Recap: What we've been doing

`1-Tensors-in-PyTorch.ipynb:` introduces a tensor representation in PyTorch

`2-Neural-networks-in-PyTorch.ipynb:` introduces a basic framework for defining neural networks via the `nn` module

Training Neural Networks

In this notebook we will explore how to train neural networks.

IDEA: We can think of neural networks as universal function approximators.

Let's consider an example below. In the middle there is some function, *F(x)*, that maps the input (images of hand-written digits) to the output (probabilities for different class labels). For instance, if we pass an image with a digit 4 to the network, we would expect to obtain probability distribution with a high likelihood corresponding to the label 4. The magic of neural networks is that we can train them with non-linear activations to approximate this function *F(x)* successfully.

<img src="assets/function_approx.png" width=600px>

GOAL: We want to train the network by showing it lots of examples of digits and then adjust the weight parameters such that our network can approximate this function successfully.

Now how do we do that? To find the optimal weight parameters, we need to know how well our network is prediciting real outputs. Here we can calculate **loss function** (also called cost or optimizing function), which serves as a measure of our prediction error.

There are several different types of loss functions but one of the most widely used one is the **mean squared error** (MSE). MSE is often used in regression and binary classification problems. The formula for MSE is:

$$\ell = \frac{1}{2n}\sum_{i}^{n}\left(y_i - \hat{y}_i\right)^2$$

where n is the number of training examples, y_i are the true labels, and $\hat{y}_i$ are the predicted labels.

We can adjust the weight parameters such that this loss is minimized. Once the loss is minimized we know that out network is making as good predictions as it can.

IDEA: We find the minimum loss using a process called **gradient descent**. The gradient is the slope of the loss function with respect to the weight parameters. The gradient always points to the direction of the fastest change. For instance, consider the picture of the mountain below:

<img src='assets/gradient_descent.png' width=350px>

In this picture, the gradient is always going to point up the mountain. Imagine that our loss function is approximated by this mountain where we have the highest loss at the peak of the mountain and the lowest loss down in the valley. Therefore, if we want to minimize the loss, we have to go downwards and follow the direction of the negative gradient. You can think of this like descending a mountain by following the steepest slope to the base.

Backpropagation

For single layer networks, gradient descent is straightforward to implement. However, it's more complicated for deeper, multilayer neural networks like the one we've built. Complicated enough that it took about 30 years before researchers figured out how to train multilayer networks.

Training multilayer networks is done through **backpropagation** which is really just an application of the chain rule from calculus. It's easiest to understand if we convert a two layer network into a graph representation.

<img src='assets/backprop_diagram.png' width=550px>

In the forward pass through the network, our data and operations go from bottom to top here. We pass the input x through a linear transformation L_1 with weights W_1 and biases b_1. The output then goes through the sigmoid operation S and another linear transformation L_2. Finally we calculate the loss ℓ. We use the loss as a measure of how bad the network's predictions are. The goal then is to adjust the weights and biases to minimize the loss.

To train the weights with gradient descent, we propagate the gradient of the loss backwards through the network. Each operation has some gradient between the inputs and outputs. As we send the gradients backwards, we multiply the incoming gradient with the gradient for the operation. Mathematically, this is really just calculating the gradient of the loss with respect to the weights using the chain rule.

$$\frac{\partial \ell}{\partial W_1} = \frac{\partial L_1}{\partial W_1} \frac{\partial S}{\partial L_1} \frac{\partial L_2}{\partial S} \frac{\partial \ell}{\partial L_2}$$

Note: I'm glossing over a few details here that require some knowledge of vector calculus, but they aren't necessary to understand what's going on.

We update our weights using this gradient with some learning rate α.

$$W_1' = W_1 - \alpha \frac{\partial \ell}{\partial W_1}$$

The learning rate α is set such that the weight update steps are small enough that the iterative method settles in a minimum.

Losses in PyTorch

Let's start by seeing how we calculate the loss with PyTorch. Through the `nn` module, PyTorch provides losses such as the cross-entropy loss (`nn.CrossEntropyLoss`). You'll usually see the loss assigned to `criterion`. As noted in the last part, with a classification problem such as MNIST, we're using the softmax function to predict class probabilities. With a softmax output, you want to use cross-entropy as the loss. To actually calculate the loss, you first define the criterion then pass in the output of your network and the correct labels.

Something really important to note here. Looking at the documentation for `nn.CrossEntropyLoss` (https://pytorch.org/docs/stable/nn.html#torch.nn.CrossEntropyLoss),

> This criterion combines `nn.LogSoftmax()` and `nn.NLLLoss()` in one single class.
>
> The input is expected to contain scores for each class.

This means we need to pass in the raw output of our network into the loss, not the output of the softmax function. This raw output is usually called the *logits* or *scores*. We use the logits because softmax gives you probabilities which will often be very close to zero or one but floating-point numbers can't accurately represent values near zero or one (read more here (https://docs.python.org/3/tutorial/floatingpoint.html)). It's usually best to avoid doing calculations with probabilities, typically we use log-probabilities.

