Hacker's guide to Neural Networks

Hi there, I'm a CS PhD student at Stanford. I've worked on Deep Learning for a few years as part of my research and among several of my related pet projects is ConvNetJS - a Javascript library for training Neural Networks. Javascript allows one to nicely visualize what's going on and to play around with the various hyperparameter settings, but I still regularly hear from people who ask for a more thorough treatment of the topic. This article (which I plan to slowly expand out to lengths of a few book chapters) is my humble attempt. It's on web instead of PDF because all books should be, and eventually it will hopefully include animations/demos etc.

My personal experience with Neural Networks is that everything became much clearer when I started ignoring full-page, dense derivations of backpropagation equations and just started writing code. Thus, this tutorial will contain **very little math** (I don't believe it is necessary and it can sometimes even obfuscate simple concepts). Since my background is in Computer Science and Physics, I will instead develop the topic from what I refer to as **hackers's perspective**. My exposition will center around code and physical intuitions instead of mathematical derivations. Basically, I will strive to present the algorithms in a way that I wish I had come across when I was starting out.

> *"…everything became much clearer when I started writing code."*

You might be eager to jump right in and learn about Neural Networks, backpropagation, how they can be applied to datasets in practice, etc. But before we get there, I'd like us to first forget about all that. Let's take a step back and understand what is really going on at the core. Lets first talk about real-valued circuits.

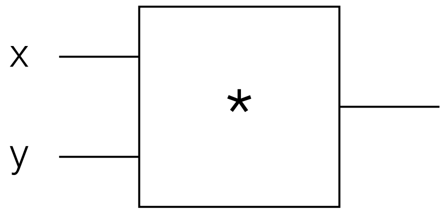
Update note: I suspended my work on this guide a while ago and redirected a lot of my energy to teaching CS231n (Convolutional Neural Networks) class at Stanford. The notes are on cs231.github.io and the course slides can be found here. These materials are highly related to material here, but more comprehensive and sometimes more polished.

Chapter 1: Real-valued Circuits

In my opinion, the best way to think of Neural Networks is as real-valued circuits, where real values (instead of boolean values `{0,1}`) “flow” along edges and interact in gates. However, instead of gates such as `AND`, `OR`, `NOT`, etc, we have binary gates such as `*` (multiply), `+` (add), `max` or unary gates such as `exp`, etc. Unlike ordinary boolean circuits, however, we will eventually also have **gradients** flowing on the same edges of the circuit, but in the opposite direction. But we’re getting ahead of ourselves. Let’s focus and start out simple.

Base Case: Single Gate in the Circuit

Lets first consider a single, simple circuit with one gate. Here’s an example:



The circuit takes two real-valued inputs `x` and `y` and computes `x * y` with the `*` gate. Javascript version of this would very simply look something like this:

```
var forwardMultiplyGate = function(x, y) {
  return x * y;
};
forwardMultiplyGate(-2, 3); // returns -6. Exciting.
```

And in math form we can think of this gate as implementing the real-valued function:

$$f(x, y) = xy$$

As with this example, all of our gates will take one or two inputs and produce a **single** output value.

The Goal

The problem we are interested in studying looks as follows:

1. We provide a given circuit some specific input values (e.g. `x = -2`, `y = 3`)
2. The circuit computes an output value (e.g. `-6`)

3. The core question then becomes: *How should one tweak the input slightly to increase the output?*

In this case, in what direction should we change `x,y` to get a number larger than `-6`? Note that, for example, `x = -1.99` and `y = 2.99` gives `x * y = -5.95`, which is higher than `-6.0`. Don't get confused by this: `-5.95` is better (higher) than `-6.0`. It's an improvement of `0.05`, even though the *magnitude* of `-5.95` (the distance from zero) happens to be lower.

Strategy #1: Random Local Search

Okay. So wait, we have a circuit, we have some inputs and we just want to tweak them slightly to increase the output value? Why is this hard? We can easily "forward" the circuit to compute the output for any given `x` and `y`. So isn't this trivial? Why don't we tweak `x` and `y` randomly and keep track of the tweak that works best:

```
// circuit with single gate for now
var forwardMultiplyGate = function(x, y) { return x * y; };
var x = -2, y = 3; // some input values

// try changing x,y randomly small amounts and keep track of what works be
var tweak_amount = 0.01;
var best_out = -Infinity;
var best_x = x, best_y = y;
for(var k = 0; k < 100; k++) {
  var x_try = x + tweak_amount * (Math.random() * 2 - 1); // tweak x a bit
  var y_try = y + tweak_amount * (Math.random() * 2 - 1); // tweak y a bit
  var out = forwardMultiplyGate(x_try, y_try);
  if(out > best_out) {
    // best improvement yet! Keep track of the x and y
    best_out = out;
    best_x = x_try, best_y = y_try;
  }
}
```

When I run this, I get `best_x = -1.9928`, `best_y = 2.9901`, and `best_out = -5.9588`. Again, `-5.9588` is higher than `-6.0`. So, we're done, right? Not quite: This is a perfectly fine strategy for tiny problems with a few gates if you can afford the compute time, but it won't do if we want to eventually consider huge circuits with millions of inputs. It turns out that we can do much better.

Stategy #2: Numerical Gradient

Here's a better way. Remember again that in our setup we are given a circuit (e.g. our circuit with a single `*` gate) and some particular input (e.g. `x = -2, y = 3`). The gate computes the output (`-6`) and now we'd like to tweak `x` and `y` to make the output higher.

A nice intuition for what we're about to do is as follows: Imagine taking the output value that comes out from the circuit and tugging on it in the positive direction. This positive tension will in turn translate through the gate and induce forces on the inputs `x` and `y`. Forces that tell us how `x` and `y` should change to increase the output value.

What might those forces look like in our specific example? Thinking through it, we can intuit that the force on `x` should also be positive, because making `x` slightly larger improves the circuit's output. For example, increasing `x` from `x = -2` to `x = -1` would give us output `-3` - much larger than `-6`. On the other hand, we'd expect a negative force induced on `y` that pushes it to become lower (since a lower `y`, such as `y = 2`, down from the original `y = 3` would make output higher: `2 x -2 = -4`, again, larger than `-6`). That's the intuition to keep in mind, anyway. As we go through this, it will turn out that forces I'm describing will in fact turn out to be the **derivative** of the output value with respect to its inputs (`x` and `y`). You may have heard this term before.

> *The derivative can be thought of as a force on each input as we pull on the output to become higher.*

So how do we exactly evaluate this force (derivative)? It turns out that there is a very simple procedure for this. We will work backwards: Instead of pulling on the circuit's output, we'll iterate over every input one by one, increase it very slightly and look at what happens to the output value. The amount the output changes in response is the derivative. Enough intuitions for now. Lets look at the mathematical definition. We can write down the derivative for our function with respect to the inputs. For example, the derivative with respect to `x` can be computed as:

$$\frac{\partial f(x,y)}{\partial x} = \frac{f(x+h,y) - f(x,y)}{h}$$

Where h is small - it's the tweak amount. Also, if you're not very familiar with calculus it is important to note that in the left-hand side of the equation above, the horizontal line does *not* indicate division. The entire symbol $\frac{\partial f(x,y)}{\partial x}$ is a single thing: the derivative of the function $f(x,y)$ with respect to x. The horizontal line on the right *is* division. I know it's confusing but it's standard notation. Anyway, I hope it doesn't look too scary because it isn't: The circuit was giving some initial output $f(x,y)$, and then we changed one of the inputs by a tiny amount h and read the new output $f(x+h,y)$. Subtracting those two quantities tells us the change, and the division by h just normalizes this change by the (arbitrary) tweak amount we used. In other words it's expressing exactly what I described above and translates directly to this code:

```
var x = -2, y = 3;
var out = forwardMultiplyGate(x, y); // -6
var h = 0.0001;

// compute derivative with respect to x
var xph = x + h; // -1.9999
var out2 = forwardMultiplyGate(xph, y); // -5.9997
var x_derivative = (out2 - out) / h; // 3.0

// compute derivative with respect to y
var yph = y + h; // 3.0001
var out3 = forwardMultiplyGate(x, yph); // -6.0002
var y_derivative = (out3 - out) / h; // -2.0
```

Lets walk through `x` for example. We turned the knob from `x` to `x + h` and the circuit responded by giving a higher value (note again that yes, `-5.9997` is *higher* than `-6`: `-5.9997 > -6`). The division by `h` is there to normalize the circuit's response by the (arbitrary) value of `h` we chose to use here. Technically, you want the value of `h` to be infinitesimal (the precise mathematical definition of the gradient is defined as the limit of the expression as `h` goes to zero), but in practice `h=0.00001` or so works fine in most cases to get a good approximation. Now, we see that the derivative w.r.t. `x` is `+3`. I'm making the positive sign explicit, because it indicates that the circuit is tugging on x to become higher. The actual value, `3` can be interpreted as the *force* of that tug.

