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
```

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=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
```

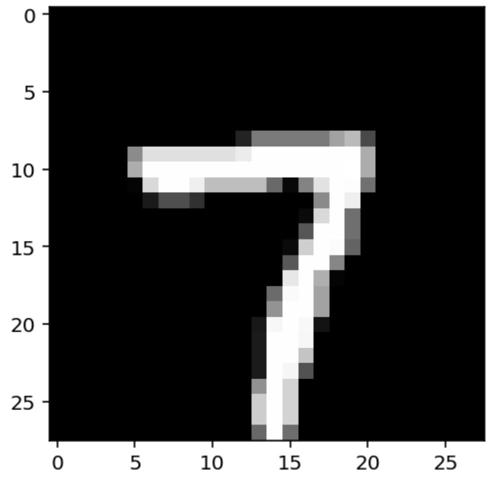
In [3]:

```
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])
```

In [4]:

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



In [5]:

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

# Flatten the input images
inputs = images.view(images.shape[0], -1)

# Create parameters
w1 = torch.randn(784, 256)
b1 = torch.randn(256)

w2 = torch.randn(256, 10)
b2 = torch.randn(10)

h = activation(torch.mm(inputs, w1) + b1)

out = torch.mm(h, w2) + b2
```

In [6]:

```
def softmax(x):
    return torch.exp(x)/torch.sum(torch.exp(x), dim=1).view(-1, 1)

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])
```

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
```

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(dim=1)
)
```

In [10]:

```python
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
```

In [11]:

```
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[11]:

```
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)
)
```

In [12]:

```
print(model.fc1.weight)
print(model.fc1.bias)
```

```
Parameter containing:
tensor([[ 0.0001,  0.0019, -0.0189,  ...,  0.0085, -0.0298, -0.0005],
        [ 0.0066, -0.0239,  0.0048,  ..., -0.0294, -0.0094,  0.0009],
        [ 0.0078,  0.0339,  0.0252,  ..., -0.0136, -0.0156,  0.0115],
        ...,
        [-0.0229,  0.0339,  0.0183,  ..., -0.0052,  0.0130, -0.0313],
        [-0.0341, -0.0178,  0.0019,  ...,  0.0183, -0.0258,  0.0191],
        [ 0.0209,  0.0071,  0.0042,  ..., -0.0191,  0.0245, -0.0042]],
       requires_grad=True)
Parameter containing:
tensor([-0.0095,  0.0350,  0.0226,  0.0004, -0.0094, -0.0204, -0.0149,  0.0148,
        -0.0230,  0.0156, -0.0030, -0.0322, -0.0278,  0.0142,  0.0009, -0.0097,
        -0.0085,  0.0237,  0.0286,  0.0143,  0.0199,  0.0194, -0.0268,  0.0080,
         0.0129, -0.0341,  0.0031, -0.0107, -0.0349, -0.0220,  0.0265, -0.0189,
         0.0271,  0.0233,  0.0094,  0.0306,  0.0076, -0.0029,  0.0072, -0.0194,
        -0.0272,  0.0089,  0.0171, -0.0018,  0.0192, -0.0273,  0.0054, -0.0076,
         0.0341,  0.0237,  0.0177, -0.0021,  0.0339,  0.0161,  0.0082, -0.0009,
         0.0308, -0.0354,  0.0182, -0.0304,  0.0228,  0.0141,  0.0356, -0.0056,
         0.0070,  0.0012, -0.0060, -0.0212, -0.0116, -0.0154,  0.0229, -0.0157,
         0.0317, -0.0345,  0.0098,  0.0310,  0.0340, -0.0171, -0.0034, -0.0056,
        -0.0268,  0.0285,  0.0141,  0.0069, -0.0051,  0.0102,  0.0122, -0.0268,
        -0.0245, -0.0209,  0.0265, -0.0175, -0.0085,  0.0075, -0.0059, -0.0109,
        -0.0105, -0.0194,  0.0174, -0.0266, -0.0054,  0.0325,  0.0035, -0.0265,
         0.0136,  0.0108,  0.0269, -0.0213, -0.0288, -0.0276, -0.0319, -0.0032,
        -0.0033, -0.0110,  0.0195,  0.0268, -0.0227,  0.0135,  0.0187,  0.0216,
         0.0183, -0.0050, -0.0107,  0.0087,  0.0071,  0.0083, -0.0259, -0.0027],
       requires_grad=True)
```

In [13]:

```
def view_classify(img, ps, version="MNIST"):
    ''' 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))
    if version == "MNIST":
        ax2.set_yticklabels(np.arange(10))
    elif version == "Fashion":
        ax2.set_yticklabels(['T-shirt/top',
                            'Trouser',
                            'Pullover',
                            'Dress',
                            'Coat',
                            'Sandal',
                            'Shirt',
                            'Sneaker',
                            'Bag',
                            'Ankle Boot'], size='small');
    ax2.set_title('Class Probability')
    ax2.set_xlim(0, 1.1)

plt.tight_layout()
```

```
<Figure size 432x288 with 0 Axes>
```