In [17]:
```
import torch
from torch import nn
import torch.nn.functional as F
from torchvision import datasets, transforms

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

In [18]:
```
# Build a feed-forward network
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(2.3413, grad_fn=<NllLossBackward>)
```

In my experience it's more convenient to build the model with a log-softmax output using `nn.LogSoftmax` or `F.log_softmax` (documentation (https://pytorch.org/docs/stable/nn.html#torch.nn.LogSoftmax)). Then you can get the actual probabilites by taking the exponential `torch.exp(output)`. With a log-softmax output, you want to use the negative log likelihood loss, `nn.NLLLoss` (documentation (https://pytorch.org/docs/stable/nn.html#torch.nn.NLLLoss)).

> **Exercise:** Build a model that returns the log-softmax as the output and calculate the loss using the negative log likelihood loss.

In [19]:
```
# Build a feed-forward network
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(2.3011, grad_fn=<NllLossBackward>)
```

Autograd

Now that we know how to calculate a loss, how do we use it to perform backpropagation? Torch provides a module, `autograd`, for automatically calculating the gradients of tensors. We can use it to calculate the gradients of all our parameters with respect to the loss. Autograd works by keeping track of operations performed on tensors, then going backwards through those operations, calculating gradients along the way. To make sure PyTorch keeps track of operations on a tensor and calculates the gradients, you need to set `requires_grad = True` on a tensor. You can do this at creation with the `requires_grad` keyword, or at any time with `x.requires_grad_(True)`.

You can turn off gradients for a block of code with the `torch.no_grad()` content:

```
x = torch.zeros(1, requires_grad=True)
>>> with torch.no_grad():
...     y = x * 2
>>> y.requires_grad
False
```

Also, you can turn on or off gradients altogether with `torch.set_grad_enabled(True|False)`.

The gradients are computed with respect to some variable `z` with `z.backward()`. This does a backward pass through the operations that created `z`.

In [20]:
```
x = torch.randn(2,2, requires_grad=True)
print(x)
```

```
tensor([[-1.5114, -1.2123],
        [ 1.6235,  0.7491]], requires_grad=True)
```

In [21]:
```
y = x**2
print(y)
```

```
tensor([[2.2844, 1.4696],
        [2.6359, 0.5611]], grad_fn=<PowBackward0>)
```

Below we can see the operation that created `y`, a power operation `PowBackward0`.

In [22]:
```
## grad_fn shows the function that generated this variable
print(y.grad_fn)
```

```
<PowBackward0 object at 0x124627710>
```

The autgrad module keeps track of these operations and knows how to calculate the gradient for each one. In this way, it's able to calculate the gradients for a chain of operations, with respect to any one tensor. Let's reduce the tensor `y` to a scalar value, the mean.

In [23]:
```
z = y.mean()
print(z)
```

```
tensor(1.7378, grad_fn=<MeanBackward1>)
```

You can check the gradients for `x` and `y` but they are empty currently.

In [24]:
```
print(x.grad)
```

```
None
```

To calculate the gradients, you need to run the `.backward` method on a Variable, `z` for example. This will calculate the gradient for `z` with respect to `x`

$$\frac{\partial z}{\partial x} = \frac{\partial}{\partial x}\left[\frac{1}{n}\sum_i^n x_i^2\right] = \frac{x}{2}$$

In [25]:
```
z.backward()
print(x.grad)
print(x/2)
```

```
tensor([[-0.7557, -0.6061],
        [ 0.8118,  0.3745]])
tensor([[-0.7557, -0.6061],
        [ 0.8118,  0.3745]], grad_fn=<DivBackward0>)
```

These gradients calculations are particularly useful for neural networks. For training we need the gradients of the weights with respect to the cost. With PyTorch, we run data forward through the network to calculate the loss, then, go backwards to calculate the gradients with respect to the loss. Once we have the gradients we can make a gradient descent step.

Loss and Autograd together

When we create a network with PyTorch, all of the parameters are initialized with `requires_grad = True`. This means that when we calculate the loss and call `loss.backward()`, the gradients for the parameters are calculated. These gradients are used to update the weights with gradient descent. Below you can see an example of calculating the gradients using a backwards pass.

In [28]:
```
# Build a feed-forward network
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 [29]:
```
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([[ 0.0037,  0.0037,  0.0037,  ...,  0.0037,  0.0037,  0.0037],
        [ 0.0018,  0.0018,  0.0018,  ...,  0.0018,  0.0018,  0.0018],
        [ 0.0037,  0.0037,  0.0037,  ...,  0.0037,  0.0037,  0.0037],
        ...,
        [-0.0029, -0.0029, -0.0029,  ..., -0.0029, -0.0029, -0.0029],
        [-0.0000, -0.0000, -0.0000,  ..., -0.0000, -0.0000, -0.0000],
        [-0.0030, -0.0030, -0.0030,  ..., -0.0030, -0.0030, -0.0030]])
```

Training the network!

