Part 2 - Neural Networks in PyTorch (Exercises)

March 8, 2020

1 Neural networks with PyTorch

Deep learning networks tend to be massive with dozens or hundreds of layers, that's where the term "deep" comes from. You can build one of these deep networks using only weight matrices as we did in the previous notebook, but in general it's very cumbersome and difficult to implement. PyTorch has a nice module `nn` that provides a nice way to efficiently build large neural networks.

In [1]:
```
# Import necessary packages

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

import numpy as np
import torch

import helper

import matplotlib.pyplot as plt
```

Now we're going to build a larger network that can solve a (formerly) difficult problem, identifying text in an image. Here we'll use the MNIST dataset which consists of greyscale handwritten digits. Each image is 28x28 pixels, you can see a sample below

Our goal is to build a neural network that can take one of these images and predict the digit in the image.

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 [2]:
```
### Run this cell

from torchvision import datasets, transforms

# 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('~/.pytorch/MNIST_data/', download=True, train=True, transform
        trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
```

We have the training data loaded into `trainloader` and we make that an iterator with `iter(trainloader)`. Later, we'll use this to loop through the dataset for training, like

```
for image, label in trainloader:
    ## do things with images and labels
```

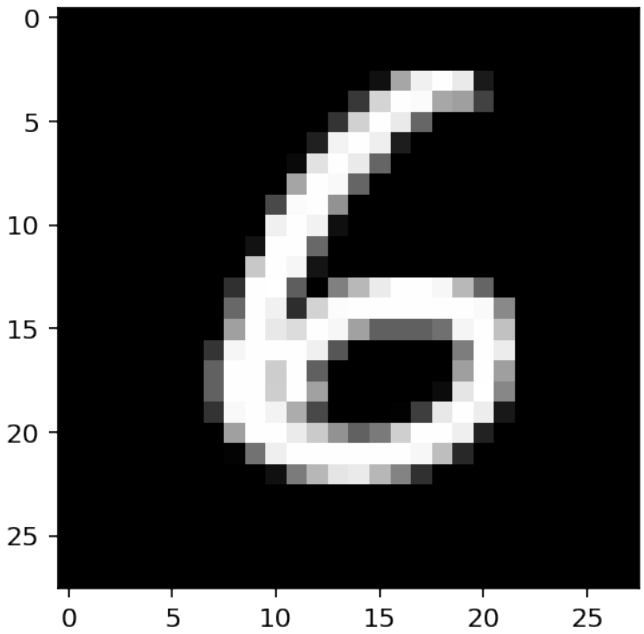
You'll notice I created the `trainloader` with a batch size of 64, and `shuffle=True`. The batch size is the number of images we get in one iteration from the data loader and pass through our network, often called a *batch*. And `shuffle=True` tells it to shuffle the dataset every time we start going through the data loader again. 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 [3]: dataiter = iter(trainloader)
        images, labels = dataiter.next()
        print(type(images))
        print(images.shape)
        print(labels.shape)
```

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

This is what one of the images looks like.

```
In [4]: plt.imshow(images[1].numpy().squeeze(), cmap='Greys_r');
```



First, let's try to build a simple network for this dataset using weight matrices and matrix multiplications. Then, we'll see how to do it using PyTorch's `nn` module which provides a much more convenient and powerful method for defining network architectures.

The networks you've seen so far are called *fully-connected* or *dense* networks. Each unit in one layer is connected to each unit in the next layer. In fully-connected networks, the input to each layer must be a one-dimensional vector (which can be stacked into a 2D tensor as a batch of multiple examples). However, our images are 28x28 2D tensors, so we need to convert them into 1D vectors. Thinking about sizes, we need to convert the batch of images with shape `(64, 1, 28, 28)` to a have a shape of `(64, 784)`, 784 is 28 times 28. This is typically called *flattening*, we flattened the 2D images into 1D vectors.

Previously you built a network with one output unit. Here we need 10 output units, one for each digit. We want our network to predict the digit shown in an image, so what we'll do is calculate probabilities that the image is of any one digit or class. This ends up being a discrete probability distribution over the classes (digits) that tells us the most likely class for the image. That means we need 10 output units for the 10 classes (digits). We'll see how to convert the network output into a probability distribution next.

> **Exercise:** Flatten the batch of images `images`. Then build a multi-layer network with 784 input units, 256 hidden units, and 10 output units using random tensors for the weights and biases. For now, use a sigmoid activation for the hidden layer. Leave the output layer without an activation, we'll add one that gives us a probability distribution next.

In [5]:
```
# Flat the images
images_flat = images.view(64, -1)

n_input = images_flat.shape[1]
n_hidden = 256
n_output = 10

W_h = torch.randn(n_input, n_hidden)
W_o = torch.randn(n_hidden, n_output)