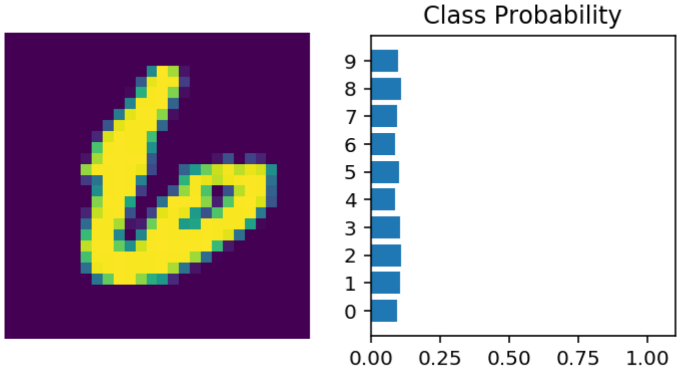
In [14]:

```
# 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)
# or 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]
view_classify(img.view(1, 28, 28), ps)
```



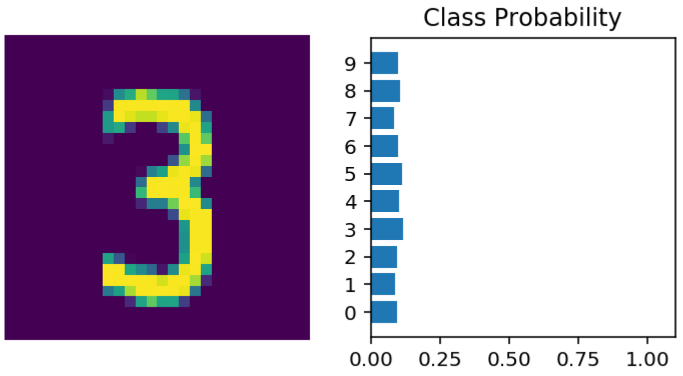
In [15]:

```
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,:])
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(dim=1)
)
```



In [16]:

```python
print(model[0])
model[0].weight
```

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

Out[16]:

```
Parameter containing:
tensor([[ 0.0288,  0.0328, -0.0029,  ...,  0.0293,  0.0024, -0.0138],
        [ 0.0135,  0.0257,  0.0319,  ...,  0.0098, -0.0299,  0.0301],
        [ 0.0052,  0.0303, -0.0215,  ..., -0.0111,  0.0210,  0.0073],
        ...,
        [-0.0180,  0.0251,  0.0069,  ...,  0.0282, -0.0064,  0.0177],
        [-0.0292,  0.0206,  0.0183,  ..., -0.0126, -0.0308, -0.0286],
        [ 0.0240,  0.0157, -0.0199,  ..., -0.0299,  0.0173, -0.0185]],
       requires_grad=True)
```

In [17]:

```
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[17]:

```
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(dim=1)
)
```

In [18]:

```
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 [21]:

```
model = nn.Sequential(nn.Linear(784, 128),
                      nn.ReLU(),
                      nn.Linear(128, 64),
                      nn.ReLU(),
                      nn.Linear(64, 10))

# Define the loss
criterion = nn.CrossEntropyLoss()

# Get our data
images, labels = next(iter(trainloader))
# Flatten images
images = images.view(images.shape[0], -1)

# Forward pass, get our logits
logits = model(images)
# Calculate the loss with the logits and the labels
loss = criterion(logits, labels)

print(loss)
```

```
tensor(2.3097, grad_fn=<NllLossBackward>)
```

In [22]:

```
model = nn.Sequential(nn.Linear(784, 128),
                      nn.ReLU(),
                      nn.Linear(128, 64),
                      nn.ReLU(),
                      nn.Linear(64, 10),
                      nn.LogSoftmax(dim=1))

# Define the loss
criterion = nn.NLLLoss()

# Get our data
images, labels = next(iter(trainloader))
# Flatten images
images = images.view(images.shape[0], -1)

# Forward pass, get our log-probabilities
logps = model(images)
# Calculate the loss with the logps and the labels
loss = criterion(logps, labels)

print(loss)
```

```
tensor(2.3137, grad_fn=<NllLossBackward>)
```

In [23]:

```
model = nn.Sequential(nn.Linear(784, 128),
                      nn.ReLU(),
                      nn.Linear(128, 64),
                      nn.ReLU(),
                      nn.Linear(64, 10),
                      nn.LogSoftmax(dim=1))

criterion = nn.NLLLoss()
images, labels = next(iter(trainloader))
images = images.view(images.shape[0], -1)

logps = model(images)
loss = criterion(logps, labels)
```

