Neural Networks in PyTorch

In this notebook we will learn how to build deep neural networks in PyTorch. In the last notebook, `1-Tensors-in-PyTorch.ipynb`, you saw how we can calculate the output from a single neural network using tensors and matrix multiplication. However, for deep neural networks it's not very practical to manually compute output from every single hidden layer. As a result, PyTorch has a nice module `nn`, which contains a lot of classes and functions to build deep neural networks very efficiently.

To start, we will be using a dataset called MNIST, which consists of greyscale handwritten digits, ranging from 0 to 9. Each image is 28 x 28 pixels and you can see the sample below:

The goal of the neural network is to identify what the number is in each of these images. To accomplish this goal, we will show our network a set of images and correct labels associated with these images, and the network will learn to determine what the correct label is for a new image. The MNIST dataset is available in the `torchvision` package:

In [1]:
```
# As usual, we import necessary packages:
import numpy as np
import torch
from torchvision import datasets, transforms
import helper
from utils import activation
import matplotlib.pyplot as plt
```

In [2]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
```

In [3]:
```
# 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 = transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size = 64, shuffle = Tru
e)
```

This creates an object called `trainloader`, which we can make an iterator with `iter(trainloader)`. Alternatively, we can also use a for loop to get the data:

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

You'll see that we created `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 it through the network. The option `shuffle = True` is a useful command to tell the algorithm to shuffle the dataset everytime we start going through the data loader again. Though here we grab only the first batch to check the data.

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

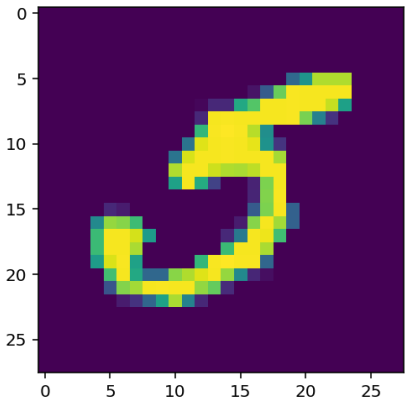
```
<class 'torch.Tensor'>
torch.Size([64, 1, 28, 28])
torch.Size([64])
```

We see that `images` is a tensor with shape `(64, 1, 28, 28)`, which corresponds to 64 images in a batch, 1 color channel, and 28 x 28 images. `labels` is a vector with 64 elements representing each of the 64 labels.

To check the images, we plot one of them. Because `images` is a torch tensor, we have to convert it into a numpy array using `images[0].numpy()`:

In [53]:
```
plt.imshow(images[8].numpy().squeeze())
```

Out[53]:
```
<matplotlib.image.AxesImage at 0x123caf9e8>
```



Now, let's try to build a simple neural network for this dataset using weight matrices and matrix multiplication. Then we'll see how PyTorch's `nn` module provides much more powerful framework for defining network architectures.

The networks we've seen so far are *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 1D vector, which can be stacked into a 2D tensor as a batch with multiple examples. Here, our input images are 2D 28 x 28 tensors and therefore, we need to convert them into 1D vectors. Thinking about sizes, we need to convert the batch with shape `(64, 1, 28, 28)` into a vector with shape `(64, 784)`. This operation is called *flattening*, since we flatten a 2D tensor into a 1D vector.

In the previous notebook, we built a network with one output. However, here we want to have 10 output units, one for eaxh digit. We want our network to predict the digit shown in an image, so what we'll do is to 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]:
```
# TODO: Implement a simple neural network on a MNIST batch
## Your solution, output of your network, should have shape (64,10):
torch.manual_seed(1)
features = images.view(images.shape[0], 784)
n_hidden = 256
n_output = 10
W1 = torch.randn(784, n_hidden)
W2 = torch.randn(n_hidden, n_output)
B1 = torch.randn(1, n_hidden)
B2 = torch.randn(1, n_output)
```

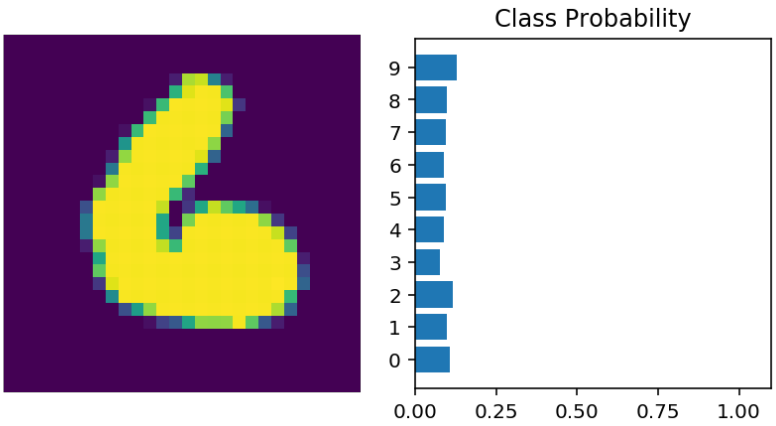
In [6]:
```
# Check the shapes:
print(features.shape)
print(W1.shape)
print(B1.shape)
print(W2.shape)
print(B2.shape)
```