There's one last piece we need to start training, an optimizer that we'll use to update the weights with the gradients. We get these from PyTorch's optim package (https://pytorch.org/docs/stable/optim.html). For example we can use stochastic gradient descent with `optim.SGD`. You can see how to define an optimizer below.

In [30]:
```
from torch import optim

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

Now we know how to use all the individual parts so it's time to see how they work together. Let's consider just one learning step before looping through all the data. The general process with PyTorch:

- Make a forward pass through the network
- Use the network output to calculate the loss
- Perform a backward pass through the network with `loss.backward()` to calculate the gradients
- Take a step with the optimizer to update the weights

Below I'll go through one training step and print out the weights and gradients so you can see how it changes. Note that I have a line of code `optimizer.zero_grad()`. When you do multiple backwards passes with the same parameters, the gradients are accumulated. This means that you need to zero the gradients on each training pass or you'll retain gradients from previous training batches.

In [31]:
```
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.forward(images)
loss = criterion(output, labels)
loss.backward()
print('Gradient -', model[0].weight.grad)
```

```
Initial weights -  Parameter containing:
tensor([[-0.0201, -0.0272,  0.0240,  ...,  0.0108, -0.0207, -0.0054],
        [-0.0245,  0.0337, -0.0337,  ...,  0.0061,  0.0334,  0.0114],
        [ 0.0158,  0.0220, -0.0082,  ..., -0.0197,  0.0327,  0.0214],
        ...,
        [ 0.0307,  0.0267, -0.0147,  ...,  0.0035,  0.0290, -0.0318],
        [ 0.0035, -0.0180, -0.0071,  ...,  0.0174, -0.0158, -0.0293],
        [ 0.0051, -0.0215,  0.0075,  ...,  0.0233, -0.0296,  0.0217]],
       requires_grad=True)
Gradient - tensor([[ 0.0001,  0.0001,  0.0001,  ...,  0.0001,  0.0001,  0.0001],
        [-0.0023, -0.0023, -0.0023,  ..., -0.0023, -0.0023, -0.0023],
        [ 0.0025,  0.0025,  0.0025,  ...,  0.0025,  0.0025,  0.0025],
        ...,
        [ 0.0019,  0.0019,  0.0019,  ...,  0.0019,  0.0019,  0.0019],
        [ 0.0007,  0.0007,  0.0007,  ...,  0.0007,  0.0007,  0.0007],
        [-0.0020, -0.0020, -0.0020,  ..., -0.0020, -0.0020, -0.0020]])
```

In [32]:
```
# Take an update step and few the new weights
optimizer.step()
print('Updated weights - ', model[0].weight)
```

```
Updated weights -  Parameter containing:
tensor([[-0.0201, -0.0272,  0.0240,  ...,  0.0108, -0.0207, -0.0054],
        [-0.0245,  0.0338, -0.0337,  ...,  0.0062,  0.0334,  0.0114],
        [ 0.0158,  0.0220, -0.0082,  ..., -0.0198,  0.0327,  0.0213],
        ...,
        [ 0.0307,  0.0267, -0.0147,  ...,  0.0035,  0.0290, -0.0319],
        [ 0.0035, -0.0180, -0.0071,  ...,  0.0174, -0.0158, -0.0293],
        [ 0.0052, -0.0214,  0.0076,  ...,  0.0233, -0.0296,  0.0218]],
       requires_grad=True)
```

Training for real

Now we'll put this algorithm into a loop so we can go through all the images. Some nomenclature, one pass through the entire dataset is called an *epoch*. So here we're going to loop through `trainloader` to get our training batches. For each batch, we'll doing a training pass where we calculate the loss, do a backwards pass, and update the weights.

> **Exercise:** Implement the training pass for our network. If you implemented it correctly, you should see the training loss drop with each epoch.

In [33]:
```
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.forward(images)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
    else:
        print(running_loss/len(trainloader))
```

```
1.9611555509475758
0.9276951081526559
0.5417190034156923
0.4331077055286751
0.38472682178846557
```

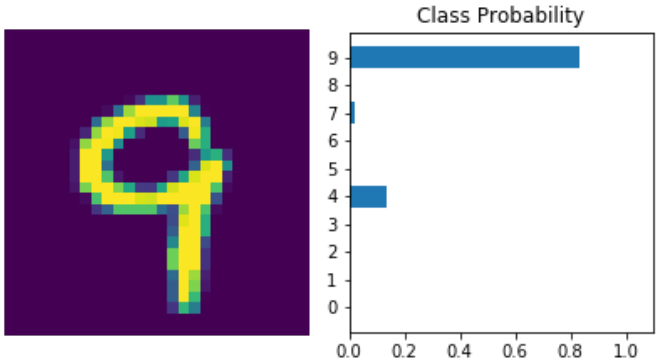
After the training has finished we can check the network predictions.

In [36]:
```
%matplotlib inline
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.forward(img)

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



In [ ]: