return x

model = OneLayer()
loss_fn = CrossEntropyLoss()
optimizer = SGD(model.parameters())


epochs = 10
running_loss = 0

for e in range(epochs):
    for i, data in tqdm(enumerate(train_dataloader)):
        model.train()

        inputs, labels = data

        optimizer.zero_grad()

        flattened_inputs = inputs.view(inputs.size(0), -1)
        outputs = model(flattened_inputs)

        loss = loss_fn(outputs, labels)
        loss.backward()

        optimizer.step()

        running_loss += loss.item()

    print(f"Epoch {e}: Average loss of {running_loss / 64} per batch")
    running_loss = 0

    # test benchmark
```

```
    model.eval()

    total_correct = 0
    total_samples = 0

    for i, (inputs, labels) in tqdm(enumerate(test_dataloader)):
        flattened_inputs = inputs.view(inputs.size(0), -1)

        outputs = model(flattened_inputs)
        _, predicted = torch.max(outputs, 1)

        total_correct += (predicted == labels).sum().item()
        total_samples += labels.size(0)

    accuracy = 100 * total_correct / total_samples
    print(f"Epoch {e}, Accuracy: {accuracy}")

torch.save(model.state_dict(), "model/one_layer_model.pth")


L1_NEURONS = 500
L2_NEURONS = 300

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

        self.fc1 = nn.Linear(28 * 28, L1_NEURONS)
        self.fc2 = nn.Linear(L1_NEURONS, L2_NEURONS)
        self.fc3 = nn.Linear(L2_NEURONS, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))

        return x

model = TwoLayers()
loss_fn = CrossEntropyLoss()
optimizer = SGD(model.parameters(), weight_decay=1e-4, lr=0.01) # weight decay for L2 regularization

epochs = 60
running_loss = 0

```

```python
for e in range(epochs):
    for i, data in tqdm(enumerate(train_dataloader)):
        model.train()

        inputs, labels = data

        optimizer.zero_grad()

        flattened_inputs = inputs.view(inputs.size(0), -1)
        outputs = model(flattened_inputs)

        loss = loss_fn(outputs, labels)
        loss.backward()

        optimizer.step()

        running_loss += loss.item()

    print(f"Epoch {e}: Average loss of {running_loss / 64} per batch")
    running_loss = 0

    # test benchmark
    model.eval()

    total_correct = 0
    total_samples = 0

    for i, (inputs, labels) in tqdm(enumerate(test_dataloader)):
        flattened_inputs = inputs.view(inputs.size(0), -1)

        outputs = model(flattened_inputs)
        _, predicted = torch.max(outputs, 1)

        total_correct += (predicted == labels).sum().item()
        total_samples += labels.size(0)

    accuracy = 100 * total_correct / total_samples
    print(f"Epoch {e}, Accuracy: {accuracy}")

torch.save(model.state_dict(), "model/2-layer-0-01-learning-rate.pth")


L1_OUTPUT_CHANNELS = 16
L2_OUTPUT_CHANNELS = 32
```

```python

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

        # convolutional layers
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=L1_OUTPUT_CHANNELS, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=L1_OUTPUT_CHANNELS, out_channels=L2_OUTPUT_CHANNELS, kernel_size=3, padding=1)

        # pooling layer
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # fully-connected layers
        self.fc1 = nn.Linear(L2_OUTPUT_CHANNELS * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)


    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, L2_OUTPUT_CHANNELS * 7 * 7) # flatten images
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x

model = ConvNetwork()
loss_fn = CrossEntropyLoss()
optimizer = SGD(model.parameters(), weight_decay=1e-5, lr=0.01) # weight decay for L2 regularization


epochs = 60
running_loss = 0

for e in range(epochs):
    for i, data in tqdm(enumerate(train_dataloader)):
        model.train()

        inputs, labels = data

        optimizer.zero_grad()
```

```

        outputs = model(inputs)

        loss = loss_fn(outputs, labels)
        loss.backward()

        optimizer.step()

        running_loss += loss.item()

    print(f"Epoch {e}: Average loss of {running_loss / 64} per batch")
    running_loss = 0

    # test benchmark
    model.eval()

    total_correct = 0
    total_samples = 0

    for i, (inputs, labels) in tqdm(enumerate(test_dataloader)):
        outputs = model(inputs)
        _, predicted = torch.max(outputs, 1)

        total_correct += (predicted == labels).sum().item()
        total_samples += labels.size(0)

    accuracy = 100 * total_correct / total_samples
    print(f"Epoch {e}, Accuracy: {accuracy}")

torch.save(model.state_dict(), "model/conv-network.pth")
```