Becoming a Backprop Ninja

Over time you will become much more efficient in writing the backward pass, even for complicated circuits and all at once. Lets practice backprop a bit with a few examples. In what follows, lets not worry about Unit, Circuit classes because they obfuscate things a bit, and lets just use variables such as a,b,c,x, and refer to their gradients as da,db,dc,dx respectively. Again, we think of the variables as the "forward flow" and their gradients as "backward flow" along every wire. Our first example was the * gate:

```
var x = a * b;
// and given gradient on x (dx), we saw that in backprop we would compute:
var da = b * dx;
var db = a * dx;
```

In the code above, I'm assuming that the variable dx is given, coming from somewhere above us in the circuit while we're doing backprop (or it is +1 by default otherwise). I'm writing it out because I want to explicitly show how the gradients get chained together. Note from the equations that the * gate acts as a *switcher* during backward pass, for lack of better word. It remembers what its inputs were, and the gradients on each one will be the value of the other during the forward pass. And then of course we have to multiply with the gradient from above, which is the chain rule. Here's the + gate in this condensed form:

```
var x = a + b;
// ->
var da = 1.0 * dx;
var db = 1.0 * dx;
```

Where 1.0 is the local gradient, and the multiplication is our chain rule. What about adding three numbers?:

```
// lets compute x = a + b + c in two steps:
var q = a + b; // gate 1
var x = q + c; // gate 2

// backward pass:
dc = 1.0 * dx; // backprop gate 2
dq = 1.0 * dx;
da = 1.0 * dq; // backprop gate 1
db = 1.0 * dq;
```

You can see what's happening, right? If you remember the backward flow diagram, the + gate simply takes the gradient on top and routes it equally to all of its inputs (because its local gradient is always simply 1.0 for all its inputs, regardless of their actual values). So we can do it much faster:

```
var x = a + b + c;
var da = 1.0 * dx; var db = 1.0 * dx; var dc = 1.0 * dx;
```

Okay, how about combining gates?:

```
var x = a * b + c;
// given dx, backprop in-one-sweep would be =>
da = b * dx;
db = a * dx;
dc = 1.0 * dx;
```

If you don't see how the above happened, introduce a temporary variable `q = a * b` and then compute `x = q + c` to convince yourself. And here is our neuron, lets do it in two steps:

```
// let's do our neuron in two steps:
var q = a*x + b*y + c;
var f = sig(q); // sig is the sigmoid function
// and now backward pass, we are given df, and:
var df = 1;
var dq = (f * (1 - f)) * df;
// and now we chain it to the inputs
var da = x * dq;
var dx = a * dq;
var dy = b * dq;
var db = y * dq;
var dc = 1.0 * dq;
```

I hope this is starting to make a little more sense. Now how about this:

```
var x = a * a;
var da = //???
```

You can think of this as value `a` flowing to the `*` gate, but the wire gets split and becomes both inputs. This is actually simple because the backward flow of gradients always adds up. In other words nothing changes:

```
var da = a * dx; // gradient into a from first branch
da += a * dx; // and add on the gradient from the second branch

// short form instead is:
var da = 2 * a * dx;
```

In fact, if you know your power rule from calculus you would also know that if you have f(a)=a^2 then ∂f(a)/∂a = 2a(∂f(a)/∂a)=2a, which is exactly what we get if we think of it as wire splitting up and being two inputs to a gate.

Lets do another one:

```
var x = a*a + b*b + c*c;
// we get:
var da = 2*a*dx;
var db = 2*b*dx;
var dc = 2*c*dx;
```

Okay now lets start to get more complex:

```
var x = Math.pow(((a * b + c) * d), 2); // pow(x,2) squares the input JS
```

When more complex cases like this come up in practice, I like to split the expression into manageable chunks which are almost always composed of simpler expressions and then I chain them together with chain rule:

```
var x1 = a * b + c;
var x2 = x1 * d;
var x = x2 * x2; // this is identical to the above expression for x
// and now in backprop we go backwards:
var dx2 = 2 * x2 * dx; // backprop into x2
var dd = x1 * dx2; // backprop into d
var dx1 = d * dx2; // backprop into x1
var da = b * dx1;
var db = a * dx1;
var dc = 1.0 * dx1; // done!
```

That wasn't too difficult! Those are the backprop equations for the entire expression, and we've done them piece by piece and backpropped to all the variables. Notice again how for every variable during forward pass we have an equivalent variable during backward pass that contains its gradient with respect to the circuit's final output. Here are a few more useful functions and their local gradients that are useful in practice:

```
var x = 1.0/a; // division
var da = -1.0/(a*a);
```

Here's what division might look like in practice then:

```
var x = (a + b)/(c + d);
// lets decompose it in steps:
var x1 = a + b;
```

```
var x2 = c + d;
var x3 = 1.0 / x2;
var x = x1 * x3; // equivalent to above
// and now backprop, again in reverse order:
var dx1 = x3 * dx;
var dx3 = x1 * dx;
var dx2 = (-1.0/(x2*x2)) * dx3; // local gradient as shown above, and chain rule
var da = 1.0 * dx1; // and finally into the original variables
var db = 1.0 * dx1;
var dc = 1.0 * dx2;
var dd = 1.0 * dx2;
```

Hopefully you see that we are breaking down expressions, doing the forward pass, and then for every variable (such as a) we derive its gradient da as we go backwards, one by one, applying the simple local gradients and chaining them with gradients from above. Here's another one:

```
var x = Math.max(a, b);
var da = a === x ? 1.0 * dx : 0.0;
var db = b === x ? 1.0 * dx : 0.0;
```

Okay this is making a very simple thing hard to read. The max function passes on the value of the input that was largest and ignores the other ones. In the backward pass then, the max gate will simply take the gradient on top and route it to the input that actually flowed through it during the forward pass. The gate acts as a simple switch based on which input had the highest value during forward pass. The other inputs will have zero gradient. That's what the === is about, since we are testing for which input was the actual max and only routing the gradient to it.

Finally, lets look at the Rectified Linear Unit non-linearity (or ReLU), which you may have heard of. It is used in Neural Networks in place of the sigmoid function. It is simply thresholding at zero:

```
var x = Math.max(a, 0)
// backprop through this gate will then be:
var da = a > 0 ? 1.0 * dx : 0.0;
```

In other words this gate simply passes the value through if it's larger than 0, or it stops the flow and sets it to zero. In the backward pass, the gate will pass on the gradient from the top if it was activated during the forawrd pass, or if the original input was below zero, it will stop the gradient flow.

I will stop at this point. I hope you got some intuition about how you can compute entire expressions (which are made up of many gates along the way) and how you can compute backprop for every one of them.

Everything we've done in this chapter comes down to this: We saw that we can feed some input through arbitrarily complex real-valued circuit, tug at the end of the circuit with some force, and backpropagation distributes that tug through the entire circuit all the

way back to the inputs. If the inputs respond slightly along the final direction of their tug, the circuit will "give" a bit along the original pull direction. Maybe this is not immediately obvious, but this machinery is a powerful *hammer* for Machine Learning.

"Maybe this is not immediately obvious, but this machinery is a powerful hammer for Machine Learning."

Lets now put this machinery to good use.