B_h = torch.randn(1, n_hidden)
B_o = torch.randn(1, n_output)

def activation(x):
    return 1/(1 + torch.exp(-x))

hidden = activation(torch.mm(images_flat, W_h) + B_h)
out = activation(torch.mm(hidden, W_o) + B_o)

print(out.shape)
# output of your network, should have shape (64,10)
```

```
torch.Size([64, 10])
```

Now we have 10 outputs for our network. We want to pass in an image to our network and get out a probability distribution over the classes that tells us the likely class(es) the image belongs to. Something that looks like this:

Here we see that the probability for each class is roughly the same. This is representing an untrained network, it hasn't seen any data yet so it just returns a uniform distribution with equal probabilities for each class.

To calculate this probability distribution, we often use the **softmax** function. Mathematically this looks like

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_k^K e^{x_k}}$$

What this does is squish each input x_i between 0 and 1 and normalizes the values to give you a proper probability distribution where the probabilites sum up to one.

> **Exercise:** Implement a function `softmax` that performs the softmax calculation and returns probability distributions for each example in the batch. Note that you'll need to pay attention to the shapes when doing this. If you have a tensor `a` with shape `(64, 10)` and a tensor `b` with shape `(64,)`, doing `a/b` will give you an error because PyTorch will try to do the division across the columns (called broadcasting) but you'll get a size mismatch. The way to think about this is for each of the 64 examples, you only want to divide by one value, the sum in the denominator. So you need `b` to have a shape of `(64, 1)`. This way PyTorch will divide the 10 values in each row of `a` by the one value in each row of `b`. Pay attention to how you take the sum as well. You'll need to define the `dim` keyword in `torch.sum`. Setting `dim=0` takes the sum across the rows while `dim=1` takes the sum across the columns.

```
In [6]: def softmax(x):
            ## TODO: Implement the softmax function here
            x = torch.exp(x)
            return x / torch.sum(x, dim = 1).view(64, 1)

        # Here, out should be the output of the network in the previous excercise with shape (64

        probabilities = softmax(out)

        # Does it have the right shape? Should be (64, 10)
        print(probabilities.shape)
        # Does it sum to 1?
        print(probabilities.sum(dim=1))
```

```
torch.Size([64, 10])
tensor([ 1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
         1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
         1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
         1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
         1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
         1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
         1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
         1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
```

```
        1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
        1.0000])
```

1.1 Building networks with PyTorch

PyTorch provides a module `nn` that makes building networks much simpler. Here I'll show you how to build the same one as above with 784 inputs, 256 hidden units, 10 output units and a softmax output.

```
In [7]: from torch import nn

In [8]: class Network(nn.Module):
            def __init__(self):
                super().__init__()

                # Inputs to hidden layer linear transformation
                self.hidden = nn.Linear(784, 256)
                # Output layer, 10 units - one for each digit
                self.output = nn.Linear(256, 10)

                # Define sigmoid activation and softmax output
                self.sigmoid = nn.Sigmoid()
                self.softmax = nn.Softmax(dim=1)

            def forward(self, x):
                # Pass the input tensor through each of our operations
                x = self.hidden(x)
                x = self.sigmoid(x)
                x = self.output(x)
                x = self.softmax(x)

                return x
```

Let's go through this bit by bit.

```
class Network(nn.Module):
```

Here we're inheriting from `nn.Module`. Combined with `super().__init__()` this creates a class that tracks the architecture and provides a lot of useful methods and attributes. It is mandatory to inherit from `nn.Module` when you're creating a class for your network. The name of the class itself can be anything.

```
self.hidden = nn.Linear(784, 256)
```

This line creates a module for a linear transformation, $x\mathbf{W} + b$, with 784 inputs and 256 outputs and assigns it to `self.hidden`. The module automatically creates the weight and bias tensors which we'll use in the `forward` method. You can access the weight and bias tensors once the network (`net`) is created with `net.hidden.weight` and `net.hidden.bias`.

```
self.output = nn.Linear(256, 10)
```

Similarly, this creates another linear transformation with 256 inputs and 10 outputs.

```
self.sigmoid = nn.Sigmoid()
self.softmax = nn.Softmax(dim=1)
```

Here I defined operations for the sigmoid activation and softmax output. Setting `dim=1` in `nn.Softmax(dim=1)` calculates softmax across the columns.

```
def forward(self, x):
```

PyTorch networks created with `nn.Module` must have a `forward` method defined. It takes in a tensor `x` and passes it through the operations you defined in the `__init__` method.

```
x = self.hidden(x)
x = self.sigmoid(x)
x = self.output(x)
x = self.softmax(x)
```

Here the input tensor `x` is passed through each operation a reassigned to `x`. We can see that the input tensor goes through the hidden layer, then a sigmoid function, then the output layer, and finally the softmax function. It doesn't matter what you name the variables here, as long as the inputs and outputs of the operations match the network architecture you want to build. The order in which you define things in the `__init__` method doesn't matter, but you'll need to sequence the operations correctly in the `forward` method.

Now we can create a `Network` object.

```
In [9]: # Create the network and look at it's text representation
        model = Network()
        model
```

```
Out[9]: Network(
          (hidden): Linear(in_features=784, out_features=256, bias=True)
          (output): Linear(in_features=256, out_features=10, bias=True)
          (sigmoid): Sigmoid()
          (softmax): Softmax()
        )
```

You can define the network somewhat more concisely and clearly using the `torch.nn.functional` module. This is the most common way you'll see networks defined as many operations are simple element-wise functions. We normally import this module as `F`, `import torch.nn.functional as F`.

```
In [10]: import torch.nn.functional as F

         class Network(nn.Module):
             def __init__(self):
                 super().__init__()
                 # Inputs to hidden layer linear transformation
```

```
        self.hidden = nn.Linear(784, 256)
        # Output layer, 10 units - one for each digit
        self.output = nn.Linear(256, 10)

    def forward(self, x):
        # Hidden layer with sigmoid activation
        x = F.sigmoid(self.hidden(x))
        # Output layer with softmax activation
        x = F.softmax(self.output(x), dim=1)

        return x
```

1.1.1 Activation functions

So far we've only been looking at the softmax activation, but in general any function can be used as an activation function. The only requirement is that for a network to approximate a non-linear function, the activation functions must be non-linear. Here are a few more examples of common activation functions: Tanh (hyperbolic tangent), and ReLU (rectified linear unit).

In practice, the ReLU function is used almost exclusively as the activation function for hidden layers.

1.1.2 Your Turn to Build a Network

> **Exercise:** Create a network with 784 input units, a hidden layer with 128 units and a ReLU activation, then a hidden layer with 64 units and a ReLU activation, and finally an output layer with a softmax activation as shown above. You can use a ReLU activation with the `nn.ReLU` module or `F.relu` function.

In [11]:
```
## Your solution here
import torch.nn.functional as F

class Network(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden_1 = nn.Linear(784, 128)
        self.hidden_2 = nn.Linear(128, 64)
        self.output = nn.Linear(64, 10)

    def forward(self, x):
        x = F.relu(self.hidden_1(x))
        x = F.relu(self.hidden_2(x))
        x = F.softmax(self.output(x), dim=1)

        return x

model = Network()
```

1.1.3 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 [12]: print(model.hidden_1.weight)
         print(model.hidden_1.bias)

Parameter containing:
tensor([[ 3.2547e-02, -2.3442e-02, -3.0366e-02,  ..., -1.5540e-02,
         -5.7197e-03, -2.2755e-02],
        [ 1.9740e-02, -1.7644e-02, -3.9964e-03,  ..., -8.3833e-03,
         -1.0041e-02, -2.1259e-02],
        [-2.7916e-03, -6.7121e-03,  3.9643e-03,  ...,  2.5153e-02,
         -1.4185e-02, -1.7798e-02],
        ...,
        [ 8.4616e-03, -2.4144e-02,  5.2909e-03,  ...,  7.3592e-03,
          1.8891e-02,  3.4893e-02],
        [ 1.0300e-02,  2.0536e-02,  1.3138e-02,  ..., -2.5804e-03,
          1.2229e-03,  9.8737e-03],
        [ 1.6079e-02,  2.8989e-02, -7.7481e-04,  ..., -5.2841e-03,
         -2.0109e-04, -3.3549e-02]])
Parameter containing:
tensor(1.00000e-02 *
       [-0.6560,  0.8744,  0.2539,  0.1741,  1.2834, -0.1126,  3.0752,
         0.7853,  0.5586, -0.6252,  3.1989, -1.3525, -0.0640,  2.6603,
         1.5985,  2.8866,  3.4839, -3.5044,  0.1755,  0.9407, -0.6223,
         2.9342,  2.3080, -1.1260,  3.2789, -2.6000,  2.6305, -2.0217,
        -1.4025,  0.6803,  2.9896, -1.9651,  0.1828, -3.1974, -3.2910,
         1.0269,  2.4185,  1.6622,  3.2212,  0.5092,  0.5358, -1.4555,
        -0.2181, -2.1863,  0.8355, -3.3001,  0.0762, -0.2267, -0.5457,
        -3.5609,  2.9172,  2.7380,  0.9781, -1.2421,  3.1336,  2.8256,
        -2.9916, -3.2993,  2.1650,  0.2262, -0.8384, -2.9713, -2.2829,
         2.0000,  0.8861, -1.8200, -2.0566,  1.1122,  3.1335, -3.4015,
        -2.2754, -0.8441,  1.6125, -2.2318, -1.2370,  2.6179,  1.3515,
         2.1207,  3.1279, -1.5742,  3.5565, -1.8192, -1.3835,  1.8382,
        -0.5851, -3.3667, -3.2408, -1.0593, -0.6698, -0.7117,  3.2225,
         2.9935, -2.1400, -3.4322, -2.5599,  3.3818, -3.0663,  0.9342,
         1.1863,  1.4636, -1.5357,  3.1873,  2.5051,  2.6337, -0.9426,
         3.2981, -3.3948, -2.4429, -0.0028,  2.5251, -2.3306,  0.8901,
         2.7347, -0.5424,  1.9732, -2.4979, -0.2914,  0.8885, -1.0788,
         1.4585,  1.1415, -1.2800, -2.0980,  0.1833,  2.3563,  1.7253,
         2.2748, -1.9018])
```

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 [13]:
```
# Set biases to all zeros
model.hidden_1.bias.data.fill_(0)
```

Out[13]:
```
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 [14]:
```
# sample from random normal with standard dev = 0.01
model.hidden_1.weight.data.normal_(std=0.01)
```

Out[14]:
```
tensor([[-7.8110e-03,  4.3386e-03,  2.1198e-03,  ...,  1.5720e-02,
          4.3883e-03,  1.1765e-02],
        [-1.6203e-03, -1.7046e-02, -1.7431e-02,  ...,  8.7850e-03,
         -5.8042e-03, -6.3292e-03],
        [-1.5663e-02,  1.6760e-02,  5.7155e-03,  ..., -1.5894e-02,
         -6.7197e-03, -2.5968e-03],
        ...,
        [-2.7199e-03,  4.6677e-03,  3.7415e-03,  ...,  1.6500e-03,
         -1.1481e-02,  3.4652e-03],
        [-4.2796e-03, -1.2359e-02,  1.2893e-02,  ..., -3.2438e-02,
         -1.0866e-03,  5.4785e-05],
        [ 4.2730e-03, -3.8448e-03, -3.6371e-03,  ...,  1.1532e-02,
         -7.8797e-03, -1.8778e-03]])
```

1.1.4 Forward pass

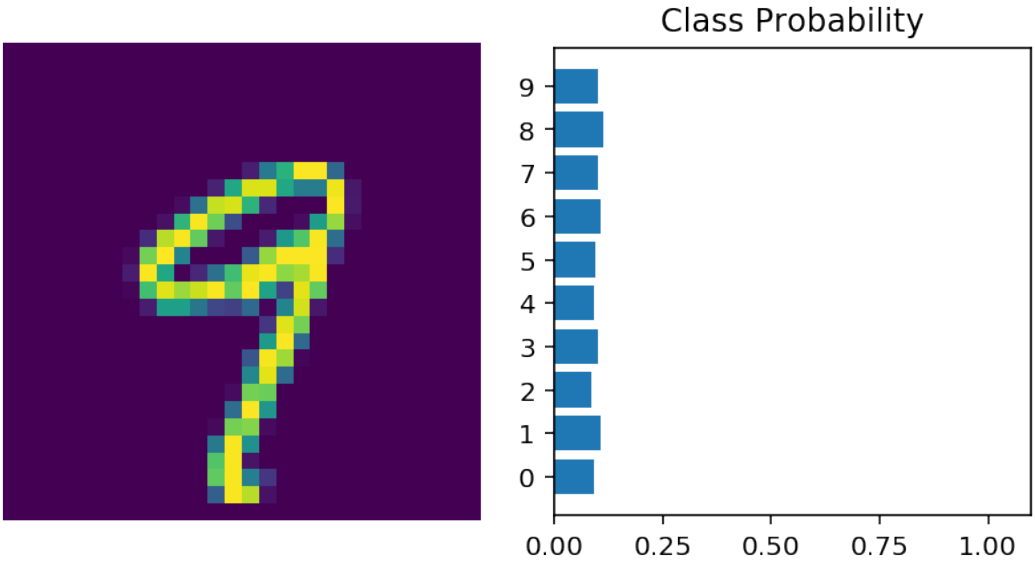
Now that we have a network, let's see what happens when we pass in an image.

In [15]:
```
# 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 automatically get batch size

# Forward pass through the network
img_idx = 1
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!

1.1.5 Using `nn.Sequential`

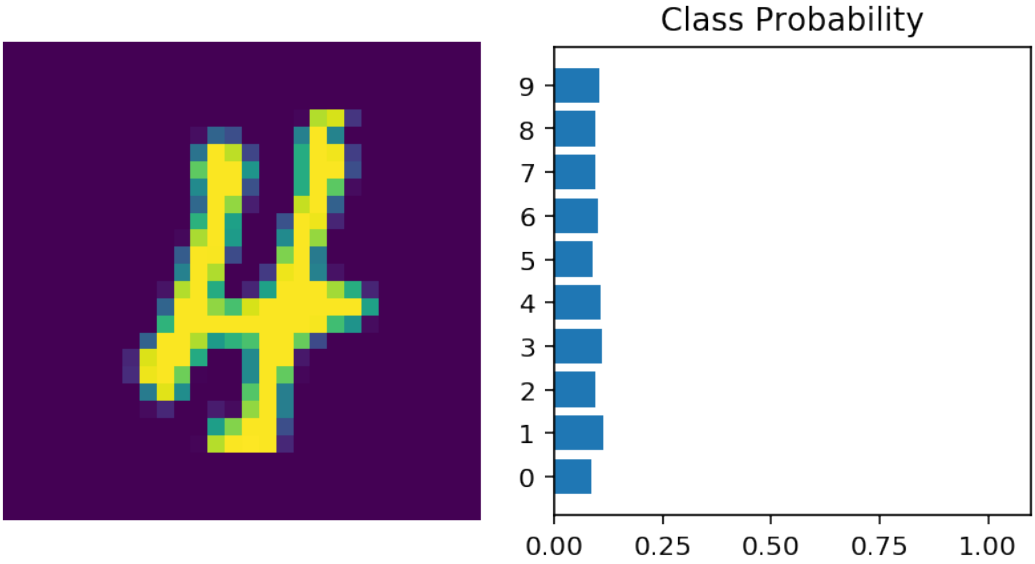
PyTorch provides a convenient way to build networks like this where a tensor is passed sequentially through operations, `nn.Sequential` (documentation). Using this to build the equivalent network:

```
In [16]: # 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()
)
```



Here our model is the same as before: 784 input units, a hidden layer with 128 units, ReLU activation, 64 unit hidden layer, another ReLU, then the output layer with 10 units, and the softmax output.

The operations are availble by passing in the appropriate index. For example, if you want to get first Linear operation and look at the weights, you'd use `model[0]`.

```
In [17]: print(model[0])
         model[0].weight

Linear(in_features=784, out_features=128, bias=True)


Out[17]: Parameter containing:
         tensor([[-1.5028e-02,  3.0927e-02, -2.9518e-02,  ..., -7.5794e-03,
                  -3.2880e-02,  1.5428e-02],
                 [-7.2033e-03,  1.8922e-02, -1.6462e-02,  ..., -2.4264e-02,
                  -1.0872e-02,  9.2309e-03],
                 [-3.1992e-03,  3.6969e-03, -1.8940e-02,  ...,  6.8256e-03,
                   2.0601e-02,  4.7628e-03],
```

```
                ...,
                [ 2.7923e-02,  3.1391e-03,  1.5700e-02,  ...,  2.3658e-02,
                  7.1466e-03,  3.1948e-02],
                [ 5.1726e-03,  1.4160e-02,  2.1933e-02,  ...,  1.8854e-02,
                  3.1966e-02, -1.8665e-02],
                [-2.8864e-03, -7.1470e-03, -2.8318e-02,  ..., -1.5876e-02,
                  7.5144e-03, -3.3174e-02]])
```

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

```
In [18]: 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[18]: 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 you can access layers either by integer or the name

```
In [19]: print(model[0])
         print(model.fc1)

Linear(in_features=784, out_features=128, bias=True)
Linear(in_features=784, out_features=128, bias=True)
```

In the next notebook, we'll see how we can train a neural network to accuractly predict the numbers appearing in the MNIST images.