Chapter 3

Computation Graphs

For our example neural network from Section 2.5, we painstakingly worked out derivates for gradient computations needed by gradient descent training. After all this hard work, it may come as surprise that you will likely never have to do this again. It can be done automatically, even for arbitrarily complex neural network architectures. There are a number of toolkits that allow you to define the network and it will take care of the rest. In this section, we will take a close look at how this works.

3.1 Neural Networks as Computation Graphs

First, we will take a different look at the networks we are building. We previously represented neural networks as graphs consisting of nodes and their connections (recall Figure 2.4 on page 15), or by mathematical equations such as

$$\begin{aligned} h &= \text{sigmoid}(W_1 x + b_1) \\ y &= \text{sigmoid}(W_2 h + b_2) \end{aligned} \tag{3.1}$$

The equations above describe the feed-forward neural network that we use as our running example. We now represent this math in form of a **computation graph**. See Figure 3.1 for an illustration of the computation graph for our network. The graph contains as nodes the parameters of the models (the weight matrices W_1, W_2 and bias vectors b_1, b_2), the input x and the mathematical operations that are carried out between them (product, sum, and sigmoid). Next to each parameter, we show their values.

Neural networks, viewed as computation graphs, are any arbitrary connected operations between an input and any number of parameters. Some of these operations may have little to do with any inspiration from neurons in the brain, so we are stretching the term *neural networks* quite a bit here. The graph does not have to have a nice tree structure as in our example, but may be any **acyclical directed graph**, i.e., anything goes as long there is a straightforward processing direction and no cycles. Another way to view such a graph is as a fancy way to

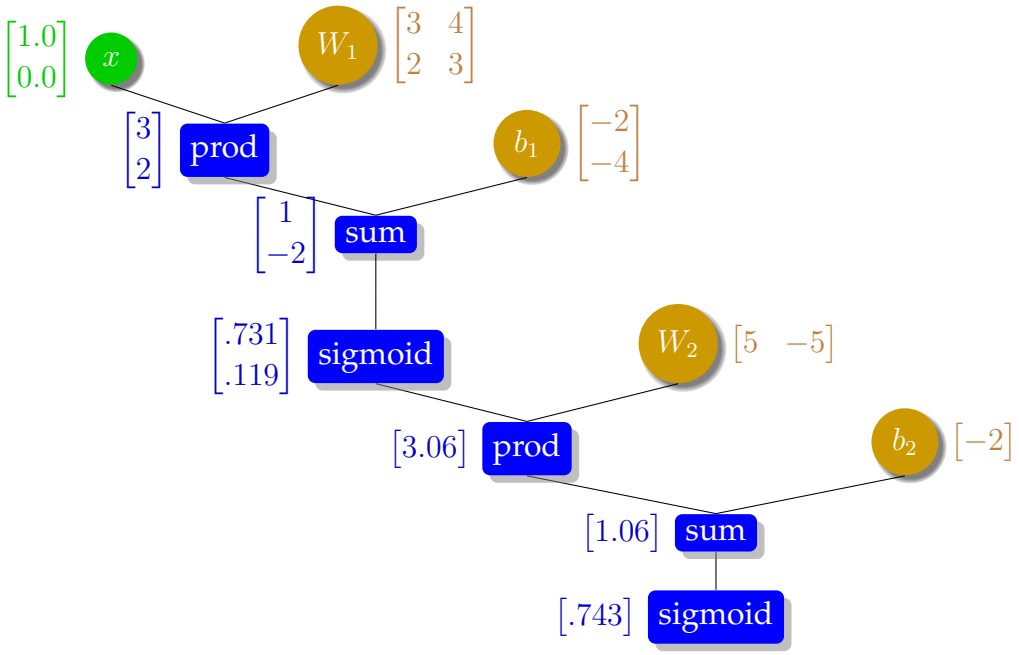

Figure 3.1: Two layer feed-forward neural network as a computation graph, consisting of the input value x, weight parameters W_1, W_2, b_1, b_2, and computation nodes (product, sum, sigmoid). To the right of each parameter node, its value is shown. To the left of input and computation nodes, we show how the input $(1, 0)^T$ is processed by the graph.

visualize a sequence of function calls that take as arguments the input, parameters, previously computed values, or any combination thereof, but have no recursion or loops.