> *The derivative with respect to some input can be computed by tweaking that input by a small amount and observing the change on the output value.*

By the way, we usually talk about the *derivative* with respect to a single input, or about a **gradient** with respect to all the inputs. The gradient is just made up of the derivatives of all the inputs concatenated in a vector (i.e. a list). Crucially, notice that if we let the inputs respond to the tug by following the gradient a tiny amount (i.e. we just add the derivative on top of every input), we can see that the value increases, as expected:

```
var step_size = 0.01;
var out = forwardMultiplyGate(x, y); // before: -6
x = x + step_size * x_derivative; // x becomes -1.97
y = y + step_size * y_derivative; // y becomes 2.98
var out_new = forwardMultiplyGate(x, y); // -5.87! exciting.
```

As expected, we changed the inputs by the gradient and the circuit now gives a slightly higher value (`-5.87 > -6.0`). That was much simpler than trying random changes to `x` and `y`, right? A fact to appreciate here is that if you take calculus you can prove that the gradient is, in fact, the direction of the steepest increase of the function. There is no need to monkey around trying out random pertubations as done in Strategy #1. Evaluating the gradient requires just three evaluations of the forward pass of our circuit instead of hundreds, and gives the best tug you can hope for (locally) if you are interested in increasing the value of the output.

Bigger step is not always better. Let me clarify on this point a bit. It is important to note that in this very simple example, using a bigger `step_size` than 0.01 will always work better. For example, `step_size = 1.0` gives output `-1` (higer, better!), and indeed infinite step size would give infinitely good results. The crucial thing to realize is that once our circuits get much more complex (e.g. entire neural networks), the function from inputs to the output value will be more chaotic and wiggly. The gradient guarantees that if you have a very small (indeed, infinitesimally small) step size, then you will definitely get a higher number when you follow its direction, and for that infinitesimally small step size there is no other direction that would have worked better. But if you use a bigger step size (e.g. `step_size = 0.01`) all bets are off. The reason we can get away with a larger step size than infinitesimally small is that our functions are usually relatively smooth. But really, we're crossing our fingers and hoping for the best.

Hill-climbing analogy. One analogy I've heard before is that the output value of our circut is like the height of a hill, and we are blindfolded and trying to climb upwards. We can sense the steepness of the hill at our feet (the gradient), so when we shuffle our feet a bit we will go upwards. But if we took a big, overconfident step, we could have stepped right into a hole.

Great, I hope I've convinced you that the numerical gradient is indeed a very useful thing to evaluate, and that it is cheap. But. It turns out that we can do *even* better.

Strategy #3: Analytic Gradient

In the previous section we evaluated the gradient by probing the circuit's output value, independently for every input. This procedure gives you what we call a **numerical gradient**. This approach, however, is *still* expensive because we need to compute the circuit's output as we tweak every input value independently a small amount. So the complexity of evaluating the gradient is linear in number of inputs. But in practice we will have hundreds, thousands or (for neural networks) even tens to hundreds of millions of inputs, and the circuits aren't just one multiply gate but huge expressions that can be expensive to compute. We need something better.

Luckily, there is an easier and *much* faster way to compute the gradient: we can use calculus to derive a direct expression for it that will be as simple to evaluate as the circuit's output value. We call this an **analytic gradient** and there will be no need for tweaking anything. You may have seen other people who teach Neural Networks derive the gradient in huge and, frankly, scary and confusing mathematical equations (if you're not well-versed in maths). But it's unnecessary. I've written plenty of Neural Nets code and I rarely have to do mathematical derivation longer than two lines, and 95% of the time it can be done without writing anything at all. That is because we will only ever derive the gradient for very small and simple expressions (think of it as the **base case**) and then I will show you how we can compose these very simply with **chain rule** to evaluate the full gradient (think inductive/recursive case).

> *The analytic derivative requires no tweaking of the inputs. It can be derived using mathematics (calculus).*

If you remember your product rules, power rules, quotient rules, etc. (see e.g. derivative rules or wiki page), it's very easy to write down the derivitative with respect to both `x` and `y` for a small expression such as `x * y`. But suppose you don't remember your calculus rules. We can go back to the definition. For example, here's the expression for the derivative w.r.t `x`:

$$\frac{\partial f(x,y)}{\partial x} = \frac{f(x+h,y) - f(x,y)}{h}$$

(Technically I'm not writing the limit as `h` goes to zero, forgive me math people). Okay and lets plug in our function ($f(x,y) = xy$) into the expression. Ready for the hardest piece of math of this entire article? Here we go:

$$\frac{\partial f(x,y)}{\partial x} = \frac{f(x+h,y) - f(x,y)}{h} = \frac{(x+h)y - xy}{h} = \frac{xy + hy - xy}{h} = \frac{hy}{h} = y$$

That's interesting. The derivative with respect to `x` is just equal to `y`. Did you notice the coincidence in the previous section? We tweaked `x` to `x+h` and calculated `x_derivative = 3.0`, which exactly happens to be the value of `y` in that example. It turns out that wasn't a coincidence at all because that's just what the analytic gradient tells us the `x` derivative should be for `f(x,y) = x * y`. The derivative with respect to `y`, by the way, turns out to be `x`, unsurprisingly by symmetry. So there is no need for any tweaking! We invoked powerful mathematics and can now transform our derivative calculation into the following code:

```
var x = -2, y = 3;
var out = forwardMultiplyGate(x, y); // before: -6
var x_gradient = y; // by our complex mathematical derivation above
var y_gradient = x;
var step_size = 0.01;
x += step_size * x_gradient; // -1.97
y += step_size * y_gradient; // 2.98
var out_new = forwardMultiplyGate(x, y); // -5.87. Higher output! Nice.
```

To compute the gradient we went from forwarding the circuit hundreds of times (Strategy #1) to forwarding it only on order of number of times twice the number of inputs (Strategy #2), to forwarding it a single time! And it gets EVEN better, since the more expensive strategies (#1 and #2) only give an approximation of the gradient, while #3 (the fastest one by far) gives you the *exact* gradient. No approximations. The only downside is that you should be comfortable with some calculus 101.

Lets recap what we have learned:

- INPUT: We are given a circuit, some inputs and compute an output value.
- OUTPUT: We are then interested finding small changes to each input (independently) that would make the output higher.
- Strategy #1: One silly way is to **randomly search** for small pertubations of the inputs and keep track of what gives the highest increase in output.
- Strategy #2: We saw we can do much better by computing the gradient. Regardless of how complicated the circuit is, the **numerical gradient** is very simple (but relatively expensive) to compute. We compute it by *probing* the circuit's output value as we tweak the inputs one at a time.
- Strategy #3: In the end, we saw that we can be even more clever and analytically derive a direct expression to get the **analytic gradient**. It is identical to the numerical gradient, it is fastest by far, and there is no need for any tweaking.

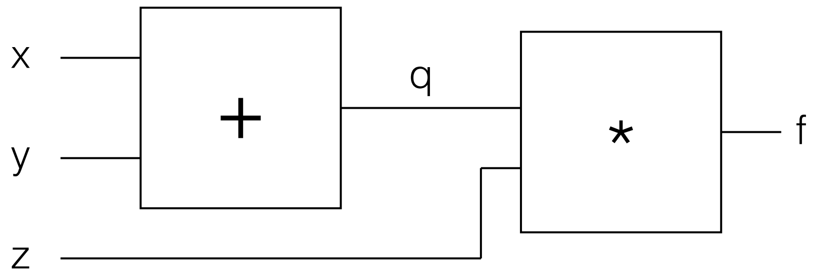
In practice by the way (and we will get to this once again later), all Neural Network libraries always compute the analytic gradient, but the correctness of the implementation is verified by comparing it to the numerical gradient. That's because the numerical gradient is very easy to evaluate (but can be a bit expensive to compute), while the analytic gradient can contain bugs at times, but is usually extremely efficient to compute. As we will see, evaluating the gradient (i.e. while doing *backprop*, or *backward pass*) will turn out to cost about as much as evaluating the *forward pass*.

Recursive Case: Circuits with Multiple Gates

But hold on, you say: *"The analytic gradient was trivial to derive for your super-simple expression. This is useless. What do I do when the expressions are much larger? Don't the equations get huge and complex very fast?"*. Good question. Yes the expressions get much more complex. No, this doesn't make it much harder. As we will see, every gate will be hanging out by itself, completely unaware of any details of the huge and complex circuit that it could be part of. It will only worry about its inputs and it will compute its local derivatives as seen in the previous section, except now there will be a single extra multiplication it will have to do.

> *A single extra multiplication will turn a single (useless gate) into a cog in the complex machine that is an entire neural network.*

I should stop hyping it up now. I hope I've piqued your interest! Lets drill down into details and get two gates involved with this next example:



The expression we are computing now is $f(x,y,z) = (x+y)z$. Lets structure the code as follows to make the gates explicit as functions:

```
var forwardMultiplyGate = function(a, b) {
  return a * b;
};
var forwardAddGate = function(a, b) {
  return a + b;
};
var forwardCircuit = function(x,y,z) {
  var q = forwardAddGate(x, y);
  var f = forwardMultiplyGate(q, z);
  return f;
};

var x = -2, y = 5, z = -4;
var f = forwardCircuit(x, y, z); // output is -12
```

In the above, I am using `a` and `b` as the local variables in the gate functions so that we don't get these confused with our circuit inputs `x,y,z`. As before, we are interested in finding the derivatives with respect to the three inputs `x,y,z`. But how do we compute it now that there are multiple gates involved? First, lets pretend that the `+` gate is not there and that we only have two variables in the circuit: `q,z` and a single `*` gate. Note that the `q` is is output of the `+` gate. If we don't worry about `x` and `y` but only about `q` and `z`, then we are back to having only a single gate, and as far as that single `*` gate is concerned, we know what the (analytic) derivates are from previous section. We can write them down (except here we're replacing `x,y` with `q,z`):

$$f(q,z) = qz \implies \frac{\partial f(q,z)}{\partial q} = z, \qquad \frac{\partial f(q,z)}{\partial z} = q$$

Simple enough: these are the expressions for the gradient with respect to `q` and `z`. But wait, we don't want gradient with respect to `q`, but with respect to the inputs: `x` and `y`. Luckily, `q` is computed as a function of `x` and `y` (by addition in our example). We can write down the gradient for the addition gate as well, it's even simpler:

$$q(x,y) = x + y \implies \frac{\partial q(x,y)}{\partial x} = 1, \qquad \frac{\partial q(x,y)}{\partial y} = 1$$

That's right, the derivatives are just 1, regardless of the actual values of `x` and `y`. If you think about it, this makes sense because to make the output of a single addition gate higher, we expect a positive tug on both `x` and `y`, regardless of their values.

Backpropagation

We are finally ready to invoke the **Chain Rule**: We know how to compute the gradient of `q` with respect to `x` and `y` (that's a single gate case with `+` as the gate). And we know how to compute the gradient of our final output with respect to `q`. The chain rule tells us how to combine these to get the gradient of the final output with respect to `x` and `y`, which is what we're ultimately interested in. Best of all, the chain rule very simply states that the right thing to do is to simply multiply the gradients together to chain them. For example, the final derivative for `x` will be:

$$\frac{\partial f(q,z)}{\partial x} = \frac{\partial q(x,y)}{\partial x} \frac{\partial f(q,z)}{\partial q}$$

There are many symbols there so maybe this is confusing again, but it's really just two numbers being multiplied together. Here is the code:

```
// initial conditions
var x = -2, y = 5, z = -4;
var q = forwardAddGate(x, y); // q is 3
var f = forwardMultiplyGate(q, z); // output is -12
```

```
// gradient of the MULTIPLY gate with respect to its inputs
// wrt is short for "with respect to"
var derivative_f_wrt_z = q; // 3
var derivative_f_wrt_q = z; // -4

// derivative of the ADD gate with respect to its inputs
var derivative_q_wrt_x = 1.0;
var derivative_q_wrt_y = 1.0;

// chain rule
var derivative_f_wrt_x = derivative_q_wrt_x * derivative_f_wrt_q; // -4
var derivative_f_wrt_y = derivative_q_wrt_y * derivative_f_wrt_q; // -4
```

That's it. We computed the gradient (the forces) and now we can let our inputs respond to it by a bit. Lets add the gradients on top of the inputs. The output value of the circuit better increase, up from -12!

```
// final gradient, from above: [-4, -4, 3]
var gradient_f_wrt_xyz = [derivative_f_wrt_x, derivative_f_wrt_y, derivati

// let the inputs respond to the force/tug:
var step_size = 0.01;
x = x + step_size * derivative_f_wrt_x; // -2.04
y = y + step_size * derivative_f_wrt_y; // 4.96
z = z + step_size * derivative_f_wrt_z; // -3.97

// Our circuit now better give higher output:
var q = forwardAddGate(x, y); // q becomes 2.92
var f = forwardMultiplyGate(q, z); // output is -11.59, up from -12! Nice!
```

Looks like that worked! Lets now try to interpret intuitively what just happened. The circuit wants to output higher values. The last gate saw inputs `q = 3, z = -4` and computed output `-12`. "Pulling" upwards on this output value induced a force on both `q` and `z`: To increase the output value, the circuit "wants" `z` to increase, as can be seen by the positive value of the derivative(`derivative_f_wrt_z = +3`). Again, the size of this derivative can be interpreted as the magnitude of the force. On the other hand, `q` felt a stronger and downward force, since `derivative_f_wrt_q = -4`. In other words the circuit wants `q` to decrease, with a force of `4`.

Now we get to the second, `+` gate which outputs `q`. By default, the `+` gate computes its derivatives which tells us how to change `x` and `y` to make `q` higher. BUT! Here is the

crucial point: the gradient on `q` was computed as negative (`derivative_f_wrt_q = -4`), so the circuit wants `q` to *decrease*, and with a force of `4`! So if the `+` gate wants to contribute to making the final output value larger, it needs to listen to the gradient signal coming from the top. In this particular case, it needs to apply tugs on `x,y` opposite of what it would normally apply, and with a force of `4`, so to speak. The multiplication by `-4` seen in the chain rule achieves exactly this: instead of applying a positive force of `+1` on both `x` and `y` (the local derivative), the full circuit's gradient on both `x` and `y` becomes `1 x -4 = -4`. This makes sense: the circuit wants both `x` and `y` to get smaller because this will make `q` smaller, which in turn will make `f` larger.

