

Problem 2 Sample Code

This sample code is meant as a guide on how to use PyTorch and how to use the relevant model layers. This not a guide on how to design a network and the network in this example is intentionally designed to have poor performance.

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
```

Loading MNIST

The `torchvision` module contains links to many standard datasets. We can load the MNIST dataset into a `Dataset` object as follows:

```
In [2]: train_dataset = datasets.MNIST('./data', train=True, download=True, # Downloads i
nto a directory ../data
                                transform=transforms.ToTensor())
test_dataset = datasets.MNIST('./data', train=False, download=False, # No need to
download again
                                transform=transforms.ToTensor())

0.0%

Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./dat
a/MNIST/raw/train-images-idx3-ubyte.gz

100.1%

Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

28.4%

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./dat
a/MNIST/raw/train-labels-idx1-ubyte.gz

0.5%5%

Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data
/MNIST/raw/t10k-images-idx3-ubyte.gz

180.4%

Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data
/MNIST/raw/t10k-labels-idx1-ubyte.gz
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw
Processing...
Done!
```

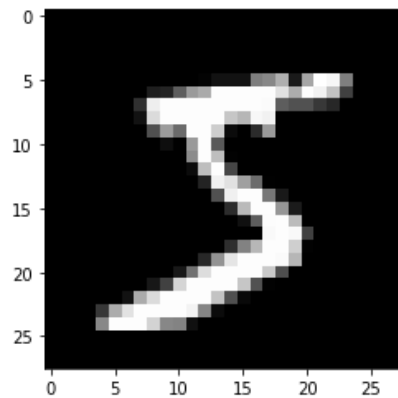
The `Dataset` object is an iterable where each element is a tuple of (input `Tensor` , target):

```
In [3]: print(len(train_dataset), type(train_dataset[0][0]), type(train_dataset[0][1]))  
60000 <class 'torch.Tensor'> <class 'int'>
```

We can convert images to numpy arrays and plot them with matplotlib:

```
In [4]: plt.imshow(train_dataset[0][0][0].numpy(), cmap='gray')
```

```
Out[4]: <matplotlib.image.AxesImage at 0x10787c910>
```



Network Definition

Let's instantiate a model and take a look at the layers.

```
In [64]: model = nn.Sequential(  
    # In problem 2, we don't use the 2D structure of an image at all. Our network  
    # takes in a flat vector of the pixel values as input.  
    nn.Flatten(),  
    nn.Linear(784, 100),  
    nn.ReLU(),  
    nn.Linear(100, 60),  
    nn.ReLU(),  
    nn.Linear(60, 40),  
    nn.ReLU(),  
    nn.Linear(40, 10),  
    nn.LogSoftmax(dim=1)  
)  
print(model)
```

```
Sequential(  
  (0): Flatten()  
  (1): Linear(in_features=784, out_features=100, bias=True)  
  (2): ReLU()  
  (3): Linear(in_features=100, out_features=60, bias=True)  
  (4): ReLU()  
  (5): Linear(in_features=60, out_features=40, bias=True)  
  (6): ReLU()  
  (7): Linear(in_features=40, out_features=10, bias=True)  
  (8): LogSoftmax()  
)
```

Training

We also choose an optimizer and a loss function.

```
In [65]: optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
         loss_fn = nn.CrossEntropyLoss()
```

We could write our training procedure manually and directly index the `Dataset` objects, but the `DataLoader` object conveniently creates an iterable for automatically creating random minibatches:

```
In [66]: train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)
         test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32, shuffle=True)
```

We now write our backpropagation loop, training for 10 epochs.

```
In [72]: # Some layers, such as Dropout, behave differently during training
model.train()

for epoch in range(20):
    for batch_idx, (data, target) in enumerate(train_loader):
        # Erase accumulated gradients
        optimizer.zero_grad()

        # Forward pass
        output = model(data)

        # Calculate loss
        loss = loss_fn(output, target)

        # Backward pass
        loss.backward()

        # Weight update
        optimizer.step()

    # Track loss each epoch
    print('Train Epoch: %d Loss: %.4f' % (epoch + 1, loss.item()))
```

```
Train Epoch: 1 Loss: 0.0000
Train Epoch: 2 Loss: 0.0258
Train Epoch: 3 Loss: 0.0001
Train Epoch: 4 Loss: 0.0004
Train Epoch: 5 Loss: 0.0078
Train Epoch: 6 Loss: 0.0007
Train Epoch: 7 Loss: 0.0000
Train Epoch: 8 Loss: 0.0000
Train Epoch: 9 Loss: 0.0001
Train Epoch: 10 Loss: 0.0863
Train Epoch: 11 Loss: 0.0000
Train Epoch: 12 Loss: 0.0066
Train Epoch: 13 Loss: 0.0001
Train Epoch: 14 Loss: 0.0001
Train Epoch: 15 Loss: 0.0003
Train Epoch: 16 Loss: 0.0000
Train Epoch: 17 Loss: 0.0002
Train Epoch: 18 Loss: 0.0048
Train Epoch: 19 Loss: 0.0000
Train Epoch: 20 Loss: 0.0000
```

Testing

We can perform forward passes through the network without saving gradients.

```
In [73]: # Putting layers like Dropout into evaluation mode
model.eval()

test_loss = 0
correct = 0

# Turning off automatic differentiation
with torch.no_grad():
    for data, target in test_loader:
        output = model(data)
        test_loss += loss_fn(output, target).item() # Sum up batch loss
        pred = output.argmax(dim=1, keepdim=True) # Get the index of the max class score
        correct += pred.eq(target.view_as(pred)).sum().item()

test_loss /= len(test_loader.dataset)

print('Test set: Average loss: %.4f, Accuracy: %d/%d (%.4f)' %
      (test_loss, correct, len(test_loader.dataset),
       100. * correct / len(test_loader.dataset)))

Test set: Average loss: 0.0044, Accuracy: 9813/10000 (98.1300)
```

In []:

In []: