NeuralNetworks

November 26, 2024

1 Imports for Python libraries

[1]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import numpy as np
import torch
import torchvision
import matplotlib.pyplot as plt
from time import time
from torchvision import datasets, transforms
from torch import nn
from torch import optim
```

2 Set up the mini-batch size

[14]:
```
#@title Batch Size
mini_batch_size = 64 #@param {type: "integer"}
```

3 Download the dataset, pre-process, and divide into mini-batches

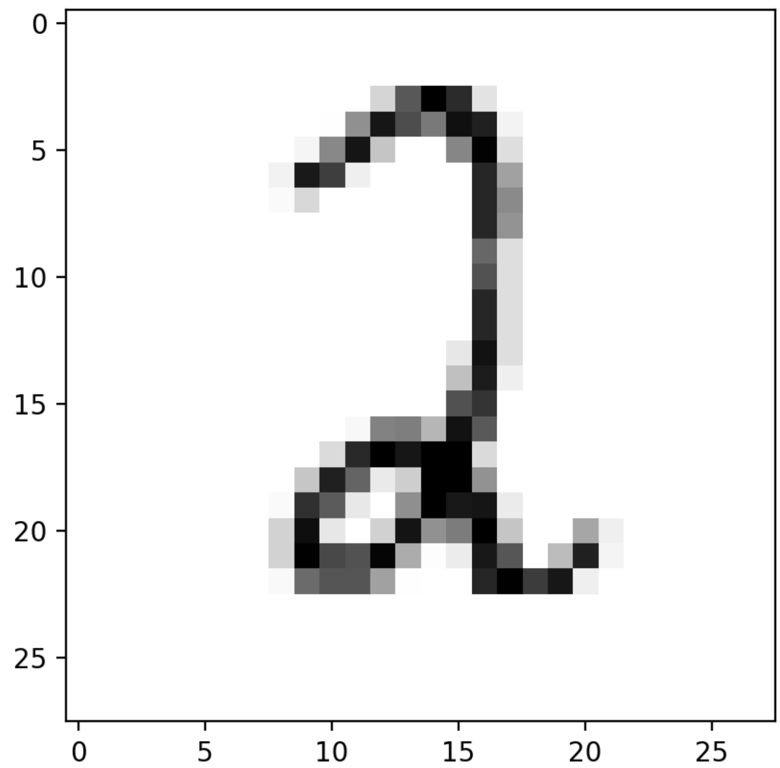
[15]:
```
### Define a transform to normalize the data
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.
↪5,), (0.5,)),])

### Download and load the training data
trainset = datasets.MNIST('MNIST_data/', download=True, train=True,␣
↪transform=transform)
valset = datasets.MNIST('MNIST_data/', download=True, train=False,␣
↪transform=transform)

trainloader = torch.utils.data.DataLoader(trainset, batch_size=mini_batch_size,␣
↪shuffle=True)
valloader = torch.utils.data.DataLoader(valset, batch_size=mini_batch_size,␣
↪shuffle=True)
```

```
dataiter = iter(trainloader)
images, labels = next(dataiter)
print(type(images))
print(images.shape)
print(labels.shape)
```

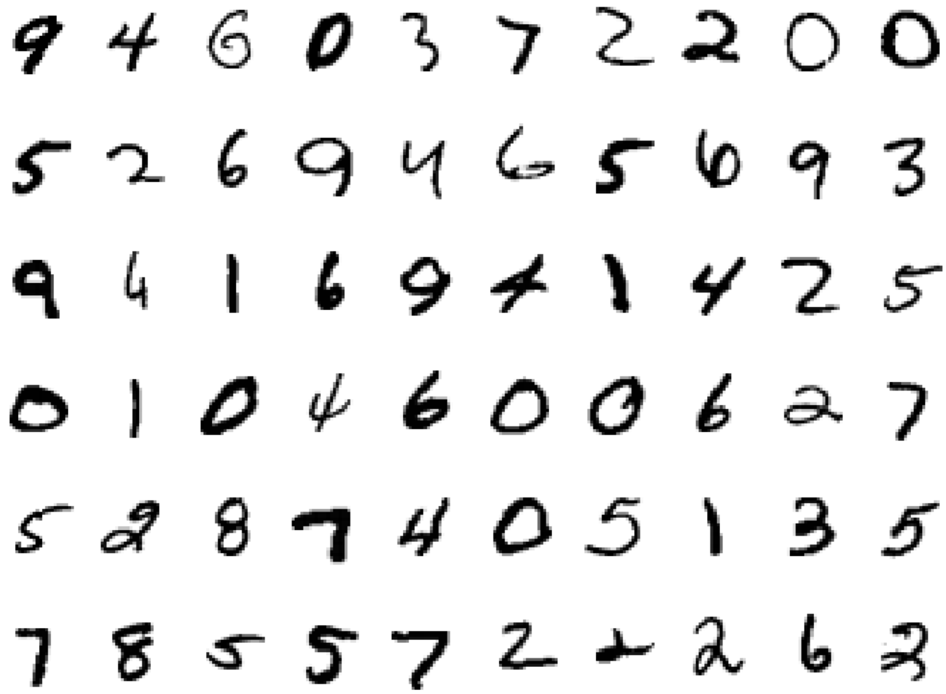
```
<class 'torch.Tensor'>
torch.Size([64, 1, 28, 28])
torch.Size([64])
```

4 Explore the processed data

[4]:
```
plt.imshow(images[0].numpy().squeeze(), cmap='gray_r'); # Change the index of␣
↪images[] to get different numbers
```



[7]:
```
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(images[index].numpy().squeeze(), cmap='gray_r')
```



5 Set up the neural network

[19]:
```
# Please change the runtime to GPU if you'd like to have some speed-up on Colab
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