In [24]:

```
print('Before backward pass: \n', model[0].weight.grad)

loss.backward()

print('After backward pass: \n', model[0].weight.grad)
```

```
Before backward pass: 
 None
After backward pass: 
 tensor([[ 9.2931e-04,  9.2931e-04,  9.2931e-04,  ...,  9.2931e-04,
          9.2931e-04,  9.2931e-04],
        [-1.5003e-05, -1.5003e-05, -1.5003e-05,  ..., -1.5003e-05,
         -1.5003e-05, -1.5003e-05],
        [ 2.1729e-03,  2.1729e-03,  2.1729e-03,  ...,  2.1729e-03,
          2.1729e-03,  2.1729e-03],
        ...,
        [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  ...,  0.0000e+00,
          0.0000e+00,  0.0000e+00],
        [ 1.6365e-04,  1.6365e-04,  1.6365e-04,  ...,  1.6365e-04,
          1.6365e-04,  1.6365e-04],
        [ 8.6821e-04,  8.6821e-04,  8.6821e-04,  ...,  8.6821e-04,
          8.6821e-04,  8.6821e-04]])
```

In [25]:

```
from torch import optim

# Optimizers require the parameters to optimize and a learning rate
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

In [26]:

```
print('Initial weights - ', model[0].weight)

images, labels = next(iter(trainloader))
images.resize_(64, 784)

# Clear the gradients, do this because gradients are accumulated
optimizer.zero_grad()

# Forward pass, then backward pass, then update weights
output = model(images)
loss = criterion(output, labels)
loss.backward()
print('Gradient -', model[0].weight.grad)
```

```
Initial weights -  Parameter containing:
tensor([[-0.0105,  0.0027, -0.0096,  ..., -0.0188, -0.0175, -0.0176],
        [-0.0041, -0.0008, -0.0152,  ...,  0.0237,  0.0351,  0.0072],
        [ 0.0353,  0.0006,  0.0207,  ..., -0.0072,  0.0313, -0.0135],
        ...,
        [ 0.0123,  0.0277,  0.0325,  ...,  0.0087,  0.0282,  0.0218],
        [-0.0047, -0.0051,  0.0099,  ...,  0.0155,  0.0099, -0.0183],
        [-0.0129, -0.0269,  0.0245,  ..., -0.0034, -0.0055, -0.0108]],
       requires_grad=True)
Gradient - tensor([[-0.0006, -0.0006, -0.0006,  ..., -0.0006, -0.0006, -0.0006],
        [-0.0022, -0.0022, -0.0022,  ..., -0.0022, -0.0022, -0.0022],
        [-0.0015, -0.0015, -0.0015,  ..., -0.0015, -0.0015, -0.0015],
        ...,
        [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
        [-0.0001, -0.0001, -0.0001,  ..., -0.0001, -0.0001, -0.0001],
        [-0.0055, -0.0055, -0.0055,  ..., -0.0055, -0.0055, -0.0055]])
```

In [27]:

```
# Take an update step and few the new weights
optimizer.step()
print('Updated weights - ', model[0].weight)
```

```
Updated weights -  Parameter containing:
tensor([[-0.0105,  0.0027, -0.0096,  ..., -0.0188, -0.0175, -0.0176],
        [-0.0041, -0.0008, -0.0151,  ...,  0.0237,  0.0351,  0.0072],
        [ 0.0354,  0.0006,  0.0207,  ..., -0.0072,  0.0313, -0.0135],
        ...,
        [ 0.0123,  0.0277,  0.0325,  ...,  0.0087,  0.0282,  0.0218],
        [-0.0047, -0.0051,  0.0099,  ...,  0.0155,  0.0099, -0.0183],
        [-0.0128, -0.0269,  0.0245,  ..., -0.0034, -0.0054, -0.0108]],
       requires_grad=True)
```

In [28]:

```
model = nn.Sequential(nn.Linear(784, 128),
                      nn.ReLU(),
                      nn.Linear(128, 64),
                      nn.ReLU(),
                      nn.Linear(64, 10),
                      nn.LogSoftmax(dim=1))

criterion = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.003)

epochs = 5
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)

        # TODO: Training pass
        optimizer.zero_grad()

        output = model(images)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
    else:
        print(f"Training loss: {running_loss/len(trainloader)}")
```

```
Training loss: 1.9680475859499689
Training loss: 0.9329902477610086
Training loss: 0.5457367622839616
Training loss: 0.4400247997089998
Training loss: 0.39179654448017126
```

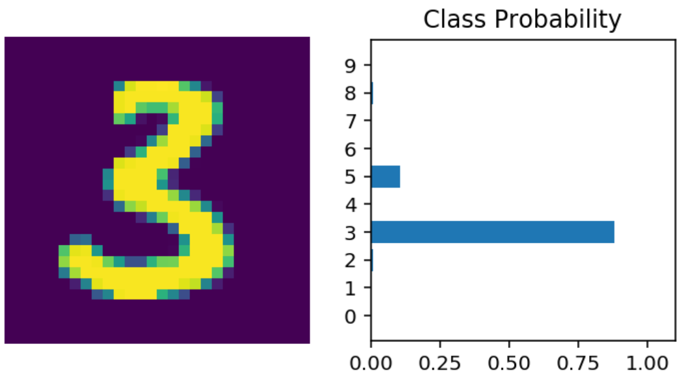
In [39]:

```
import helper

images, labels = next(iter(trainloader))

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

# Output of the network are log-probabilities, need to take exponential for probabilities
ps = torch.exp(logps)
view_classify(img.view(1, 28, 28), ps)
```



In [ ]:

In [ ]: