Part 2 - Neural Networks in PyTorch

July 10, 2018

1 Neural networks with PyTorch

Next I'll show you how to build a neural network with PyTorch.

```
In [2]: # Import things like usual

        %matplotlib inline
        %config InlineBackend.figure_format = 'retina'

        import numpy as np
        import torch

        import helper

        import matplotlib.pyplot as plt
        from torchvision import datasets, transforms
```

First up, we need to get our dataset. This is provided through the `torchvision` package. The code below will download the MNIST dataset, then create training and test datasets for us. Don't worry too much about the details here, you'll learn more about this later.

```
In [3]: # Define a transform to normalize the data
        transform = transforms.Compose([transforms.ToTensor(),
                                      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                                     ])
        # Download and load the training data
        trainset = datasets.MNIST('MNIST_data/', download=True, train=True, transform=transform)
        trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)

        # Download and load the test data
        testset = datasets.MNIST('MNIST_data/', download=True, train=False, transform=transform)
        testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=True)
```

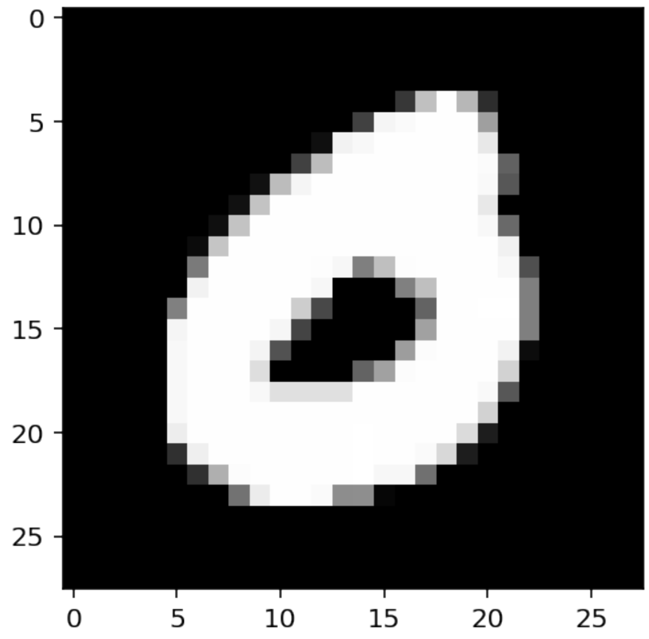
```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
```

```
Processing...
Done!
```

```
In [4]: dataiter = iter(trainloader)
        images, labels = dataiter.next()
```

We have the training data loaded into `trainloader` and we make that an iterator with `iter(trainloader)`. We'd use this to loop through the dataset for training, but here I'm just grabbing the first batch so we can check out the data. We can see below that `images` is just a tensor with size (64, 1, 28, 28). So, 64 images per batch, 1 color channel, and 28x28 images.

```
In [5]: plt.imshow(images[1].numpy().squeeze(), cmap='Greys_r');
```



1.1 Building networks with PyTorch

Here I'll use PyTorch to build a simple feedfoward network to classify the MNIST images. That is, the network will receive a digit image as input and predict the digit in the image.

To build a neural network with PyTorch, you use the `torch.nn` module. The network itself is a class inheriting from `torch.nn.Module`. You define each of the operations separately, like `nn.Linear(784, 128)` for a fully connected linear layer with 784 inputs and 128 units.

The class needs to include a `forward` method that implements the forward pass through the network. In this method, you pass some input tensor `x` through each of the operations you defined earlier. The `torch.nn` module also has functional equivalents for things like ReLUs in `torch.nn.functional`. This module is usually imported as `F`. Then to use a ReLU activation on some layer (which is just a tensor), you'd do `F.relu(x)`. Below are a few different commonly used activation functions.