### Layer details for the neural network
input_size = 784
hidden_sizes = [128, 64]
output_size = 10

### Build a feed-forward network
model = nn.Sequential(
    nn.Linear(input_size, hidden_sizes[0]), # Fully Connected Layer
    nn.ReLU(), # Activation
    nn.Linear(hidden_sizes[0], hidden_sizes[1]), # Fully Connected Layer
    nn.ReLU(), # Activation
    nn.Linear(hidden_sizes[1], output_size), # Fully Connected Layer
    nn.LogSoftmax(dim=1) # (Log) Softmax Layer: Output a probability distribution and apply log
)
print(model)
model.to(device)
```

```
Sequential(
  (0): Linear(in_features=784, out_features=128, bias=True)
  (1): ReLU()
  (2): Linear(in_features=128, out_features=64, bias=True)
  (3): ReLU()
  (4): Linear(in_features=64, out_features=10, bias=True)
  (5): LogSoftmax(dim=1)
)
```

[19]:
```
Sequential(
  (0): Linear(in_features=784, out_features=128, bias=True)
  (1): ReLU()
  (2): Linear(in_features=128, out_features=64, bias=True)
  (3): ReLU()
  (4): Linear(in_features=64, out_features=10, bias=True)
  (5): LogSoftmax(dim=1)
)
```

6 Set up the optimization model

[20]:
```
#@title Optimizer
lr = 3e-3 #@param {type: "number"}
optimizer = optim.Adam(model.parameters(), lr=lr) # Feel free to try out other
 ↪optimizers as you see fit!
```

7 Set up the loss function to optimize over

[21]:
```
time0 = time()
epochs = 15
criterion = nn.NLLLoss() # Negative log likelihood loss function is used
images, labels = next(iter(trainloader))
images = images.view(images.shape[0], -1).to(device)

logps = model(images) # Model spits out the log probability of image belonging
 ↪to different classes
loss = criterion(logps, labels.to(device))
```

8 Train the neural network

[22]:
```
for e in range(epochs):
    running_loss = 0
    for images, labels in trainloader:
        # Flatten MNIST images into a 784 long vector
        images = images.view(images.shape[0], -1).to(device)
        labels = labels.to(device)
```

```
        # Training pass
        optimizer.zero_grad()

        output = model(images).to(device)
        loss = criterion(output, labels)

        # backpropagation: calculate the gradient of the loss function w.r.t␣
↪model parameters
        loss.backward()

        # And optimizes its weights here
        optimizer.step()

        running_loss += loss.item()
    else:
        print("Epoch {} - Training loss: {}".format(e, running_loss/
↪len(trainloader)))
print("\nTraining Time (in minutes) =", (time()-time0)/60)
```

```
Epoch 0 - Training loss: 0.34144784285744495
Epoch 1 - Training loss: 0.1740617364394798
Epoch 2 - Training loss: 0.14824035517803863
Epoch 3 - Training loss: 0.12345252463868114
Epoch 4 - Training loss: 0.11443911887891987
Epoch 5 - Training loss: 0.10384408547542989
Epoch 6 - Training loss: 0.0993284839158282
Epoch 7 - Training loss: 0.09590731687776284
Epoch 8 - Training loss: 0.09157071839238622
Epoch 9 - Training loss: 0.08829374302188302
Epoch 10 - Training loss: 0.07945390294806393
Epoch 11 - Training loss: 0.07720497616464699
Epoch 12 - Training loss: 0.07428096306788524
Epoch 13 - Training loss: 0.07470227380091798
Epoch 14 - Training loss: 0.06939642548617939

Training Time (in minutes) = 3.182766552766164
```

9 Evaluate the trained neural network

[23]:
```
correct_count, all_count = 0, 0
for images, labels in valloader:
    for i in range(len(labels)):
        img = images[i].view(1, 784).to(device)
        labels = labels.to(device)
        # Forward pass only during evaluation
        with torch.no_grad():
```

```
            logps = model(img)

        # Output of the network are log-probabilities, need to take exponential␣
 ↪for probabilities
        ps = torch.exp(logps)
        probab = list(ps.cpu().numpy()[0])
        pred_label = probab.index(max(probab))
        true_label = labels.cpu().numpy()[i]
        if true_label == pred_label:
            correct_count += 1
        all_count += 1

print("Number Of Images Tested =", all_count)
print("\nModel Accuracy =", (correct_count/all_count))
```

```
Number Of Images Tested = 10000

Model Accuracy = 0.97
```

10 Predict using the trained neural network

[ ]:
```
def view_classify(img, ps):
    """ Function for viewing an image and it's predicted classes."""
    ps = ps.data.numpy().squeeze()

    fig, (ax1, ax2) = plt.subplots(figsize=(6,9), ncols=2)
    ax1.imshow(img.resize_(1, 28, 28).numpy().squeeze())
    ax1.axis('off')
    ax2.barh(np.arange(10), ps)
    ax2.set_aspect(0.1)
    ax2.set_yticks(np.arange(10))
    ax2.set_yticklabels(np.arange(10))
    ax2.set_title('Class Probability')
    ax2.set_xlim(0, 1.1)
    plt.tight_layout()
```

[ ]:
```
images, labels = next(iter(valloader))

img = images[0].view(1, 784).to(device)
# Turn off gradients
with torch.no_grad():
    logps = model(img)

# Output of the network are log-probabilities, need to take exponential for␣
 ↪probabilities
ps = torch.exp(logps)
probab = list(ps.cpu().numpy()[0])
```

```python
print("Predicted Digit =", probab.index(max(probab)))
view_classify(img.cpu().view(1, 28, 28), ps.cpu())
```