```
torch.Size([64, 784])
torch.Size([784, 256])
torch.Size([1, 256])
torch.Size([256, 10])
torch.Size([1, 10])
```

In [7]:
```
hidden = activation(torch.add(torch.mm(features, W1), B1))
print(hidden.shape)
out = torch.add(torch.mm(hidden, W2), B2)
print(out.shape)
```

```
torch.Size([64, 256])
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 (https://en.wikipedia.org/wiki/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 [57]: out[0]
```

```
Out[57]: tensor([ -5.0436,  -8.1012, -11.0228,  -2.3096,   9.6730,  30.8278,  -9.1326,
                    3.1400,  -7.8228,  -5.8484])
```

In [8]:
```
def softmax(x):
    ## TODO: Implement the softmax function here
    batch_size = 64
    x = torch.exp(x)/(torch.sum(torch.exp(x), dim = 1).view(batch_size, 1))
    return x

# Here, out should be the output of the network in the previous excercise with sha
pe (64,10)
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])
```

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 [9]:
```
from torch import nn
```

In [10]:
```
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
```

We can go through this step-by-step. Here we are creating a new class:

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

and we need to subclass it from `nn.Module` so that we are inheriting from `nn.Module`. Combined with `super().__init__()`, PyTorch 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.

```
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 [11]:
```
# Create the network and look at its text representation
model = Network()
model
```

Out[11]:
```
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()
)
```

We can also use functional definitions using `torch.nn.functional()` module. This is a cleaner and the most commonly used way to define the networks because many operations are simple element-wise functions. We normally import this module as `F`: `import torch.nn.functional as F`.

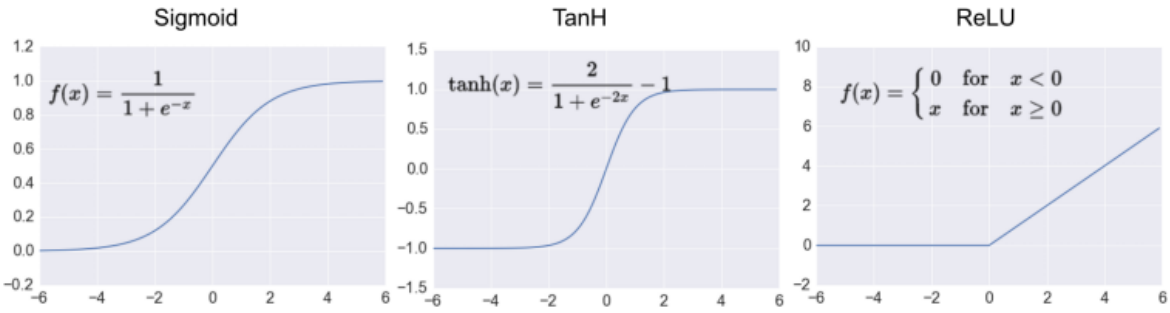
In [12]:
```
import torch.nn.functional as F
class Network(nn.Module):
    def __init__(self):
        super().__init__()
        # Inputs to the first hidden layer
        self.hidden = nn.Linear(784, 256)
        # Output layer with 10 units
        self.output = nn.Linear(256, 10)

    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, dim = 1)

        return x
```

