Building and Evaluating Neural Networks for MNIST

Clara Petrescu-Stompor
clara.petrescu.stompor@gmail.com

October 30, 2025

Loading the Training and Test Datasets

First, I load both dataset (training set and test set) using the module `torch`, I also load the open dataset `MNIST` and the `DataLoader` from the module `torch` to split the images into 100 batches and iterate over them using "Python's multiprocessing". The code for this can be found in appendix A. Note that I used the `ToTensor()` method from the module `transforms` to transform images into `Tensors`.

Then, I plot the first image of both the training set and the test set. The plot of the first image can be found in Figure 1, and the second image in Figure 2. Note that at the same time, I also check the shape of the images using the `shape` attribute and retrieve the `max` (resp. `min`) values of the image to ensure that the maximum is less than 1 (resp. The minimum is greater than 0), to ensure that the values are normalized within the range $[0, 1]$. The corresponding code can be found in B and C. Indeed, I found that the images have dimensions $(28, 28)$ and the maximum of the values is 1 and the minimum is 0.

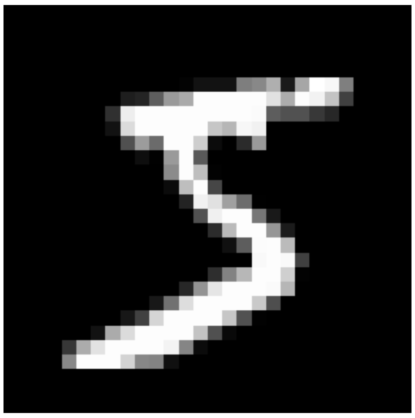

Figure 1: Plot of the first image of the training set in greyscale

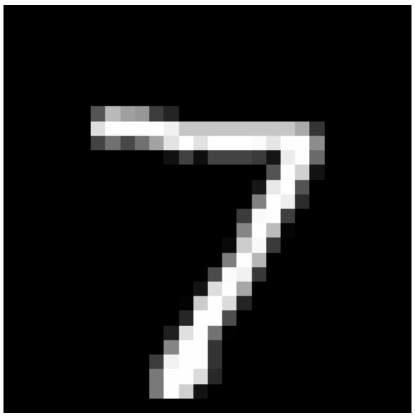

Figure 2: Plot of the first image of the test set in greyscale

Single Hidden-Layer Neural Network

Using the module `nn` from `torch`, I have built an inherited class from `nn` called `SingleLayerNeuralNetwork` that allows me to create an instance of neural network that I will train and test. The class attributes described the single hidden layer, I chose a hidden layer with 400 neurons. Since an image has a dimension of $(28, 28)$, the model needs $28 * 28$ inputs and since there exists 10 digits, there are 10 possible outputs. The code for creating our model using this class can be found in appendix D.

Then, I define two functions to easily train and test a model and compute its accuracy after each epoch given parameters like the model to train/test, the number of epochs, the optimizer, and the loss function. Since the first layer is a linear layer, I need to flatten the image before using it as input. Its shape will change from $(28, 28)$ to $28 * 28$. The code of my functions can be found in appendix E. Note that the `test_neural_network` function also allows me to compute the accuracy of the given model.

Next, I set the optimizer, the cross validation functions and the numbers of epochs. I chose the following parameters :

- `learning rate (lr)` = `0.001`
- `momentum` = `0.99`
- `Number of epochs (n_epochs)` = `20`

After processing the `test_train_neural_network` function on the model using the code in appendix F, I reach an *accuracy* of `92.54 %`, the `accuracy` after each epoch can be found in the table 1.

Epoch	Accuracy
1	92.39
2	92.42
3	92.44
4	92.47
5	92.48
6	92.48
7	92.5
8	92.52
9	92.52
10	92.52
11	92.57
12	92.56
13	92.56
14	92.58
15	92.57
16	92.57
17	92.57
18	92.6
19	92.57
20	92.54

Table 1: Accuracy after each Epoch for a model with a Single Hidden Layer

Two hidden layers Neural Network

Then I constructed a fully-connected feedforward network with two hidden layers that have 500 and 300 units, respectively. As before, I used `ReLU`, `SGD`, and cross entropy loss. The code to do this can be found in appendix G. I chose the parameters :

- `learning rate (lr)` = `0.001`
- `momentum` = `0.99`
- `Number of epochs (n_epochs)` = `50`
- `Weight decay` = `0.0001`

The **learning rate (lr)** controls how strongly new information replaces old information. A rate of 0 means the agent won't learn anything, while a rate of 1 makes the agent consider only the most recent information. So the **learning rate (lr)** controls how quickly the model is adapted to the problem. Smaller learning rates require more training epochs given the smaller changes made to the weights each update, whereas larger learning rates result in rapid changes and require fewer training epochs. So because I choose a `Number of epochs (n_epochs)` of `50` (which is a big number but not so much), we chose a rather small (but not so much)`learning rate (lr)` of `0.001`.

It is also known that `momentum` values of 0.9 and 0.99 achieve reasonable train and test accuracy within about 50 training epochs as opposed to 200 training epochs when momentum is not used. This is why I chose a `momentum` of `0.99`.

The `Weight decay` acts as a regularizing force that penalizes large weights, promoting smaller, more generalizable models. I chose a rather small `Weight decay` (a value of `0.0001`) to prevent underfitting. But I didn't choose a too small value to avoid overfitting, because the **learning rate (lr)** is already a bit low.

Then I trained the network for 50 epochs. The code to do this can be found in appendix H. The validation accuracy after each epoch can be found in Table 2. As we can see in this Table 2, I succeeded in reaching an *accuracy* of `97.99 %`.

Epoch	Accuracy
1	83.08
2	84.53
3	85.55
4	86.26
5	86.74
6	87.12
7	95.96
8	96.3
9	96.62
10	96.9
11	97.05
12	97.25
13	97.39
14	97.51
15	97.57
16	97.56
17	97.57
18	97.56
19	97.6
20	97.67
21	97.67
22	97.69
23	97.71
24	97.68
25	97.75
26	97.76
27	97.74
28	97.7
29	97.7
30	97.74
31	97.75
32	97.74
33	97.76
34	97.79
35	97.87
36	97.88
37	97.96
38	97.97
39	97.84
40	97.85
41	97.88
42	97.84
43	97.92
44	97.93
45	97.97
46	97.98
47	97.98
48	97.99
49	97.99
50	97.99

Table 2: Accuracy after each Epoch for a model with Double Hidden Layer

Convolutional neural network

I constructed a feedforward network with two convolutional layers in the same way as before. The code of the class to create an instance of the model can be found in appendix I. This neural network is composed of one convolutional layer (which transforms the $(28, 28)$ image input into a $(24, 24)$ matrix with 16 features for each cell) followed by a convolutional layer with a $(2, 2)$ kernel which takes the maximum value among the 4 selected by the kernel. Thus, I transform a $(24, 24)$ matrix into a $(12, 12)$ with 16 features which considerably reduces the number of computations and accentuates the grey tones. I repeat this process until reaching a $(4, 4)$ matrix with 32 features for each cell. Then, using a `Flatten()` layer, I flatten the matrix into a vector that will be put in a hidden linear layer with 50 neurons followed by the final layer to obtain an output with 10 possible values.

Then, using the following parameters :

- `learning rate (lr)` = `0.001`
- `momentum` = `0.99`
- `Number of epochs (n_epochs)` = `50`

And after training and testing my model, I manage to obtain an `accuracy` of `99.1%`. The code to train and test the model is very similar to the previous code, except that I do not need to `reshape` the input images since the first layer of our model needs a $(28, 28)$ matrix as input. It can be found in appendix J. The table 3 below reports accuracy found after each epoch with the model.

Epoch	Accuracy
1	95.17
2	97.73
3	97.76
4	98.21
5	98.47
6	98.46
7	98.34
8	98.72
9	98.74
10	98.52
11	98.44
12	98.75
13	98.68
14	98.69
15	98.89
16	98.8
17	98.74
18	98.69
19	98.99
20	98.91
21	98.93
22	98.87
23	98.82
24	98.89
25	98.79
26	98.86
27	99.05
28	99.0
29	99.02
30	99.01
31	98.96
32	99.02
33	99.07
34	99.09
35	99.06
36	99.09
37	99.08
38	99.1
39	99.1
40	99.1
41	99.1
42	99.1
43	99.1
44	99.11
45	99.11
46	99.1
47	99.1
48	99.11
49	99.1
50	99.09

Table 3: Accuracy after each Epoch for a model with Convolutional Layers

A Code to load the data

```python
# import pytorch
import torch

# import datasets and loader modules
from torch.utils.data import DataLoader
from torchvision.datasets import MNIST
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt

# Download training
training_data = MNIST(
    root ='data',
    train=True,
    download=True,
    transform=ToTensor()
)
print(training_data)

# Download test data
test_data = MNIST(
    root ='data',
    train=False,
    download=True,
    transform=ToTensor()
)
print(test_data)

batch_size = 100

# Create data loaders.
train_dataloader = DataLoader(
    training_data, batch_size=batch_size
)
test_dataloader = DataLoader(
    test_data, batch_size=batch_size
)
train_dataloader
```

B Code to plot the first image of the training set

```python
train_image_tensor, label = training_data[0]
# check the dimensions
print(train_image_tensor.shape, label)
# Check scale for values
```

```python
print(torch.max(train_image_tensor), torch.min(train_image_tensor))
plt.imshow(train_image_tensor[0,:,:],cmap = 'gray')
plt.axis('off')
```

C Code to plot the first image of the test set

```python
test_image_tensor, label = test_data[0]
# check the dimensions
print(test_image_tensor.shape, label)
# Check scale for values
print(torch.max(test_image_tensor), torch.min(test_image_tensor))
plt.imshow(test_image_tensor[0,:,:],cmap = 'gray')
plt.axis('off')
```

D Code of the class for the single hidden layer neural network

```python
# Build the neural network
from torch import nn

# Device configuration, this is to check if GPU
# is available and run on GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

input_size = 28*28
hidden_size = 400
output_size = 10

# Construct a fully-connected feedforward network with one hidden layer
class SingleLayerNeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.input_layer = nn.Linear(input_size, hidden_size)
        self.output_layer = nn.Linear(hidden_size, output_size)
        self.act = nn.ReLU()

    def forward(self, x):
        res = self.input_layer(x)
        res = self.act(self.output_layer(res))
        return res

# Initiate a model
nn_model_pb2 = SingleLayerNeuralNetwork().to(device)
```

E Code to define the functions to test and train of a neural network model

```python
from pandas import DataFrame

# Test the model
def test_neural_network(epoch, nn_model, accuracy_dict):
    with torch.no_grad():
        n_correct = 0
        n_samples = 0
        for images, labels in test_dataloader:
            images = images.reshape(-1, input_size).to(device)
            labels = labels.to(device)

            outputs = nn_model(images)

            _, predictions = torch.max(outputs, 1)
            n_samples += labels.shape[0]
            n_correct += (predictions == labels).sum().item()

        acc = 100.0 * n_correct / n_samples
        accuracy_dict["Epoch"].append(epoch + 1)
        accuracy_dict["Accuracy"].append(acc)
        print(f'Epoch␣{epoch+1}:␣Accuracy␣=␣{acc}')

# Test & Train the model for each epoch
def test_train_neural_network(n_epochs, nn_model, loss_fn, optimizer):
    # To export Accuracy in the report
    accuracy_dict = {
        "Epoch" : list(),
        "Accuracy" : list()
    }

    for epoch in range(n_epochs):
        # Train the model for the current epoch
        for i, (images, labels) in enumerate(train_dataloader):
            # reshape to batch_size, input_size
            images = images.reshape(-1, input_size).to(device)
            labels = labels.to(device)

            # apply forwar of our nn_model
            outputs = nn_model(images)

            loss = loss_fn(outputs, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        # Test the model for the current epoch
        test_neural_network(epoch, nn_model, accuracy_dict)

    return DataFrame(accuracy_dict)
```

F Code to train and test the model with a single hidden layer

```python
# Set parameters
loss_fn_pb2 =  nn.CrossEntropyLoss()
optimizer_pb2 = torch.optim.SGD(
    nn_model_pb2.parameters(), lr=0.001, momentum=0.99
)

# train & test the model
n_epochs_pb2 = 20

accuracy_pb2 = test_train_neural_network(
    n_epochs_pb2,
    nn_model_pb2,
    loss_fn_pb2,
    optimizer_pb2
)
```

G Code of the class for the double hidden layer neural network

```python
# Hidden layers size
hidden_size_pb3_1 = 500
hidden_size_pb3_2 = 300

# Fully-connected feedforward network with two hidden layer
class TwoLayersNeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.input_layer = nn.Linear(input_size,
        hidden_size_pb3_1)
        self.hidden_layers = nn.Linear(hidden_size_pb3_1,
        hidden_size_pb3_2)
        self.output_layer = nn.Linear(hidden_size_pb3_2,
        output_size)
        self.act1 = nn.ReLU()
        self.act2 = nn.ReLU()

    def forward(self, x):
        res = self.input_layer(x)
        res = self.act1(self.hidden_layers(res))
        res = self.act2(self.output_layer(res))
        return res

# Initiate a model
nn_model_pb3 = TwoLayersNeuralNetwork().to(device)

# Cross Entropy Loss
loss_fn_pb3 =  nn.CrossEntropyLoss()

# Stochastic Gradient Descent
optimizer_pb3 = torch.optim.SGD(
    nn_model_pb3.parameters(),
    lr=0.001,
    momentum=0.99,
    weight_decay=0.0001
)
```

H Code to train and test the model with double hidden layer

```
# Number of epochs
n_epochs_pb3 = 50 # at least 40

# Accuracy after the training and testing the model
accuracy_pb3 = test_train_neural_network(
    n_epochs_pb3,
    nn_model_pb3,
    loss_fn_pb3,
    optimizer_pb3,
)
```

I Code of the class for the convolutional neural network

```
hidden_size_pb4 = 50

# Feedforward network with at
# least two convolutional layers
class ConvolutionalLayersNeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(
            in_channels=1,
            out_channels=16,
            kernel_size=5,
            stride=1,
            padding=0
        )
        self.pool1 = nn.MaxPool2d(
            kernel_size=2,
            stride=2
        )
        self.conv2 = nn.Conv2d(
            in_channels=16,
            out_channels=32,
            kernel_size=5,
            stride=1,
            padding=0
        )
        self.pool2 = nn.MaxPool2d(
            kernel_size=2,
            stride=2
        )
        self.hidden_layer = nn.Linear(32*4*4, hidden_size_pb4)
        self.flat = nn.Flatten()
        self.output_layer = nn.Linear(hidden_size_pb4, output_size)
        self.act = nn.ReLU()

    def forward(self, x):
        res = self.conv1(x)
        res = self.pool1(res)
        res = self.conv2(res)
        res = self.pool2(res)
        res = self.flat(res)
        res = self.act(self.hidden_layer(res))
        res = self.output_layer(res)
        return res

nn_model_pb4 = ConvolutionalLayersNeuralNetwork().to(device)
nn_model_pb4
```

J Code to train and test the model with convolutional layers

```
n_epochs_pb4 = 50
loss_fn_pb4 =  nn.CrossEntropyLoss()
optimizer_pb4 = torch.optim.SGD(
    nn_model_pb4.parameters(),
    lr=0.001,
    momentum=0.99
)

# To export data
accuracy_pb4 = {
```

```python
    "Epoch": list(),
    "Accuracy": list()
}

for epoch in range(n_epochs_pb4):
    # Train the model
    for images, labels in train_dataloader:
        outputs = nn_model_pb4(images)

        loss = loss_fn_pb4(outputs, labels)
        optimizer_pb4.zero_grad()
        loss.backward()
        optimizer_pb4.step()

    # Test the model
    with torch.no_grad():
        n_correct = 0
        n_samples = 0
        for images, labels in test_dataloader:
            outputs = nn_model_pb4(images)
            _, predictions = torch.max(outputs, 1)
            n_samples += labels.shape[0]
            n_correct += (predictions == labels).sum().item()

        acc = 100.0 * n_correct / n_samples
        accuracy_pb4["Epoch"].append(epoch + 1)
        accuracy_pb4["Accuracy"].append(acc)
        print(f'Epoch: {epoch+1}, Accuracy: {acc}')
```