So, for this network, I'll build it with three fully connected layers, then a softmax output for predicting classes. The softmax function is similar to the sigmoid in that it squashes inputs between 0 and 1, but it's also normalized so that all the values sum to one like a proper probability distribution.

```
In [6]: from torch import nn
        from torch import optim
        import torch.nn.functional as F
```

```
In [7]: class Network(nn.Module):
            def __init__(self):
                super().__init__()
                # Defining the layers, 128, 64, 10 units each
                self.fc1 = nn.Linear(784, 128)
                self.fc2 = nn.Linear(128, 64)
                # Output layer, 10 units - one for each digit
                self.fc3 = nn.Linear(64, 10)

            def forward(self, x):
                ''' Forward pass through the network, returns the output logits '''

                x = self.fc1(x)
                x = F.relu(x)
                x = self.fc2(x)
                x = F.relu(x)
                x = self.fc3(x)
                x = F.softmax(x, dim=1)

                return x

        model = Network()
        model
```

```
Out[7]: Network(
          (fc1): Linear(in_features=784, out_features=128, bias=True)
          (fc2): Linear(in_features=128, out_features=64, bias=True)
          (fc3): Linear(in_features=64, out_features=10, bias=True)
        )
```

1.1.1 Initializing weights and biases

The weights and such are automatically initialized for you, but it's possible to customize how they are initialized. The weights and biases are tensors attached to the layer you defined, you can get them with `model.fc1.weight` for instance.

```
In [8]: print(model.fc1.weight)
        print(model.fc1.bias)
```

```
Parameter containing:
tensor([[ 3.5128e-02,  3.2205e-02, -7.2352e-03,  ...,  9.3859e-03,
         -7.7923e-03, -3.1450e-02],
        [-1.4402e-02, -1.4887e-02, -2.0802e-02,  ...,  6.3884e-04,
         -3.2645e-02,  3.0526e-02],
        [ 1.5927e-02,  2.7244e-02,  5.8797e-03,  ...,  9.5567e-03,
          1.7324e-03, -8.4067e-03],
        ...,
        [ 5.3607e-03, -2.8351e-02,  2.0415e-02,  ..., -2.0463e-02,
         -1.2370e-02,  1.5519e-02],
        [ 2.2520e-02,  6.9887e-04, -5.6597e-03,  ..., -1.9946e-02,
         -2.7472e-02, -1.8087e-02],
        [-9.8169e-03,  1.9580e-02, -2.8101e-02,  ...,  2.5435e-02,
         -1.7955e-02,  1.0024e-02]])
Parameter containing:
tensor(1.00000e-02 *
       [-3.2312, -3.3512, -1.3517, -1.3628, -3.2653,  1.2228, -0.2015,
        -2.2442,  0.3072,  3.2574, -1.1044,  2.2227,  2.4939,  0.3231,
        -1.3076,  0.2025, -1.5382, -0.8615, -3.2184, -1.9797,  1.5546,
        -2.4076, -0.4572, -2.2412, -2.7144,  0.8122,  1.4973, -1.7377,
        -3.0005,  0.1208,  0.6050,  0.5452, -0.1868,  0.0946, -0.0151,
        -1.8808, -1.4777,  3.3272, -1.5540,  1.3796,  1.3225,  2.7141,
         2.8483, -0.3537,  2.9769,  0.1288,  1.9924, -2.2365, -1.2708,
        -0.6936, -2.2497,  2.7673,  1.7791, -2.5434, -0.4176, -2.0140,
         2.4538,  0.2761, -2.4422,  2.4025,  2.0505,  2.3220, -3.0806,
         0.5984,  2.4870,  3.0354, -2.8727,  2.8807, -3.3370,  0.0864,
        -2.1049,  1.0804, -1.4567, -0.6782,  1.9638,  0.0208,  2.6358,
         1.4173, -0.8713,  2.7019,  1.4336, -2.5052,  1.3364,  1.1603,
         2.8442,  0.6771, -0.4209,  0.3423,  2.4623, -1.3064,  1.8802,
        -0.4469,  2.0057, -0.4319,  1.4835,  1.1126,  1.1562,  1.4979,
        -2.8267, -3.1420, -0.5754, -0.0603, -2.2074, -1.9145,  3.5710,
        -0.4849, -2.6861, -2.8329, -0.3166, -3.2366, -0.5452,  1.0319,
        -2.0497, -0.2921, -3.3416,  2.8901, -1.4610, -0.5594,  2.3901,
        -0.1658, -3.2220,  1.1845,  2.1162,  0.6217,  0.0998, -1.2106,
         2.9742, -3.0559])
```

For custom initialization, we want to modify these tensors in place. These are actually autograd *Variables*, so we need to get back the actual tensors with `model.fc1.weight.data`. Once we have the tensors, we can fill them with zeros (for biases) or random normal values.

```
In [9]: # Set biases to all zeros
        model.fc1.bias.data.fill_(0)

Out[9]: tensor([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
                 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
                 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
                 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
                 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
                 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
                 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
                 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
                 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
                 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
                 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

```
In [10]: # sample from random normal with standard dev = 0.01
         model.fc1.weight.data.normal_(std=0.01)
```

```
Out[10]: tensor([[ 6.1899e-03, -2.6811e-03, -1.6023e-02,  ..., -2.0381e-03,
                  -3.7336e-03,  1.7640e-04],
                 [ 1.3443e-02,  5.1788e-03,  4.7388e-03,  ...,  1.2897e-02,
                   7.7145e-03,  3.4311e-03],
                 [ 6.5365e-03, -3.7268e-03,  5.9967e-04,  ..., -5.4286e-03,
                  -8.4894e-03, -4.1008e-03],
                 ...,
                 [ 1.7680e-03,  3.2592e-03,  1.3102e-03,  ..., -1.2746e-02,
                   8.9964e-03,  1.8795e-02],
                 [-1.7563e-02, -1.8444e-02, -1.6832e-02,  ..., -1.2667e-02,
                   2.5178e-03, -7.8623e-03],
                 [ 1.0049e-03, -3.6044e-03,  4.8563e-03,  ...,  1.0890e-03,
                  -5.1686e-03,  5.2565e-03]])
```

1.1.2 Forward pass

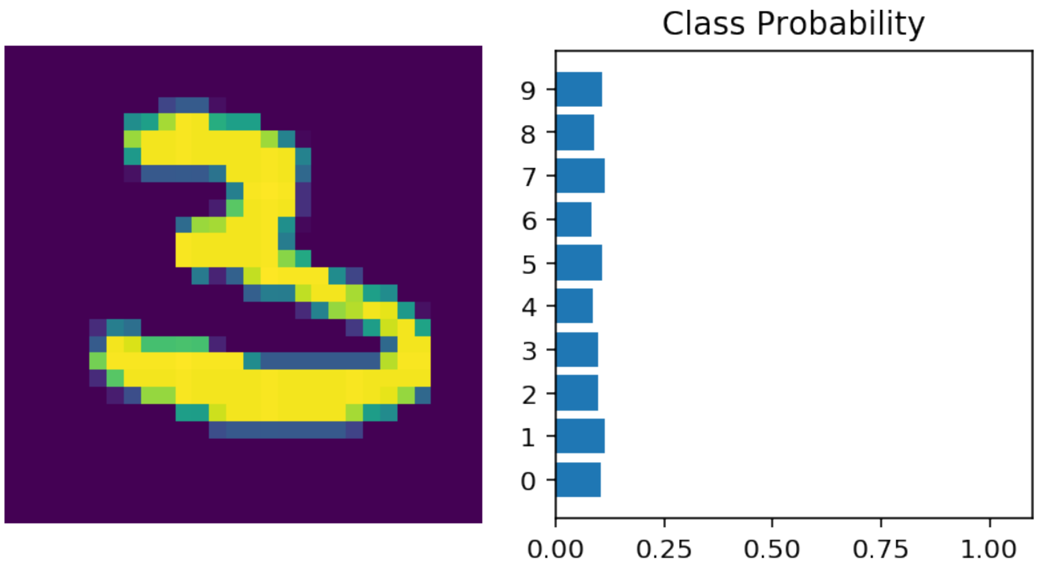
Now that we have a network, let's see what happens when we pass in an image. This is called the forward pass. We're going to convert the image data into a tensor, then pass it through the operations defined by the network architecture.