> *If this makes sense, you understand backpropagation.*

Lets **recap** once again what we learned:

- In the previous chapter we saw that in the case of a single gate (or a single expression), we can derive the analytic gradient using simple calculus. We interpreted the gradient as a force, or a tug on the inputs that pulls them in a direction which would make this gate's output higher.
- In case of multiple gates everything stays pretty much the same way: every gate is hanging out by itself completely unaware of the circuit it is embedded in. Some inputs come in and the gate computes its output and the derivate with respect to the inputs. The *only* difference now is that suddenly, something can pull on this gate from above. That's the gradient of the final circuit output value with respect to the ouput this gate computed. It is the circuit asking the gate to output higher or lower numbers, and with some force. The gate simply takes this force and multiplies it to all the forces it computed for its inputs before (chain rule). This has the desired effect:

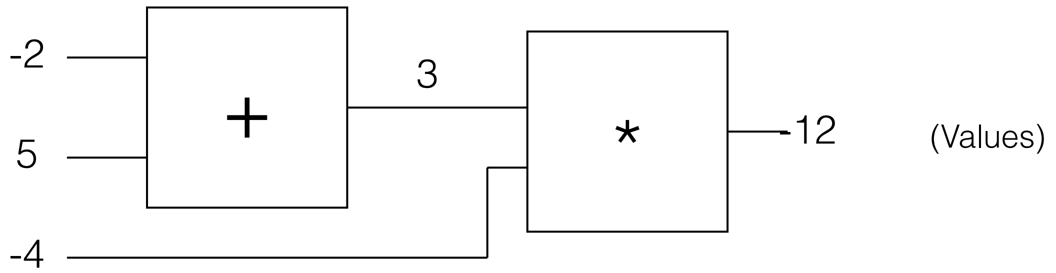
1. If a gate experiences a strong positive pull from above, it will also pull harder on its own inputs, scaled by the force it is experiencing from above
2. And if it experiences a negative tug, this means that circuit wants its value to decrease not increase, so it will flip the force of the pull on its inputs to make its own output value smaller.

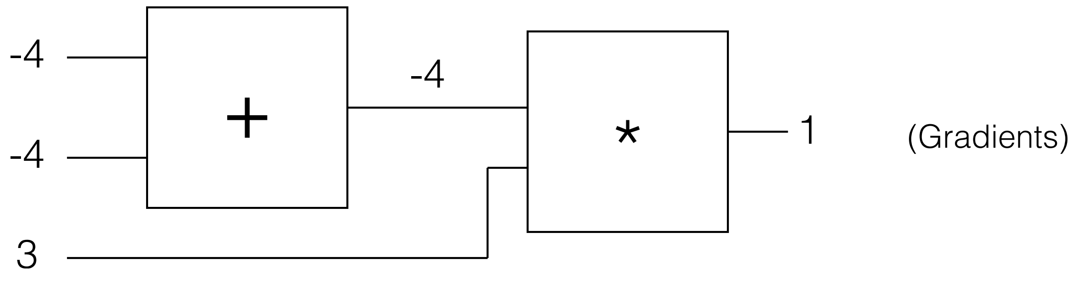
> *A nice picture to have in mind is that as we pull on the circuit's output value at the end, this induces pulls downward through the entire circuit, all the way down to the inputs.*

Isn't it beautiful? The only difference between the case of a single gate and multiple interacting gates that compute arbitrarily complex expressions is this additional multipy operation that now happens in each gate.

Patterns in the "backward" flow

Lets look again at our example circuit with the numbers filled in. The first circuit shows the raw values, and the second circuit shows the gradients that flow back to the inputs as discussed. Notice that the gradient always starts off with `+1` at the end to start off the chain. This is the (default) pull on the circuit to have its value increased.





After a while you start to notice patterns in how the gradients flow backward in the circuits. For example, the `+` gate always takes the gradient on top and simply passes it on to all of its inputs (notice the example with -4 simply passed on to both of the inputs of `+` gate). This is because its own derivative for the inputs is just `+1`, regardless of what the actual values of the inputs are, so in the chain rule, the gradient from above is just multiplied by 1 and stays the same. Similar intuitions apply to, for example, a `max(x,y)` gate. Since the gradient of `max(x,y)` with respect to its input is `+1` for whichever one of `x`, `y` is larger and `0` for the other, this gate is during backprop effectively just a gradient "switch": it will take the gradient from above and "route" it to the input that had a higher value during the forward pass.

Numerical Gradient Check. Before we finish with this section, lets just make sure that the (analytic) gradient we computed by backprop above is correct as a sanity check. Remember that we can do this simply by computing the numerical gradient and making sure that we get `[-4, -4, 3]` for `x,y,z`. Here's the code:

```
// initial conditions
var x = -2, y = 5, z = -4;
```

```
// numerical gradient check
var h = 0.0001;
var x_derivative = (forwardCircuit(x+h,y,z) - forwardCircuit(x,y,z)) / h;
var y_derivative = (forwardCircuit(x,y+h,z) - forwardCircuit(x,y,z)) / h;
var z_derivative = (forwardCircuit(x,y,z+h) - forwardCircuit(x,y,z)) / h;
```

and we get `[-4, -4, 3]`, as computed with backprop. phew! :)

Example: Single Neuron

In the previous section you hopefully got the basic intuition behind backpropagation. Lets now look at an even more complicated and borderline practical example. We will consider a 2-dimensional neuron that computes the following function:

$$f(x, y, a, b, c) = \sigma(ax + by + c)$$

In this expression, σ is the *sigmoid* function. Its best thought of as a "squashing function", because it takes the input and squashes it to be between zero and one: Very negative values are squashed towards zero and positive values get squashed towards one. For example, we have `sig(-5) = 0.006, sig(0) = 0.5, sig(5) = 0.993`. Sigmoid function is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The gradient with respect to its single input, as you can check on Wikipedia or derive yourself if you know some calculus is given by this expression:

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$

For example, if the input to the sigmoid gate is `x = 3`, the gate will compute output `f = 1.0 / (1.0 + Math.exp(-x)) = 0.95`, and then the (local) gradient on its input will simply be `dx = (0.95) * (1 - 0.95) = 0.0475`.

That's all we need to use this gate: we know how to take an input and *forward* it through the sigmoid gate, and we also have the expression for the gradient with respect to its input, so we can also *backprop* through it. Another thing to note is that technically, the sigmoid function is made up of an entire series of gates in a line that compute more *atomic* functions: an exponentiation gate, an addition gate and a division gate. Treating it so would work perfectly fine but for this example I chose to collapse all of these gates into a single gate that just computes sigmoid in one shot, because the gradient expression turns out to be simple.

Lets take this opportunity to carefully structure the associated code in a nice and modular way. First, I'd like you to note that every **wire** in our diagrams has two numbers associated with it:

1. the value it carries during the forward pass
2. the gradient (i.e the *pull*) that flows back through it in the backward pass

Lets create a simple `Unit` structure that will store these two values on every wire. Our gates will now operate over `Unit`s: they will take them as inputs and create them as outputs.

```
// every Unit corresponds to a wire in the diagrams
var Unit = function(value, grad) {
  // value computed in the forward pass
  this.value = value;
  // the derivative of circuit output w.r.t this unit, computed in backwar
  this.grad = grad;
}
```

In addition to Units we also need 3 gates: `+`, `*` and `sig` (sigmoid). Lets start out by implementing a multiply gate. I'm using Javascript here which has a funny way of simulating classes using functions. If you're not a Javascript - familiar person, all that's going on here is that I'm defining a class that has certain properties (accessed with use of `this` keyword), and some methods (which in Javascript are placed into the function's *prototype*). Just think about these as class methods. Also keep in mind that the way we will use these eventually is that we will first `forward` all the gates one by one, and then `backward` all the gates in reverse order. Here is the implementation:

```

var multiplyGate = function(){ };
multiplyGate.prototype = {
  forward: function(u0, u1) {
    // store pointers to input Units u0 and u1 and output unit utop
    this.u0 = u0;
    this.u1 = u1;
    this.utop = new Unit(u0.value * u1.value, 0.0);
    return this.utop;
  },
  backward: function() {
    // take the gradient in output unit and chain it with the
    // local gradients, which we derived for multiply gate before
    // then write those gradients to those Units.
    this.u0.grad += this.u1.value * this.utop.grad;
    this.u1.grad += this.u0.value * this.utop.grad;
  }
}
```

The multiply gate takes two units that each hold a value and creates a unit that stores its output. The gradient is initialized to zero. Then notice that in the `backward` function call we get the gradient from the output unit we produced during the forward pass (which will by now hopefully have its gradient filled in) and multiply it with the local gradient for this gate (chain rule!). This gate computes multiplication (`u0.value * u1.value`) during forward pass, so recall that the gradient w.r.t `u0` is `u1.value` and w.r.t `u1` is `u0.value`. Also note that we are using `+=` to add onto the gradient in the `backward` function. This will allow us to possibly use the output of one gate multiple times (think of it as a wire branching out), since it turns out that the gradients from these different branches just add up when computing the final gradient with respect to the circuit output. The other two gates are defined analogously:

```
var addGate = function(){ };
addGate.prototype = {
  forward: function(u0, u1) {
    this.u0 = u0;
    this.u1 = u1; // store pointers to input units
    this.utop = new Unit(u0.value + u1.value, 0.0);
    return this.utop;
  },
  backward: function() {
    // add gate. derivative wrt both inputs is 1
    this.u0.grad += 1 * this.utop.grad;
    this.u1.grad += 1 * this.utop.grad;
  }
}
```

```
var sigmoidGate = function() {
  // helper function
  this.sig = function(x) { return 1 / (1 + Math.exp(-x)); };
};
sigmoidGate.prototype = {
  forward: function(u0) {
    this.u0 = u0;
    this.utop = new Unit(this.sig(this.u0.value), 0.0);
    return this.utop;
  },
  backward: function() {
    var s = this.sig(this.u0.value);
    this.u0.grad += (s * (1 - s)) * this.utop.grad;
  }
}
```

Note that, again, the `backward` function in all cases just computes the local derivative with respect to its input and then multiplies on the gradient from the unit above (i.e. chain rule). To fully specify everything lets finally write out the forward and backward flow for our 2-dimensional neuron with some example values:

```
// create input units
var a = new Unit(1.0, 0.0);
var b = new Unit(2.0, 0.0);
var c = new Unit(-3.0, 0.0);
var x = new Unit(-1.0, 0.0);
var y = new Unit(3.0, 0.0);

// create the gates
var mulg0 = new multiplyGate();
var mulg1 = new multiplyGate();
var addg0 = new addGate();
var addg1 = new addGate();
var sg0 = new sigmoidGate();

// do the forward pass
var forwardNeuron = function() {
  ax = mulg0.forward(a, x); // a*x = -1
  by = mulg1.forward(b, y); // b*y = 6
  axpby = addg0.forward(ax, by); // a*x + b*y = 5
  axpbypc = addg1.forward(axpby, c); // a*x + b*y + c = 2
  s = sg0.forward(axpbypc); // sig(a*x + b*y + c) = 0.8808
};
forwardNeuron();

console.log('circuit output: ' + s.value); // prints 0.8808
```

And now lets compute the gradient: Simply iterate in reverse order and call the `backward` function! Remember that we stored the pointers to the units when we did the forward pass, so every gate has access to its inputs and also the output unit it previously produced.

```
s.grad = 1.0;
sg0.backward(); // writes gradient into axpbypc
addg1.backward(); // writes gradients into axpby and c
addg0.backward(); // writes gradients into ax and by
```

```
mulg1.backward(); // writes gradients into b and y
mulg0.backward(); // writes gradients into a and x
```

Note that the first line sets the gradient at the output (very last unit) to be `1.0` to start off the gradient chain. This can be interpreted as tugging on the last gate with a force of `+1`. In other words, we are pulling on the entire circuit to induce the forces that will increase the output value. If we did not set this to 1, all gradients would be computed as zero due to the multiplications in the chain rule. Finally, lets make the inputs respond to the computed gradients and check that the function increased:

```
var step_size = 0.01;
a.value += step_size * a.grad; // a.grad is -0.105
b.value += step_size * b.grad; // b.grad is 0.315
c.value += step_size * c.grad; // c.grad is 0.105
x.value += step_size * x.grad; // x.grad is 0.105
y.value += step_size * y.grad; // y.grad is 0.210

forwardNeuron();
console.log('circuit output after one backprop: ' + s.value); // prints 0.
```

Success! `0.8825` is higher than the previous value, `0.8808`. Finally, lets verify that we implemented the backpropagation correctly by checking the numerical gradient:

```
var forwardCircuitFast = function(a,b,c,x,y) {
  return 1/(1 + Math.exp( - (a*x + b*y + c)));
};
var a = 1, b = 2, c = -3, x = -1, y = 3;
var h = 0.0001;
var a_grad = (forwardCircuitFast(a+h,b,c,x,y) - forwardCircuitFast(a,b,c,x
var b_grad = (forwardCircuitFast(a,b+h,c,x,y) - forwardCircuitFast(a,b,c,x
var c_grad = (forwardCircuitFast(a,b,c+h,x,y) - forwardCircuitFast(a,b,c,x
var x_grad = (forwardCircuitFast(a,b,c,x+h,y) - forwardCircuitFast(a,b,c,x
var y_grad = (forwardCircuitFast(a,b,c,x,y+h) - forwardCircuitFast(a,b,c,x
```

Indeed, these all give the same values as the backpropagated gradients `[-0.105, 0.315, 0.105, 0.105, 0.210]`. Nice!

I hope it is clear that even though we only looked at an example of a single neuron, the code I gave above generalizes in a very straight-forward way to compute gradients of arbitrary expressions (including very deep expressions #foreshadowing). All you have to do is write small gates that compute local, simple derivatives w.r.t their inputs, wire it up in a graph, do a forward pass to compute the output value and then a backward pass that chains the gradients all the way to the input.

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:

```
// lets 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 $\frac{\partial f(a)}{\partial a} = 2a$, 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 chai
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.

Chapter 2: Machine Learning

In the last chapter we were concerned with real-valued circuits that computed possibly complex expressions of their inputs (the forward pass), and also we could compute the gradients of these expressions on the original inputs (backward pass). In this chapter we will see how useful this extremely simple mechanism is in Machine Learning.

Binary Classification

As we did before, lets start out simple. The simplest, common and yet very practical problem in Machine Learning is **binary classification**. A lot of very interesting and important problems can be reduced to it. The setup is as follows: We are given a dataset of `N` vectors and every one of them is labeled with a `+1` or a `-1`. For example, in two dimensions our dataset could look as simple as:

```
vector -> label
---------------
[1.2, 0.7] -> +1
[-0.3, 0.5] -> -1
[-3, -1] -> +1
[0.1, 1.0] -> -1
[3.0, 1.1] -> -1
[2.1, -3] -> +1
```

Here, we have `N = 6` **datapoints**, where every datapoint has two **features** (`D = 2`). Three of the datapoints have **label** `+1` and the other three label `-1`. This is a silly toy example, but in practice a +1/-1 dataset could be very useful things indeed: For example spam/no spam emails, where the vectors somehow measure various features of the content of the email, such as the number of times certain enhancement drugs are mentioned.

Goal. Our goal in binary classification is to learn a function that takes a 2-dimensional vector and predicts the label. This function is usually parameterized by a certain set of parameters, and we will want to tune the parameters of the function so that its outputs are consistent with the labeling in the provided dataset. In the end we can discard the dataset and use the learned parameters to predict labels for previously unseen vectors.

Training protocol

We will eventually build up to entire neural networks and complex expressions, but lets start out simple and train a linear classifier very similar to the single neuron we saw at the end of Chapter 1. The only difference is that we'll get rid of the sigmoid because it makes things unnecessarily complicated (I only used it as an example in Chapter 1 because sigmoid neurons are historically popular but modern Neural Networks rarely, if ever, use sigmoid non-linearities). Anyway, lets use a simple linear function:

$$f(x, y) = ax + by + c$$

In this expression we think of `x` and `y` as the inputs (the 2D vectors) and `a,b,c` as the parameters of the function that we will want to learn. For example, if `a = 1, b = -2, c = -1`, then the function will take the first datapoint (`[1.2, 0.7]`) and output `1 * 1.2 + (-2) * 0.7 + (-1) = -1.2`. Here is how the training will work:

1. We select a random datapoint and feed it through the circuit
2. We will interpret the output of the circuit as a confidence that the datapoint has class `+1`. (i.e. very high values = circuit is very certain datapoint has class `+1` and very low values = circuit is certain this datapoint has class `-1`.)
3. We will measure how well the prediction aligns with the provided labels. Intuitively, for example, if a positive example scores very low, we will want to tug in the positive direction on the circuit, demanding that it should output higher value for this datapoint. Note that this is the case for the the first datapoint: it is labeled as `+1` but our predictor unction only assigns it value `-1.2`. We will therefore tug on the circuit in positive direction; We want the value to be higher.
4. The circuit will take the tug and backpropagate it to compute tugs on the inputs `a,b,c,x,y`
5. Since we think of `x,y` as (fixed) datapoints, we will ignore the pull on `x,y`. If you're a fan of my physical analogies, think of these inputs as pegs, fixed in the ground.
6. On the other hand, we will take the parameters `a,b,c` and make them respond to their tug (i.e. we'll perform what we call a **parameter update**). This, of course, will make it so that the circuit will output a slightly higher score on this particular datapoint in the future.
7. Iterate! Go back to step 1.

The training scheme I described above, is commonly referred as **Stochastic Gradient Descent**. The interesting part I'd like to reiterate is that `a,b,c,x,y` are all made up of the same *stuff* as far as the circuit is concerned: They are inputs to the circuit and the circuit will tug on all of them in some direction. It doesn't know the difference between parameters and datapoints. However, after the backward pass is complete we ignore all tugs on the datapoints (`x,y`) and keep swapping them in and out as we iterate over examples in the dataset. On the other hand, we keep the parameters (`a,b,c`) around and keep tugging on them every time we sample a datapoint. Over time, the pulls on these parameters will tune these values in such a way that the function outputs high scores for positive examples and low scores for negative examples.

Learning a Support Vector Machine

As a concrete example, lets learn a **Support Vector Machine**. The SVM is a very popular linear classifier; Its functional form is exactly as I've described in previous section, $f(x, y) = ax + by + c$. At this point, if you've seen an explanation of SVMs you're probably expecting me to define the SVM loss function and plunge into an explanation of slack variables, geometrical intuitions of large margins, kernels, duality, etc. But here, I'd like to take a different approach. Instead of definining loss functions, I would like to base the explanation on the *force specification* (I just made this term up by the way) of a Support Vector Machine, which I personally find much more intuitive. As we will see, talking about the force specification and the loss function are identical ways of seeing the same problem. Anyway, here it is:

Support Vector Machine "Force Specification":

- If we feed a positive datapoint through the SVM circuit and the output value is less than 1, pull on the circuit with force `+1`. This is a positive example so we want the score to be higher for it.
- Conversely, if we feed a negative datapoint through the SVM and the output is greater than -1, then the circuit is giving this datapoint dangerously high score: Pull on the circuit downwards with force `-1`.
- In addition to the pulls above, always add a small amount of pull on the parameters `a,b` (notice, not on `c`!) that pulls them towards zero. You can think of both `a,b` as being attached to a physical spring that is attached at zero. Just as with a physical spring, this will make the pull proprotional to the value of each of `a,b` (Hooke's law in physics, anyone?). For example, if `a` becomes very high it will experience a strong pull of magnitude `|a|` back towards zero. This pull is something we call **regularization**, and it ensures that neither of our parameters `a` or `b` gets disproportionally large. This would be undesirable because both `a,b` get multiplied to the input features `x,y` (remember the equation is `a*x + b*y + c`), so if either of them is too high, our classifier would be overly sensitive to these features. This isn't a nice property because features can often be noisy in practice, so we want our classifier to change relatively smoothly if they wiggle around.

Lets quickly go through a small but concrete example. Suppose we start out with a random parameter setting, say, `a = 1, b = -2, c = -1`. Then:

- If we feed the point `[1.2, 0.7]`, the SVM will compute score `1 * 1.2 + (-2) * 0.7 - 1 = -1.2`. This point is labeled as `+1` in the training data, so we want the score to be higher than 1. The gradient on top of the circuit will thus be positive: `+1`, which will backpropagate to `a,b,c`. Additionally, there will also be a regularization pull on `a` of `-1` (to make it smaller) and regularization pull on `b` of `+2` to make it larger, toward zero.
- Suppose instead that we fed the datapoint `[-0.3, 0.5]` to the SVM. It computes `1 * (-0.3) + (-2) * 0.5 - 1 = -2.3`. The label for this point is `-1`, and since `-2.3` is smaller than `-1`, we see that according to our force specification the SVM should be happy: The computed score is very negative, consistent with the negative label of this

example. There will be no pull at the end of the circuit (i.e it's zero), since there no changes are necessary. However, there will *still* be the regularization pull on `a` of `-1` and on `b` of `+2`.

Okay there's been too much text. Lets write the SVM code and take advantage of the circuit machinery we have from Chapter 1:

```
// A circuit: it takes 5 Units (x,y,a,b,c) and outputs a single Unit
// It can also compute the gradient w.r.t. its inputs
var Circuit = function() {
  // create some gates
  this.mulg0 = new multiplyGate();
  this.mulg1 = new multiplyGate();
  this.addg0 = new addGate();
  this.addg1 = new addGate();
};
Circuit.prototype = {
  forward: function(x,y,a,b,c) {
    this.ax = this.mulg0.forward(a, x); // a*x
    this.by = this.mulg1.forward(b, y); // b*y
    this.axpby = this.addg0.forward(this.ax, this.by); // a*x + b*y
    this.axpbypc = this.addg1.forward(this.axpby, c); // a*x + b*y + c
    return this.axpbypc;
  },
  backward: function(gradient_top) { // takes pull from above
    this.axpbypc.grad = gradient_top;
    this.addg1.backward(); // sets gradient in axpby and c
    this.addg0.backward(); // sets gradient in ax and by
    this.mulg1.backward(); // sets gradient in b and y
    this.mulg0.backward(); // sets gradient in a and x
  }
}
```

That's a circuit that simply computes `a*x + b*y + c` and can also compute the gradient. It uses the gates code we developed in Chapter 1. Now lets write the SVM, which doesn't care about the actual circuit. It is only concerned with the values that come out of it, and it pulls on the circuit.

```
// SVM class
var SVM = function() {

  // random initial parameter values
  this.a = new Unit(1.0, 0.0);
  this.b = new Unit(-2.0, 0.0);
  this.c = new Unit(-1.0, 0.0);

  this.circuit = new Circuit();
};
SVM.prototype = {
  forward: function(x, y) { // assume x and y are Units
    this.unit_out = this.circuit.forward(x, y, this.a, this.b, this.c);
    return this.unit_out;
  },
  backward: function(label) { // label is +1 or -1

    // reset pulls on a,b,c
    this.a.grad = 0.0;
    this.b.grad = 0.0;
    this.c.grad = 0.0;

    // compute the pull based on what the circuit output was
    var pull = 0.0;
    if(label === 1 && this.unit_out.value < 1) {
      pull = 1; // the score was too low: pull up
    }
    if(label === -1 && this.unit_out.value > -1) {
      pull = -1; // the score was too high for a positive example, pull do
    }
    this.circuit.backward(pull); // writes gradient into x,y,a,b,c

    // add regularization pull for parameters: towards zero and proportion
    this.a.grad += -this.a.value;
    this.b.grad += -this.b.value;
  },
  learnFrom: function(x, y, label) {
    this.forward(x, y); // forward pass (set .value in all Units)
    this.backward(label); // backward pass (set .grad in all Units)
    this.parameterUpdate(); // parameters respond to tug
  },
  parameterUpdate: function() {
    var step_size = 0.01;
    this.a.value += step_size * this.a.grad;
    this.b.value += step_size * this.b.grad;
    this.c.value += step_size * this.c.grad;
  }
};
```

Now lets train the SVM with Stochastic Gradient Descent:

```
var data = []; var labels = [];
data.push([1.2, 0.7]); labels.push(1);
data.push([-0.3, -0.5]); labels.push(-1);
data.push([3.0, 0.1]); labels.push(1);
data.push([-0.1, -1.0]); labels.push(-1);
data.push([-1.0, 1.1]); labels.push(-1);
data.push([2.1, -3]); labels.push(1);
var svm = new SVM();

// a function that computes the classification accuracy
var evalTrainingAccuracy = function() {
  var num_correct = 0;
  for(var i = 0; i < data.length; i++) {
    var x = new Unit(data[i][0], 0.0);
    var y = new Unit(data[i][1], 0.0);
    var true_label = labels[i];

    // see if the prediction matches the provided label
    var predicted_label = svm.forward(x, y).value > 0 ? 1 : -1;
    if(predicted_label === true_label) {
      num_correct++;
    }
  }
  return num_correct / data.length;
};

// the learning loop
for(var iter = 0; iter < 400; iter++) {
  // pick a random data point
  var i = Math.floor(Math.random() * data.length);
  var x = new Unit(data[i][0], 0.0);
  var y = new Unit(data[i][1], 0.0);
  var label = labels[i];
  svm.learnFrom(x, y, label);

  if(iter % 25 == 0) { // every 10 iterations...
    console.log('training accuracy at iter ' + iter + ': ' + evalTrainingA
  }
}
```

This code prints the following output:

```
training accuracy at iteration 0: 0.3333333333333333
training accuracy at iteration 25: 0.3333333333333333
training accuracy at iteration 50: 0.5
training accuracy at iteration 75: 0.5
training accuracy at iteration 100: 0.3333333333333333
training accuracy at iteration 125: 0.5
training accuracy at iteration 150: 0.5
training accuracy at iteration 175: 0.5
training accuracy at iteration 200: 0.5
training accuracy at iteration 225: 0.6666666666666666
training accuracy at iteration 250: 0.6666666666666666
training accuracy at iteration 275: 0.8333333333333334
training accuracy at iteration 300: 1
training accuracy at iteration 325: 1
training accuracy at iteration 350: 1
training accuracy at iteration 375: 1
```

We see that initially our classifier only had 33% training accuracy, but by the end all training examples are correctly classifier as the parameters `a,b,c` adjusted their values according to the pulls we exerted. We just trained an SVM! But please don't use this code anywhere in production :) We will see how we can make things much more efficient once we understand what is going on at the core.

Number of iterations needed. With this example data, with this example initialization, and with the setting of step size we used, it took about 300 iterations to train the SVM. In practice, this could be many more or many less depending on how hard or large the problem is, how you're initializating, normalizing your data, what step size you're using, and so on. This is just a toy demonstration, but later we will go over all the best practices for actually training these classifiers in practice. For example, it will turn out that the setting of the step size is very imporant and tricky. Small step size will make your model slow to train. Large step size will train faster, but if it is too large, it will make your classifier chaotically jump around and not converge to a good final result. We will eventually use witheld validation data to properly tune it to be just in the sweet spot for your particular data.

One thing I'd like you to appreciate is that the circuit can be arbitrary expression, not just the linear prediction function we used in this example. For example, it can be an entire neural network.

By the way, I intentionally structured the code in a modular way, but we could have trained an SVM with a much simpler code. Here is really what all of these classes and computations boil down to:

```
var a = 1, b = -2, c = -1; // initial parameters
for(var iter = 0; iter < 400; iter++) {
  // pick a random data point
  var i = Math.floor(Math.random() * data.length);
  var x = data[i][0];
  var y = data[i][1];
  var label = labels[i];

  // compute pull
  var score = a*x + b*y + c;
  var pull = 0.0;
  if(label === 1 && score < 1) pull = 1;
  if(label === -1 && score > -1) pull = -1;

  // compute gradient and update parameters
  var step_size = 0.01;
  a += step_size * (x * pull - a); // -a is from the regularization
  b += step_size * (y * pull - b); // -b is from the regularization
  c += step_size * (1 * pull);
}
```

this code gives an identical result. Perhaps by now you can glance at the code and see how these equations came about.

Variable pull? A quick note to make at this point: You may have noticed that the pull is always 1,0, or -1. You could imagine doing other things, for example making this pull proportional to how bad the mistake was. This leads to a variation on the SVM that some people refer to as *squared hinge loss* SVM, for reasons that will later become clear. Depending on various features of your dataset, that may work better or worse. For example, if you have very bad outliers in your data, e.g. a negative data point that gets a score `+100`, its influence will be relatively minor on our classifier because we will only pull with force of `-1` regardless of how bad the mistake was. In practice we refer to this property of a classifier as **robustness** to outliers.

Lets **recap**. We introduced the **binary classification** problem, where we are given N D-dimensional vectors and a label +1/-1 for each. We saw that we can combine these features with a set of parameters inside a real-valued circuit (such as a **Support Vector Machine** circuit in our example). Then, we can repeatedly pass our data through the circuit and each time tweak the parameters so that the circuit's output value is consistent with the provided labels. The tweaking relied, crucially, on our ability to **backpropagate** gradients through the circuit. In the end, the final circuit can be used to predict values for unseen instances!

Generalizing the SVM into a Neural Network

Of interest is the fact that an SVM is just a particular type of a very simple circuit (circuit that computes `score = a*x + b*y + c` where `a,b,c` are weights and `x,y` are data points). This can be easily extended to more complicated functions. For example, lets write a 2-layer Neural Network that does the binary classification. The forward pass will look like this:

```
// assume inputs x,y
var n1 = Math.max(0, a1*x + b1*y + c1); // activation of 1st hidden neuron
var n2 = Math.max(0, a2*x + b2*y + c2); // 2nd neuron
var n3 = Math.max(0, a3*x + b3*y + c3); // 3rd neuron
var score = a4*n1 + b4*n2 + c4*n3 + d4; // the score
```

The specification above is a 2-layer Neural Network with 3 hidden neurons (n1, n2, n3) that uses Rectified Linear Unit (ReLU) non-linearity on each hidden neuron. As you can see, there are now several parameters involved, which means that our classifier is more complex and can represent more intricate decision boundaries than just a simple linear decision rule such as an SVM. Another way to think about it is that every one of the three hidden neurons is a linear classifier and now we're putting an extra linear classifier on top of that. Now we're starting to go *deeper* :). Okay, lets train this 2-layer Neural Network. The code looks very similar to the SVM example code above, we just have to change the forward pass and the backward pass:

```
// random initial parameters
var a1 = Math.random() - 0.5; // a random number between -0.5 and 0.5
// ... similarly initialize all other parameters to randoms
for(var iter = 0; iter < 400; iter++) {
  // pick a random data point
  var i = Math.floor(Math.random() * data.length);
  var x = data[i][0];
  var y = data[i][1];
  var label = labels[i];

  // compute forward pass
  var n1 = Math.max(0, a1*x + b1*y + c1); // activation of 1st hidden neur
  var n2 = Math.max(0, a2*x + b2*y + c2); // 2nd neuron
  var n3 = Math.max(0, a3*x + b3*y + c3); // 3rd neuron
  var score = a4*n1 + b4*n2 + c4*n3 + d4; // the score

  // compute the pull on top
  var pull = 0.0;
  if(label === 1 && score < 1) pull = 1; // we want higher output! Pull up
  if(label === -1 && score > -1) pull = -1; // we want lower output! Pull
```

```
// now compute backward pass to all parameters of the model

// backprop through the last "score" neuron
var dscore = pull;
var da4 = n1 * dscore;
var dn1 = a4 * dscore;
var db4 = n2 * dscore;
var dn2 = b4 * dscore;
var dc4 = n3 * dscore;
var dn3 = c4 * dscore;
var dd4 = 1.0 * dscore; // phew

// backprop the ReLU non-linearities, in place
// i.e. just set gradients to zero if the neurons did not "fire"
var dn3 = n3 === 0 ? 0 : dn3;
var dn2 = n2 === 0 ? 0 : dn2;
var dn1 = n1 === 0 ? 0 : dn1;

// backprop to parameters of neuron 1
var da1 = x * dn1;
var db1 = y * dn1;
var dc1 = 1.0 * dn1;

// backprop to parameters of neuron 2
var da2 = x * dn2;
var db2 = y * dn2;
var dc2 = 1.0 * dn2;

// backprop to parameters of neuron 3
var da3 = x * dn3;
var db3 = y * dn3;
var dc3 = 1.0 * dn3;

