Deep Learning with Python Ch. 2 Part 3

The engine of neural networks

- As we saw in the previous slides, each neural layer from MNIST transforms its input layer as follows:

  ```
  output = relu(dot(W, input) + b)
  ```

  In this expression *W* and *b* are tensors that are attributes of the layer. They are the *weights* or *trainable parameters* of the layer (the *kernel* and *bias* attributes respectively). This is what gets learned by the network from the training data.
- Initially these tensors are filled with small random values. So at first the network will produce gibberish. But this is a starting point. Then the weights are adjusted based on feedback. This gradual adjustment (called *training*) is what neural network learning is all about.

The training loop

The learning happens within a *training loop* that runs the following loop as long as necessary

1. Choose a batch of training examples x and corresponding targets y.
2. Run the network on x (called the *forward pass*) to obtain predictions y_pred.
3. Compute the loss of the network on the batch, a measure of the difference between y and y_pred.
4. Update all the weights of the network in a way that slightly reduces the loss of the batch.

Eventually we end up with a network with very low loss on the training examples. From afar this seems like magic, but up close it just consists of elementary steps.

Details of steps

- Step 1 is just I/O code. Steps 2 and 3 involve the application of a small number of tensor operations of the sort that we saw in the previous slides. The difficult part is Step 4, updating the network weights.
- We could try systematically considering different values for different weights to see what reduces the loss the most, but that would be extremely inefficient.
- Instead we will take advantage of the fact that all operations used in the network are *differentiable* and compute the *gradient* of the loss with regard to the networks values. We can then move the coefficients in the opposite direction of the gradient, thus decreasing the loss.

Gradients

- A *gradient* is the derivative of a tensor operation. It is the generalization of the concept of derivative to functions of multidimensional inputs, that is, to functions that take tensors as inputs.
- It is theoretically possible to find the minimums of a function (the places where its derivative is 0), then find the one of these where the function has the lowest value. This would give us the best weights immediately. But this is computationally intractable for the number of weights that neural networks have. Instead we do what is called *stochastic gradient descent* to gradually move toward the minimum loss.
- To elaborate a bit, Step 4 of our loop becomes:
  - Compute the gradient of the loss with regard to the network's parameters (a *backward pass*)
  - Move the parameters a little bit in the opposite direction from the gradient, reducing the loss on the batch a bit.

Summary so far

- The elaborated algorithm is called *mini-batch stochastic gradient descent* (aka mini-batch SGD). Stochastic just means random and it is because the batches are chosen randomly.
- One issue is what is called the *step size* which is how much you adjust the weights in any one step.
- Another issue is how big your batch sizes are. If you use all the training data in each pass, then you get more accurate results, but it is very inefficient. If you have batch sizes of one, it takes a very long time to converge.

Backpropagation

- So far we have just assumed that combinations of our tensor functions are differentiable. In fact, since a neural network function consists of many tensor functions chained together, calculating the derivative requires the use of the *chain rule*. This is exactly what is required to update the weights. The resulting algorithm is called *backpropagation* (or reverse-mode differentiation) and works backward, using the chain rule, to compute the contribution of each parameter to the loss value.
- Tools such as *TensorFlow* are capable of symbolic differentiation and are able to compute the required gradient function from the chain of tensor functions automatically. So there is no reason to implement backpropagation by hand.