```
In [11]: # Grab some data
         dataiter = iter(trainloader)
         images, labels = dataiter.next()

         # Resize images into a 1D vector, new shape is (batch size, color channels, image pixel
         images.resize_(64, 1, 784)
         # or images.resize_(images.shape[0], 1, 784) to not automatically get batch size

         # Forward pass through the network
         img_idx = 0
         ps = model.forward(images[img_idx,:])

         img = images[img_idx]
         helper.view_classify(img.view(1, 28, 28), ps)
```



As you can see above, our network has basically no idea what this digit is. It's because we haven't trained it yet, all the weights are random!

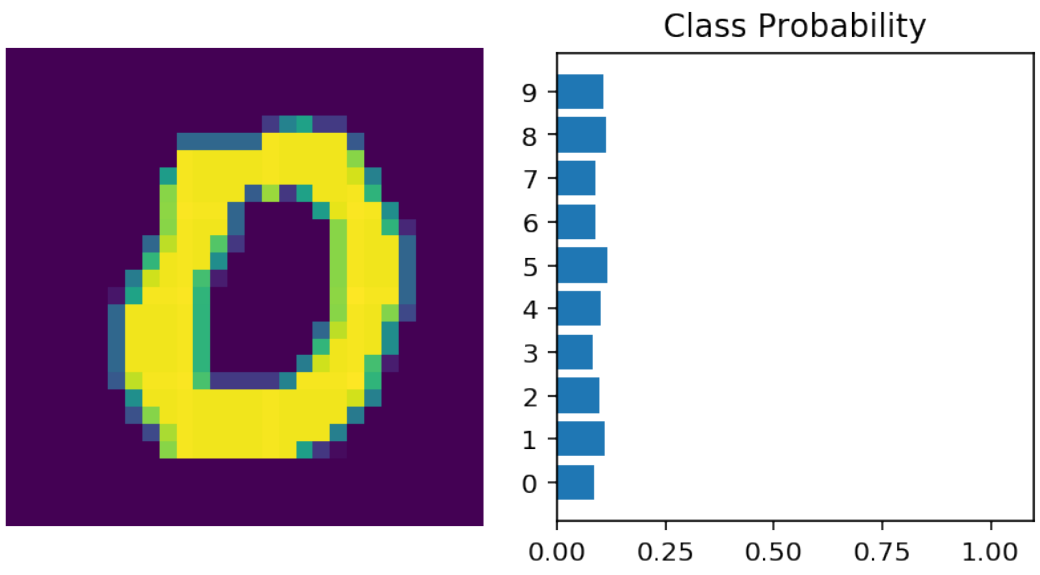
PyTorch provides a convenient way to build networks like this where a tensor is passed sequentially through operations, `nn.Sequential` (documentation). Using this to build the equivalent network:

```
In [12]: # Hyperparameters for our 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]),
                               nn.ReLU(),
                               nn.Linear(hidden_sizes[0], hidden_sizes[1]),
                               nn.ReLU(),
                               nn.Linear(hidden_sizes[1], output_size),
                               nn.Softmax(dim=1))
         print(model)

         # Forward pass through the network and display output
         images, labels = next(iter(trainloader))
         images.resize_(images.shape[0], 1, 784)
         ps = model.forward(images[0,:])
         helper.view_classify(images[0].view(1, 28, 28), ps)
```

```
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): Softmax()
)
```



You can also pass in an `OrderedDict` to name the individual layers and operations. Note that a dictionary keys must be unique, so *each operation must have a different name*.

```
In [13]: from collections import OrderedDict
         model = nn.Sequential(OrderedDict([
                               ('fc1', nn.Linear(input_size, hidden_sizes[0])),
                               ('relu1', nn.ReLU()),
                               ('fc2', nn.Linear(hidden_sizes[0], hidden_sizes[1])),
                               ('relu2', nn.ReLU()),
                               ('output', nn.Linear(hidden_sizes[1], output_size)),
                               ('softmax', nn.Softmax(dim=1))]))
         model
```

```
Out[13]: Sequential(
           (fc1): Linear(in_features=784, out_features=128, bias=True)
           (relu1): ReLU()
           (fc2): Linear(in_features=128, out_features=64, bias=True)
           (relu2): ReLU()
           (output): Linear(in_features=64, out_features=10, bias=True)
           (softmax): Softmax()
         )
```

Now it's your turn to build a simple network, use any method I've covered so far. In the next notebook, you'll learn how to train a network so it can make good predictions.

> **Exercise:** Build a network to classify the MNIST images with *three* hidden layers. Use 400 units in the first hidden layer, 200 units in the second layer, and 100 units in the third layer. Each hidden layer should have a ReLU activation function, and use softmax on the output layer.

```
In [19]: ## TODO: Your network here
         input_size = 784
         hidden_sizes = [400,200,100]
         output_size = 10
         from collections import OrderedDict
         model = nn.Sequential(OrderedDict([
                     ('fc1', nn.Linear(input_size, hidden_sizes[0])),
                     ('relu1', nn.ReLU()),
                     ('fc2',nn.Linear(hidden_sizes[0],hidden_sizes[1])),
                     ('relu2',nn.ReLU()),
                     ('fc3', nn.Linear(hidden_sizes[1],hidden_sizes[2])),
                     ('relu3', nn.ReLU()),
                     ('fc4', nn.Linear(hidden_sizes[2],output_size)),
                     ('relu4', nn.ReLU())
                     ]))
         print(model)
```

```
Sequential(
  (fc1): Linear(in_features=784, out_features=400, bias=True)
  (relu1): ReLU()
  (fc2): Linear(in_features=400, out_features=200, bias=True)
  (relu2): ReLU()
  (fc3): Linear(in_features=200, out_features=100, bias=True)
  (relu3): ReLU()
  (fc4): Linear(in_features=100, out_features=10, bias=True)
  (relu4): ReLU()
)
```

```
In [17]: ## Run this cell with your model to make sure it works ##
         # Forward pass through the network and display output
         dataiter = iter(trainloader)
         images, labels = dataiter.next()

         # Resize images into a 1D vector, new shape is (batch size, color channels, image pixel
         images.resize_(64, 1, 400)
         # or images.resize_(images.shape[0], 1, 784) to not automatically get batch size

         # Forward pass through the network
         img_idx = 0
         ps = model.forward(images[img_idx,:])
```

```
 img = images[img_idx]
 helper.view_classify(img.view(1, 28, 28), ps)
 images, labels = next(iter(trainloader))
 images.resize_(images.shape[0], 1, 784)
 ps = model.forward(images[0,:])
 helper.view_classify(images[0].view(1, 28, 28), ps)
```

```
    ---------------------------------------------------------------------------

    RuntimeError                              Traceback (most recent call last)

    <ipython-input-17-2347ae27e551> in <module>()
     13
     14 img = images[img_idx]
---> 15 helper.view_classify(img.view(1, 28, 28), ps)
     16 images, labels = next(iter(trainloader))
     17 images.resize_(images.shape[0], 1, 400)


    RuntimeError: invalid argument 2: size '[1 x 28 x 28]' is invalid for input with 400 ele
```