// phew! End of backprop!
// note we could have also backpropped into x,y
// but we do not need these gradients. We only use the gradients
// on our parameters in the parameter update, and we discard x,y

// add the pulls from the regularization, tugging all multiplicative
// parameters (i.e. not the biases) downward, proportional to their valu
da1 += -a1; da2 += -a2; da3 += -a3;
db1 += -b1; db2 += -b2; db3 += -b3;
da4 += -a4; db4 += -b4; dc4 += -c4;
```

```
  // finally, do the parameter update
  var step_size = 0.01;
  a1 += step_size * da1;
  b1 += step_size * db1;
  c1 += step_size * dc1;
  a2 += step_size * da2;
  b2 += step_size * db2;
  c2 += step_size * dc2;
  a3 += step_size * da3;
  b3 += step_size * db3;
  c3 += step_size * dc3;
  a4 += step_size * da4;
  b4 += step_size * db4;
  c4 += step_size * dc4;
  d4 += step_size * dd4;
  // wow this is tedious, please use for loops in prod.
  // we're done!
}
```

And that's how you train a neural network. Obviously, you want to modularize your code nicely but I expended this example for you in the hope that it makes things much more concrete and simpler to understand. Later, we will look at best practices when implementing these networks and we will structure the code much more neatly in a modular and more sensible way.

But for now, I hope your takeaway is that a 2-layer Neural Net is really not such a scary thing: we write a forward pass expression, interpret the value at the end as a score, and then we pull on that value in a positive or negative direction depending on what we want that value to be for our current particular example. The parameter update after backprop will ensure that when we see this particular example in the future, the network will be more likely to give us a value we desire, not the one it gave just before the update.

A more Conventional Approach: Loss Functions

Now that we understand the basics of how these circuits function with data, lets adopt a more conventional approach that you might see elsewhere on the internet and in other tutorials and books. You won't see people talking too much about **force specifications**. Instead, Machine Learning algorithms are specified in terms of **loss functions** (or **cost functions**, or **objectives**).

As I develop this formalism I would also like to start to be a little more careful with how we name our variables and parameters. I'd like these equations to look similar to what you might see in a book or some other tutorial, so let me use more standard naming conventions.

Example: 2-D Support Vector Machine

Lets start with an example of a 2-dimensional SVM. We are given a dataset of N examples (x_{i0}, x_{i1}) and their corresponding labels y_i which are allowed to be either $+1/-1$ for positive or negative example respectively. Most importantly, as you recall we have three parameters (w_0, w_1, w_2). The SVM loss function is then defined as follows:

$$L = [\sum \_i = 1^N max(0, -y\_i(w\_0x\_i0 + w\_1x\_i1 + w\_2) + 1)] + \alpha[w\_0^2 + w\_1^2]$$

Notice that this expression is always positive, due to the thresholding at zero in the first expression and the squaring in the regularization. The idea is that we will want this expression to be as small as possible. Before we dive into some of its subtleties let me first translate it to code:

```
var X = [ [1.2, 0.7], [-0.3, 0.5], [3, 2.5] ] // array of 2-dimensional da
var y = [1, -1, 1] // array of labels
var w = [0.1, 0.2, 0.3] // example: random numbers
var alpha = 0.1; // regularization strength

function cost(X, y, w) {

  var total_cost = 0.0; // L, in SVM loss function above
  N = X.length;
  for(var i=0;i<N;i++) {
    // loop over all data points and compute their score
    var xi = X[i];
    var score = w[0] * xi[0] + w[1] * xi[1] + w[2];

    // accumulate cost based on how compatible the score is with the label
    var yi = y[i]; // label
    var costi = Math.max(0, - yi * score + 1);
    console.log('example ' + i + ': xi = (' + xi + ') and label = ' + yi);
    console.log('  score computed to be ' + score.toFixed(3));
    console.log('  => cost computed to be ' + costi.toFixed(3));
    total_cost += costi;
  }

  // regularization cost: we want small weights
  reg_cost = alpha * (w[0]*w[0] + w[1]*w[1])
  console.log('regularization cost for current model is ' + reg_cost.toFix
  total_cost += reg_cost;

  console.log('total cost is ' + total_cost.toFixed(3));
  return total_cost;
}
```

And here is the output:

```
cost for example 0 is 0.440
cost for example 1 is 1.370
cost for example 2 is 0.000
regularization cost for current model is 0.005
total cost is 1.815
```

Notice how this expression works: It measures how *bad* our SVM classifier is. Lets step through this explicitly:

- The first datapoint `xi = [1.2, 0.7]` with label `yi = 1` will give score `0.1*1.2 + 0.2*0.7 + 0.3`, which is `0.56`. Notice, this is a positive example so we want to the score to be greater than `+1`. `0.56` is not enough. And indeed, the expression for cost for this datapoint will compute: `costi = Math.max(0, -1*0.56 + 1)`, which is `0.44`. You can think of the cost as quantifying the SVM's unhappiness.
- The second datapoint `xi = [-0.3, 0.5]` with label `yi = -1` will give score `0.1*(-0.3) + 0.2*0.5 + 0.3`, which is `0.37`. This isn't looking very good: This score is very high for a negative example. It should be less than -1. Indeed, when we compute the cost: `costi = Math.max(0, 1*0.37 + 1)`, we get `1.37`. That's a very high cost from this example, as it is being misclassified.
- The last example `xi = [3, 2.5]` with label `yi = 1` gives score `0.1*3 + 0.2*2.5 + 0.3`, and that is `1.1`. In this case, the SVM will compute `costi = Math.max(0, -1*1.1 + 1)`, which is in fact zero. This datapoint is being classified correctly and there is no cost associated with it.

> *A cost function is an expression that measuress how bad your classifier is. When the training set is perfectly classified, the cost (ignoring the regularization) will be zero.*

Notice that the last term in the loss is the regularization cost, which says that our model parameters should be small values. Due to this term the cost will never actually become zero (because this would mean all parameters of the model except the bias are exactly zero), but the closer we get, the better our classifier will become.

> *The majority of cost functions in Machine Learning consist of two parts: 1. A part that measures how well a model fits the data, and 2: Regularization, which measures some notion of how complex or likely a model is.*

I hope I convinced you then, that to get a very good SVM we really want to make the **cost as small as possible**. Sounds familiar? We know exactly what to do: The cost function written above is our circuit. We will forward all examples through the circuit, compute the backward pass and update all parameters such that the circuit will output a *smaller* cost in the future. Specifically, we will compute the *gradient* and then update the parameters in the *opposite direction* of the gradient (since we want to make the cost small, not large).

> *"We know exactly what to do: The cost function written above is our circuit."*

todo: clean up this section and flesh it out a bit…

Chapter 3: Backprop in Practice

Building up a library

Example: Practical Neural Network Classifier

- Multiclass: Structured SVM
- Multiclass: Logistic Regression, Softmax

Example: Regression

Tiny changes needed to cost function. L2 regularization.

Example: Structured Prediction

Basic idea is to train an (unnormalized) energy model

Vectorized Implementations

Writing a Neural Net classfier in Python with numpy….

Backprop in practice: Tips/Tricks

- Monitoring of Cost function
- Monitoring training/validation performance
- Tweaking initial learning rates, learning rate schedules
- Optimization: Using Momentum
- Optimization: LBFGS, Nesterov accelerated gradient
- Importance of Initialization: weights and biases
- Regularization: L2, L1, Group sparsity, Dropout
- Hyperparameter search, cross-validations
- Common pitfalls: (e.g. dying ReLUs)
- Handling unbalanced datasets
- Approaches to debugging nets when something doesnt work

Chapter 4: Networks in the Wild

Case studies of models that work well in practice and have been deployed in the wild.

Case Study: Convolutional Neural Networks for images

Convolutional layers, pooling, AlexNet, etc.

Case Study: Recurrent Neural Networks for Speech and Text

Vanilla Recurrent nets, bi-directional recurrent nets. Maybe overview of LSTM

Case Study: Word2Vec

Training word vector representations in NLP

Case Study: t-SNE

Training embeddings for visualizing data

Acknowledgements