Part 3 - Training Neural Networks

July 10, 2018

1 Training Neural Networks

The network we built in the previous part isn't so smart, it doesn't know anything about our handwritten digits. Neural networks with non-linear activations work like universal function approximators. There is some function that maps your input to the output. For example, images of handwritten digits to class probabilities. The power of neural networks is that we can train them to approximate this function, and basically any function given enough data and compute time.

At first the network is naive, it doesn't know the function mapping the inputs to the outputs. We train the network by showing it examples of real data, then adjusting the network parameters such that it approximates this function.

To find these parameters, we need to know how poorly the network is predicting the real outputs. For this we calculate a **loss function** (also called the cost), a measure of our prediction error. For example, the mean squared loss is often used in regression and binary classification problems

$$\ell = \frac{1}{2n}\sum_i^n (y_i - \hat{y}_i)^2$$

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

By minimizing this loss with respect to the network parameters, we can find configurations where the loss is at a minimum and the network is able to predict the correct labels with high accuracy. We find this minimum using a process called **gradient descent**. The gradient is the slope of the loss function and points in the direction of fastest change. To get to the minimum in the least amount of time, we then want to follow the gradient (downwards). You can think of this like descending a mountain by following the steepest slope to the base.

1.1 Backpropagation

For single layer networks, gradient descent is simple 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, although it's straightforward once you learn about it.

This 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.

In the forward pass through the network, our data and operations go from right to left here. To train the weights with gradient descent, we propagate the gradient of the cost backwards through the network. Mathematically, this is really just calculating the gradient of the loss with respect to the weights using the chain rule.

$$\frac{\partial \ell}{\partial w_1} = \frac{\partial l_1}{\partial w_1} \frac{\partial s}{\partial l_1} \frac{\partial l_2}{\partial s} \frac{\partial \ell}{\partial l_2}$$

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

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

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

The first thing we need to do for training is define our loss function. In PyTorch, you'll usually see this as `criterion`. Here we're using softmax output, so we want to use `criterion = nn.CrossEntropyLoss()` as our loss. Later when training, you use `loss = criterion(output, targets)` to calculate the actual loss.

We also need to define the optimizer we're using, SGD or Adam, or something along those lines. Here I'll just use SGD with `torch.optim.SGD`, passing in the network parameters and the learning rate.

1.2 Autograd

Torch provides a module, `autograd`, for automatically calculating the gradient of tensors. It does this by keeping track of operations performed on tensors. To make sure PyTorch keeps track of operations on a tensor and calculates the gradients, you need to set `requires_grad` 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 [ ]: %matplotlib inline
        %config InlineBackend.figure_format = 'retina'

        from collections import OrderedDict

        import numpy as np
        import time

        import torch
        from torch import nn
        from torch import optim
        import torch.nn.functional as F

        import helper

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

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

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

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

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

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

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

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

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 cost, then, go backwards to calculate the gradients with respect to the cost. Once we have the gradients we can make a gradient descent step.

1.3 Get the data and define the network

The same as we saw in part 3, we'll load the MNIST dataset and define our network.

```
In [ ]: 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('MNIST_data/', download=True, train=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
```

I'll build a network with `nn.Sequential` here. Only difference from the last part is I'm not actually using softmax on the output, but instead just using the raw output from the last layer. This is because the output from softmax is a probability distribution. Often, the output will have values really close to zero or really close to one. Due to inaccuracies with representing numbers as floating points, computations with a softmax output can lose accuracy and become unstable. To get around this, we'll use the raw output, called the **logits**, to calculate the loss.

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

        # Build a feed-forward network
        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()),
                              ('logits', nn.Linear(hidden_sizes[1], output_size))]))
```

1.4 Training the network!

The first thing we need to do for training is define our loss function. In PyTorch, you'll usually see this as `criterion`. Here we're using softmax output, so we want to use `criterion = nn.CrossEntropyLoss()` as our loss. Later when training, you use `loss = criterion(output, targets)` to calculate the actual loss.

We also need to define the optimizer we're using, SGD or Adam, or something along those lines. Here I'll just use SGD with `torch.optim.SGD`, passing in the network parameters and the learning rate.

```
In [ ]: criterion = nn.CrossEntropyLoss()
        optimizer = optim.SGD(model.parameters(), lr=0.01)
```

First, 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 to get the logits
- Use the logits 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.

```
In [ ]: print('Initial weights - ', model.fc1.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.fc1.weight.grad)
        optimizer.step()
```

```
In [ ]: print('Updated weights - ', model.fc1.weight)
```

1.4.1 Training for real

Now we'll put this algorithm into a loop so we can go through all the images. This is fairly straightforward. We'll loop through the mini-batches in our dataset, pass the data through the network to calculate the losses, get the gradients, then run the optimizer.

```
In [ ]: optimizer = optim.SGD(model.parameters(), lr=0.003)
```

```
In [ ]: epochs = 3
        print_every = 40
        steps = 0
        for e in range(epochs):
            running_loss = 0
            for images, labels in iter(trainloader):
                steps += 1
                # Flatten MNIST images into a 784 long vector
                images.resize_(images.size()[0], 784)

                optimizer.zero_grad()

                # Forward and backward passes
                output = model.forward(images)
                loss = criterion(output, labels)
                loss.backward()
                optimizer.step()

                running_loss += loss.item()

                if steps % print_every == 0:
                    print("Epoch: {}/{}... ".format(e+1, epochs),
                          "Loss: {:.4f}".format(running_loss/print_every))
                running_loss = 0
```

With the network trained, we can check out it's predictions.

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

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

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

Now our network is brilliant. It can accurately predict the digits in our images. Next up you'll write the code for training a neural network on a more complex dataset.