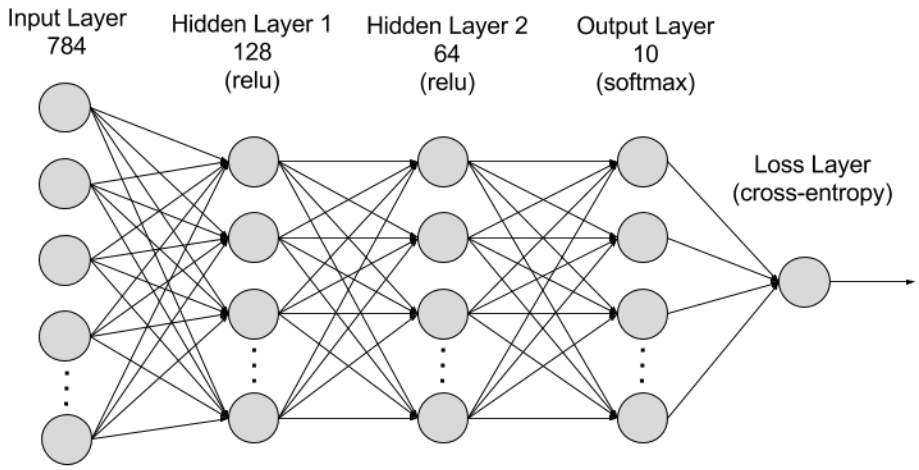
Activation functions

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.

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 [44]:
```
# TODO: build a network using torch.nn.functional module
import torch.nn.functional as F

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.output = nn.Linear(64, 10)

    def forward(self, x):
            ''' Forward pass through the network, returns the output logits '''
            # First hidden layer with sigmoid activation:
            x = self.fc1(x)
            x = F.relu(x)
            # Second hidden layer with sigmoid activation:
            x = self.fc2(x)
            x = F.relu(x)
            # Output layer with softmax activation:
            x = self.output(x)
            x = F.softmax(x, dim = 1)

            return x
```

In [45]:
```
# We can get the network structure
model = Network()
model
```

Out[45]:
```
Network(
  (fc1): Linear(in_features=784, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=64, bias=True)
  (output): Linear(in_features=64, out_features=10, bias=True)
)
```

Check: That makes sense, we defined 3 layers: fully-connected # 1 (`fc1`), fully-connected # 2 (`fc2`), and output layer (`output`).

Initializing weights and biases

The weights and biases 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 [65]:
```
print(model.fc1.weight.shape)
print(model.fc2.weight.shape)
print(model.output.weight.shape)
```

```
torch.Size([128, 784])
torch.Size([64, 128])
torch.Size([10, 64])
```

In [66]:
```
print(model.fc1.bias.shape)
print(model.fc2.bias.shape)
print(model.output.bias.shape)
```

```
torch.Size([128])
torch.Size([64])
torch.Size([10])
```

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 [67]:
```
model.fc2.weight.data
```

Out[67]:
```
tensor([[-0.0142,  0.0501,  0.0040,  ...,  0.0722,  0.0058, -0.0231],
        [ 0.0684, -0.0743, -0.0301,  ..., -0.0321, -0.0808, -0.0637],
        [-0.0865, -0.0652, -0.0806,  ...,  0.0473,  0.0393,  0.0336],
        ...,
        [ 0.0740,  0.0832,  0.0327,  ..., -0.0711, -0.0335, -0.0778],
        [ 0.0685, -0.0759, -0.0568,  ...,  0.0437, -0.0381, -0.0737],
        [-0.0777,  0.0041,  0.0192,  ...,  0.0203,  0.0383,  0.0696]])
```

In [68]:
```
model.fc2.bias.data
```

Out[68]:
```
tensor([ 0.0703,  0.0520, -0.0508,  0.0155, -0.0141, -0.0245,  0.0336, -0.0121,
         0.0739, -0.0231,  0.0527, -0.0437,  0.0343,  0.0418,  0.0267,  0.0842,
        -0.0499,  0.0548,  0.0437, -0.0550, -0.0767,  0.0341,  0.0600, -0.0445,
         0.0204,  0.0191, -0.0406, -0.0568,  0.0116,  0.0003,  0.0523, -0.0627,
         0.0018,  0.0570, -0.0254,  0.0789,  0.0455,  0.0881,  0.0479, -0.0813,
         0.0868, -0.0350,  0.0721,  0.0112, -0.0501,  0.0091,  0.0272,  0.0715,
         0.0051,  0.0079, -0.0766,  0.0451, -0.0565, -0.0384, -0.0698, -0.0796,
        -0.0337, -0.0730,  0.0003,  0.0655, -0.0771,  0.0362, -0.0323,  0.0237])
```

In [69]:
```
# Set biases to all zeros
model.fc2.bias.data.fill_(0)
```

Out[69]:
```
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.])
```

In [70]:
```
# model.fc2.bias.data.fill(0) won't work:
model.fc2.bias.data.fill(0)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-70-87b4fcde27bd> in <module>
      1 # model.fc2.bias.data.fill(0) won't work:
----> 2 model.fc2.bias.data.fill(0)

AttributeError: 'Tensor' object has no attribute 'fill'
```

In [71]:
```
print(model.fc2.bias)
```

```
Parameter containing:
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.], requires_grad=True)
```

In [72]:
```
# Set biases to all zeros
model.fc1.bias.data.normal_(mean = 1, std = 0.5)
```

Out[72]:
```
tensor([ 0.8509,  0.4905,  1.2403,  1.1724,  1.3912,  0.7141,  1.0232,  1.0295,
         0.7436,  1.0453,  0.9881,  0.2450,  0.9195,  1.8810,  1.3202,  0.9742,
         0.8750,  1.0869,  1.4423,  0.5073,  0.6316,  0.8815,  0.5175,  1.1267,
         0.9594,  0.8432,  0.9168,  1.7915,  1.3254,  1.1141,  1.4747,  0.5092,
         1.3403,  0.7390,  1.2174,  1.2195,  1.0460, -0.0562,  1.3519,  1.0346,
         1.5426,  1.1489,  2.2104,  1.6418,  0.1545,  0.5532,  1.1719,  1.3012,
        -0.0290,  0.9524,  1.2137,  1.4619,  0.7260,  1.6012,  2.1349,  1.3922,
         1.2663,  1.5798,  0.8871,  0.6703,  0.9291,  0.5629,  2.0903,  1.0985,
         2.0959,  0.2971,  0.9599,  0.7638,  1.5711,  0.9797,  0.9526, -0.3802,
         0.4172,  1.3826,  0.7901,  0.6082,  1.8110,  0.8778,  0.5296,  0.6401,
         0.7720,  0.6608,  0.2524,  0.7712,  0.9594,  1.4889,  1.5396,  0.6831,
         1.3853,  0.9672,  1.2713,  2.2495,  1.1123, -0.3315,  1.0664,  1.0016,
         0.5897,  1.1516,  1.3643,  1.2073,  0.6662,  0.7288,  0.6452,  0.7312,
         0.8587,  1.2511,  1.3014,  1.0993,  0.9654,  1.6494,  1.3995,  0.5613,
         0.4905,  1.0943,  0.7161,  1.5755,  0.8665, -0.0087,  0.3587,  0.6147,
         1.5402,  1.6170,  0.3582,  1.1440,  1.3334,  1.2324,  1.1936,  0.3325])
```

Forward pass

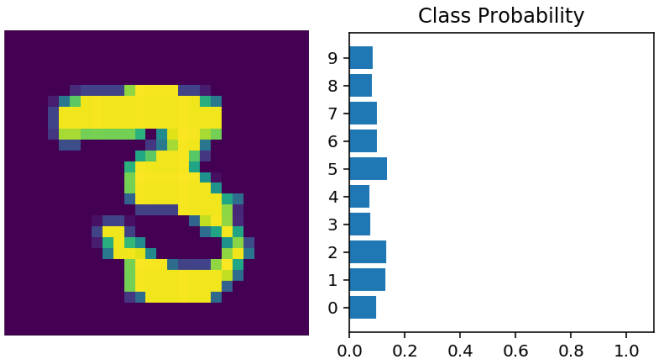
Now that we have a network, let's see what happens when we pass in an image.

In [49]:
```
# Grab some data
dataiter = iter(trainloader)
images, labels = dataiter.next()

# Resize images into a 1D vector, new shape is (batch size, color channels, image pixels)
#images.resize_(64, 1, 784)
images.resize_(images.shape[0], 1, 784)
#to 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 because all class probabilities are random. This is because we haven't trained it yet and all the weights are random!

Using `nn.Sequential`

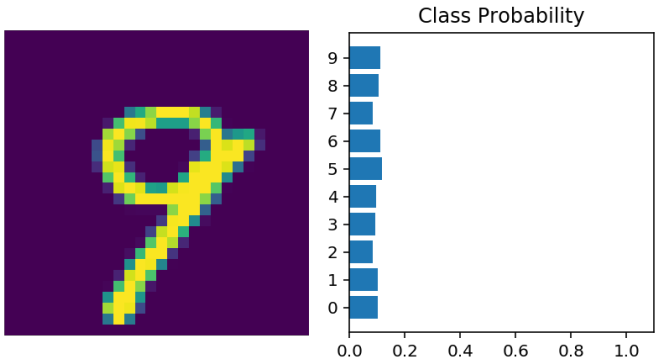
PyTorch provides a convenient way to build networks where a tensor is passed sequentially through operations, `nn.Sequential` (documentation (https://pytorch.org/docs/master/nn.html#torch.nn.Sequential)). Using `nn.Sequential`, we can build the equivalent network:

In [57]:
```
# Hyperparameters for our network
input_size = 784
hidden_sizes = [128, 64]
output_size = 10

# Build a feed-forward network:
# Fully-connected with 128 output units >> ReLU >> Fully-connected with 64 output
units >> ReLU >>
# Fully-connected with 10 output units >> Softmax
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()
)
```



In [58]:
```
ps
```

Out[58]:
```
tensor([[0.1017, 0.1009, 0.0834, 0.0917, 0.0967, 0.1173, 0.1115, 0.0833, 0.1038,
         0.1096]], grad_fn=<SoftmaxBackward>)
```

In [59]:
```
images, labels
```

Out[59]:
```
(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]]]),
 tensor([9, 4, 5, 9, 0, 9, 0, 8, 6, 5, 6, 1, 6, 7, 9, 9, 7, 6, 3, 9, 5, 1, 0, 9,
         3, 6, 0, 4, 7, 8, 6, 6, 2, 8, 1, 8, 5, 8, 1, 2, 1, 4, 0, 3, 3, 8, 9, 4,
         2, 5, 0, 8, 7, 2, 7, 6, 5, 1, 9, 5, 1, 5, 7, 1]))
```

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 [61]:
```
print(model[0])
model[0].weight
```

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

Out[61]:
```
Parameter containing:
tensor([[ 0.0183,  0.0113, -0.0223,  ...,  0.0033, -0.0173, -0.0015],
        [ 0.0177,  0.0321, -0.0194,  ...,  0.0222,  0.0202,  0.0114],
        [-0.0147,  0.0209, -0.0343,  ..., -0.0254,  0.0171, -0.0009],
        ...,
        [-0.0051, -0.0035, -0.0200,  ...,  0.0032,  0.0222, -0.0185],
        [ 0.0131, -0.0006, -0.0135,  ...,  0.0045,  0.0224, -0.0254],
        [ 0.0275,  0.0024,  0.0264,  ...,  0.0202, -0.0320, -0.0351]],
       requires_grad=True)
```

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 [62]:
```
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[62]:
```
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 [63]:
```
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 [66]:
```
print(model[2])
print(model.fc2)
```

```
Linear(in_features=128, out_features=64, bias=True)
Linear(in_features=128, out_features=64, bias=True)
```

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

In [ ]:
```
```