Processing an input with the neural network requires placing the input value into the node x and carrying out the computations. In the figure, we show this with the input vector $(1, 0)^T$. The resulting numbers should look familiar since they are the same as when previously worked through this example in Section 2.4.

Before we move on, let us take stock of what each **computation node** in the graph has to accomplish. It consists of the following:

- a function that executes its computation operation
- links to input nodes
- when processing an example, the computed value

We will add two more items to each node in the following section.

3.2 Gradient Computations

So far, we showed how the computation graph can be used process an input value. Now we will examine how it can be used to vastly simply model training. Model training requires an error function and the computation of gradients to derive update rules for parameters.

The first of these is quite straightforward. To compute the error, we need to add another computation at the end of the computation graph. This computation takes the computed output value y and the given correct output value t from the training data and produces an error value. A typical error function is the L2 norm $\frac{1}{2}(t-y)^2$. From the view of training, the result of the execution of the computation graph is an error value.

Now, for the more difficult part — devising update rules for the parameters. Looking at the computation graph, model updates originate from the error values and propagate back to the model parameter. Hence, we call the computations needed to compute the update values also the **backward pass** through the graph, opposed to the **forward pass** that computed output and error.

Consider the chain of operations that connect the weight matrix W_2 to the error computation.

$$\begin{aligned} e &= \text{L2}(y,t) \\ y &= \text{sigmoid}(s) \\ s &= \text{sum}(p,b_2) \\ p &= \text{prod}(h,W_2) \end{aligned} \tag{3.2}$$

where h are the values of the hidden layer nodes, resulting from earlier computations.

To compute the update rule for the parameter matrix W_2, we view the error as a function of these parameters and take the derivative with respect to them, in our case $\frac{dL2(W_2)}{dW_2}$. Recall that when we computed this derivate we first broke it up into steps using the chain rule. We now do the same here.

Calculus Refresher
In calculus, the **chain rule** is a formula for computing the derivative of the composition of two or more functions. That is, if f and g are functions, then the chain rule expresses the derivative of their composition $f \cdot g$ (the function which maps x to $f(g(x))$) in terms of the derivatives of f and g and the product of functions as follows:
$(f \circ g)' = (f' \circ g) \cdot g'$.
This can be written more explicitly in terms of the variable. Let $F = f \cdot g$, or equivalently, $F(x) = f(g(x))$ for all x. Then one can also write $F'(x) = f'(g(x))g'(x)$
The chain rule may be written, in Leibniz's notation, in the following way. If a variable z depends on the variable y, which itself depends on the variable x, so that y and z are therefore dependent variables, then z, via the intermediate variable of y, depends on x as well. The chain rule then states,
$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$
The two versions of the chain rule are related, if $z = f(y)$ and $y = g(x)$, then
$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} = f'(y)g'(x) = f'(g(x))g'(x)$

(adapted from Wikipedia)

$$\frac{d\text{L2}(W_2)}{dW_2} = \frac{d\text{L2}(\text{sigmoid}(\text{sum}(\text{prod}(h,W_2),b_2)),t)}{dW_2} = \frac{d\text{L2}(y,t)}{dy}\frac{d\text{sigmoid}(s)}{ds}\frac{d\text{sum}(p,b_2)}{dp}\frac{d\text{prod}(h,W_2)}{dW_2} \tag{3.3}$$

Note that for the purpose for computing an update rule for W_2, we treat all the other variables in this computation (the target value t, the bias vector b_2, the hidden node values h) as constants. This breaks up the derivative of the error with respect to the parameters W_2 into a chain of derivatives along the line of the nodes of the computation graph.

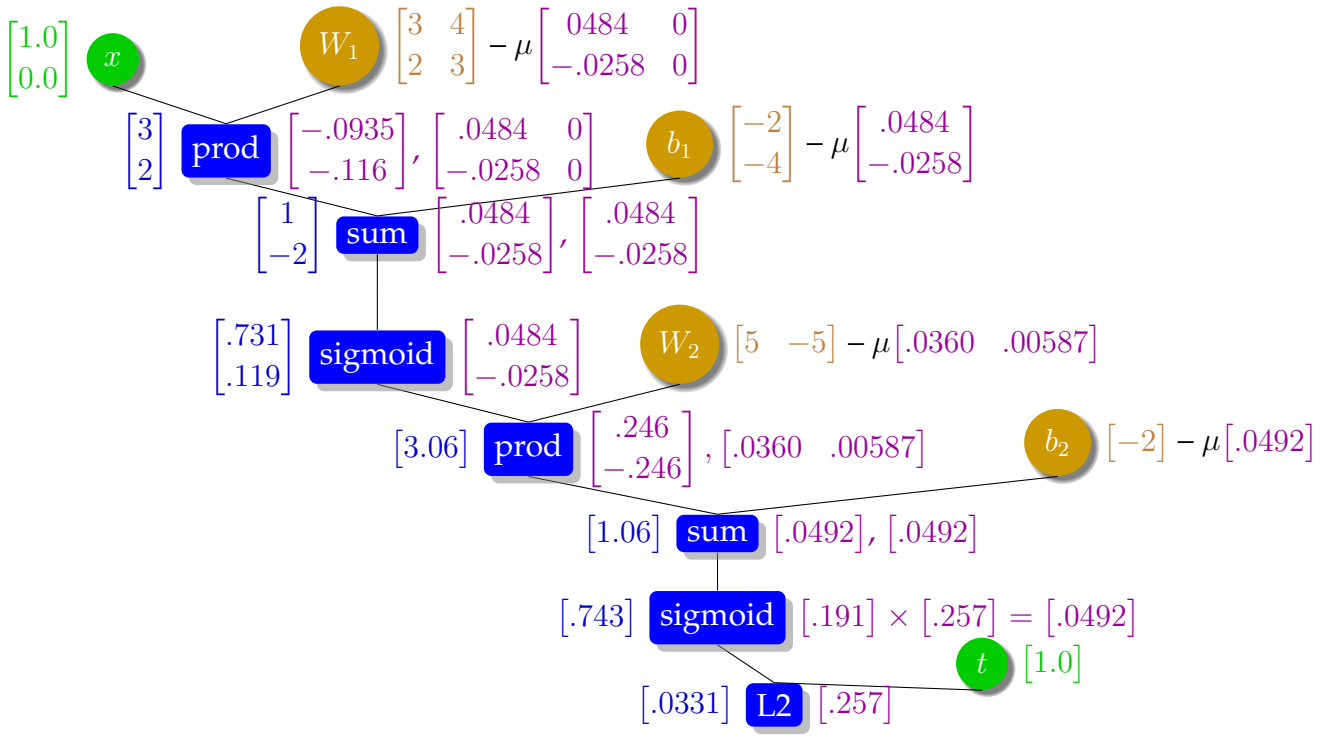

Figure 3.2: Computation graph with gradients computed in the backward pass for the training example $(0, 1)^T \rightarrow 1.0$. Gradients are computed with respect to the input of the nodes, so some nodes that have two inputs also have two gradients. See text for details on the computations of the values.

Hence, all we have to do for gradient computations is to come up with derivatives for each node in the computation graph. In our example these are

$$\begin{aligned}
\frac{d\text{L2}(y,t)}{dy} &= \frac{d\frac{1}{2}(t-y)^2}{dy} = t - y \\
\frac{d\text{sigmoid}(s)}{ds} &= \text{sigmoid}(s)(1 - \text{sigmoid}(s)) \\
\frac{d\text{sum}(p, b_2)}{dp} &= \frac{dp + b_2}{dp} = 1 \\
\frac{d\text{prod}(h, W_2)}{dW_2} &= \frac{dW_2 h}{dW_2} = h
\end{aligned} \tag{3.4}$$

If we want to compute the gradient update for a parameter such as W_2, we compute values in a backward pass, starting from the error term y. See Figure 3.2 for an illustration.

To give more detail on the computation of the gradients in the backward pass, starting at the bottom of the graph:

- For the **L2** node, we use the formula

$$\frac{d\text{L2}(y,t)}{dy} = \frac{d\frac{1}{2}(t-y)^2}{dy} = t - y \tag{3.5}$$

The given target output value given as training data is $t = 1$, while we computed $y = 0.743$ in the forward pass. Hence, the gradient for the L2 norm is $1 - 0.743 = 0.257$. Note that we are using values computed in the forward pass for these gradient computations.

- For the lower **sigmoid** node, we use the formula

$$\frac{d\text{sigmoid}(s)}{ds} = \text{sigmoid}(s)(1 - \text{sigmoid}(s)) \tag{3.6}$$

Recall that the formula for the sigmoid is $\text{sigmoid}(s) = \frac{1}{1+e^{-s}}$. Plugging in the value for $s = 1.06$ computed in the forward pass into this formula gives us 0.191. The chain rule requires us to multiply this with the value that we just computed for the **L2** node, i.e., 0.257, which gives us $0.191 \times 0.257 = 0.0492$.

- For the lower **sum** node, we simply copy the previous value, since the derivate is 1:

$$\frac{d\text{sum}(p, b_2)}{dp} = \frac{dp + b_2}{dp} = 1 \tag{3.7}$$

Note that there are two gradients associated with the sum node. One with respect to the output of the **prod** node, and one with the b_2 parameter. In both cases, the derivative is 1, so both values are the same. Hence, the gradient in both cases is 0.0492.

- For the lower **prod** node, we use the formula

$$\frac{d\text{prod}(h, W_2)}{dW_2} = \frac{dW_2 h}{dW_2} = h \tag{3.8}$$

So far, we dealt with scalar values. Here we encounter vectors for the first time: the value of the hidden nodes $h = (0.731, 0.119)^T$. The chain rule requires us to multiply this with the previously computed scalar 0.0492:

$$\left(\begin{bmatrix} 0.731 \\ 0.119 \end{bmatrix} \times 0.0492 \right)^T = \begin{bmatrix} 0.0360 & 0.00587 \end{bmatrix}$$

As for the sum node, there are two inputs and hence two gradients. The other gradient is with respect to the output of the upper **sigmoid** node.

$$\frac{d\text{prod}(h, W_2)}{dh} = \frac{dW_2 h}{dh} = W_2 \tag{3.9}$$

Similarly to above, we compute

$$(W_2 \times 0.0492)^T = \left(\begin{bmatrix} 5 & -5 \end{bmatrix} \times 0.0492 \right)^T = \begin{bmatrix} 0.246 \\ -0.246 \end{bmatrix}$$

Having all the gradients in place, we can now read of the relevant values for weight updates. These are the gradients associated with trainable parameters. For the W_2 weight matrix, this is the second gradient of the **prod** node. So the new value for W_2 at time step $t+1$ is

$$W_2^{t+1} = W_2^t - \mu \frac{d\text{prod}(x, W_2^t)}{dW_2^t} = \begin{bmatrix} 5 & 5 \end{bmatrix} - \mu \begin{bmatrix} 0.0360 & 0.00587 \end{bmatrix} \tag{3.10}$$

The remaining computations are carried out in very similar fashion, since they form simply another layer of the feed-forward neural network.

Our example did not include one special case: the output of a computation may be used multiple times in subsequent steps of a computation graphs. So, there are multiple output nodes that feed back gradients in the back-propagation pass. In this case, we add up the gradients from these descendent steps to factor in their added impact.

Let us take a second look at what a node in a computation graph comprises:

- a function that computes its value
- links to input nodes (to obtain argument values)
- when processing an example in the forward pass, the computed value
- a function that executes its gradient computation
- links to children nodes (to obtain downstream gradient values)
- when processing an example in the forward pass, the computed gradient

From an object oriented programming view, a node in a computation graph provides a forward and backward function for value and gradient computations. As instantiated in an computation graph, it is connected with specific inputs and outputs, and is also aware of the dimensions of its variables its value and gradient. During forward and backward pass, these variables are filled in.

3.3 Deep Learning Frameworks

In the next sections, we will encounter various network architectures. What all of these share, however, are the need for vector and matrix operations, as well as the computation of derivatives to obtain weight update formulas. It would be quite tedious to write almost identical code to deal with each these variants. Hence, a number of frameworks have emerged to support developing neural network methods for any chosen problem. At the time of writing, the most prominent ones are **Theano**[1] (a Python library that dymanically generates and compiles C++ code and is build on NumPy), **Torch**[2] (a machine learning library and a script language based on the Lua programming language), **pyTorch**[3] (the Python variant of Torch), **DyNet**[4] (a C++ implementation by natural language processing researchers that can be used as a library in C++ or Python), and **Tensorflow**[5] (a more recent entry to the genre from Google).

[1] http://deeplearning.net/software/theano/
[2] http://torch.ch/
[3] http://pytorch.org/
[4] http://dynet.readthedocs.io/
[5] http://www.tensorflow.org/

These frameworks are less geared towards ready-to-use neural network architectures, but provide efficient implementations of the vector space operations and computation of derivatives, with seamless support of GPUs. Our example from Section 2 can be implemented in a few lines of Python code, as we will show in this section, using the example of Theano (other frameworks are quite similar).

You can execute the following commands on the Python command line interface if you first installed Theano (`pip install Theano`).

```
> import numpy
> import theano
> import theano.tensor as T
```

The mapping of the input layer `x` to the hidden layer `h` uses a weight matrix `W`, a bias vector `b`, and a mapping function which consists of the linear combination `T.dot` and the sigmoid activation function.

```
> x = T.dmatrix()
> W = theano.shared(value=numpy.array([[3.0,2.0],[4.0,3.0]]))
> b = theano.shared(value=numpy.array([-2.0,-4.0]))
> h = T.nnet.sigmoid(T.dot(x,W)+b)
```

Note that we define `x` as a matrix. This allows us to process several training examples at once (a sequence of vectors). A good way to think about these definitions of `x` and `h` is in term of a functional programming language. They symbolically define operations. To actually define a function that can be called, the Theano method `function` is used.

```
> h_function = theano.function([x], h)
> h_function([[1,0]])
array([[ 0.73105858, 0.11920292]])
```

This example call to `h_function` computes the values for the hidden nodes (compare to the numbers in Table 2.1 on page 15).

The mapping from the hidden layer `h` to the output layer `y` is defined in the same fashion.

```
W2 = theano.shared(value=numpy.array([5.0,-5.0] ))
b2 = theano.shared(-2.0)
y_pred = T.nnet.sigmoid(T.dot(h,W2)+b2)
```

Again, we can define a callable function to test the full network.

```
> predict = theano.function([x], y_pred)
> predict([[1,0]])
array([[ 0.7425526]])
```

Model training requires the definition of a cost function (we use the L2 norm). To formulate it, we first need to define the variable for the correct output. The overall cost is computed as average over all training examples.

```
> y = T.dvector()
> l2 = (y-y_pred)**2
> cost = l2.mean()
```

Gradient descent training requires the computation of the derivative of the cost function with respect to the model parameters (i.e., the values in the weight matrices `W` and `W2` and the bias vectors `b` and `b2`. A great benefit of using Theano is that it computes the derivatives for you. The following is also an example of a function with multiple inputs and multiple outputs.

```
> gW, gb, gW2, gb2 = T.grad(cost, [W,b,W2,b2])
```

We have now all we need to define training. The function updates the model parameters and returns the current predictions and cost. It uses a learning rate of 0.1.

```
> train = theano.function(inputs=[x,y],outputs=[y_pred,cost],
     updates=((W, W-0.1*gW), (b, b-0.1*gb),
             (W2, W2-0.1*gW2), (b2, b2-0.1*gb2)))
```

Let us define the training data.

```
> DATA_X = numpy.array([[0,0],[0,1],[1,0],[1,1]])
> DATA_Y = numpy.array([0,1,1,0])
> predict(DATA_X)
array([ 0.18333462, 0.7425526 , 0.7425526 , 0.33430961])
> train(DATA_X,DATA_Y)
[array([ 0.18333462, 0.7425526 , 0.7425526 , 0.33430961]),
array(0.06948320612438118)]
```

The training function returns the prediction and cost before the updates. If we call the training function again, then the predictions and cost have changed for the better.

```
> train(DATA_X,DATA_Y)
[array([ 0.18353091, 0.74260499, 0.74321824, 0.33324929]),
array(0.06923193686092949)]
```

Typically, we would loop over the training function until convergence. As discussed above, we may also break up the training data into mini-batches and train on one mini-batch at a time.