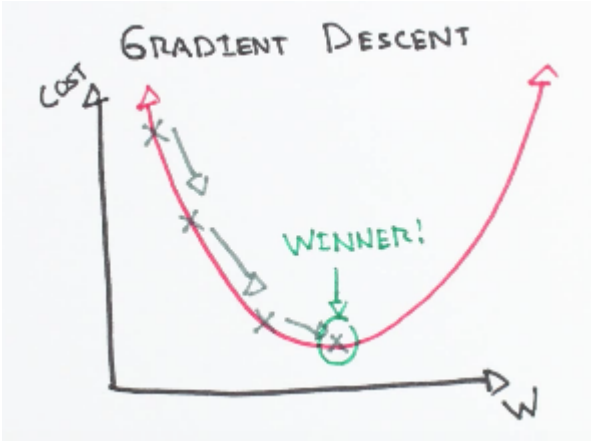
Gradient Descent Solution

Let's assume our input list contains only 4 numbers, for example, $[3, 5, 9, 6]$. Then our output list, a, contains 5 numbers, which we can write as $[a, b, c, d, e]$. We know that the score for this list is calculated like this:

$$\text{score} = |3 - ab| + |5 - bc| + |9 - cd| + |6 - de|$$

So we want to minimise the score by changing the values in the output list. At first, we just initialise the values in our output list to random numbers (or any arbitrary suitable value), but through the process I'm about to describe, these values converge on a good solution.

Since we have an equation for the score, we could just try *every* value for each number in the list (brute force) and pick the best. Well, if we did that, we could also plot a graph (in 6 dimensions) of the score for each possible set of values in the list. Let's say that score is our y value and a, b, c, d and e are like the values of x, but in multiple dimensions (so we each variable corresponds to an axis on the graph). So if we could find the lowest point on that graph, that point would be the solution of the problem, since it has the minimum score. We do this by gradient descent:



The gradient is the steepness, so a positive gradient indicates the value of score is rising, and a negative gradient indicates it's falling. If we *subtract* the gradient from our current value, we'd get closer to the bottom of the graph (see the image above). But how can we calculate the gradient for a list? It's not that much different from regular differentiation.

The main change we need to make is to our score function itself, since there's no good way of differentiating absolute values that I can be bothered to learn about. The absolute value is only there to make the score positive, so we'll achieve the same effect in a more gradient-friendly way... squaring. Here's the amended score:

$$\text{score} = (3 - ab)^2 + (5 - bc)^2 + (9 - cd)^2 + (6 - de)^2$$

It basically does the same thing, but now, we can differentiate it. This gets a bit tricky at some points but if we can do it for a list of 5 numbers, we can very easily generalise that to a list of any length.

With multivariable differentiation, we need to differentiate the score with respect to a, then differentiate it with respect to b, and so on - each value in the list gets *its own gradient*. This makes sense, because each value in the list corresponds to an axis or direction on our graph, and the gradient may be different along different axes (think about a 3D bowl-shaped graph for this).

The key thing to realise at this point is that changing the value of one variable does not change the value of another - they're completely separate. For example, if you change a, b remains the same, and so do the rest of the values in the list. And values that remain the same are called... constants! This means that if we're differentiating with respect to a, we can treat b like any constant, just like we would treat 2 or 5. In a way, it's like adjusting the sliders on an amplifier and seeing how the result changes. You move one slider up and down, keeping the others constant, to see what the effect is:

So now we're ready to differentiate. We'll do this 3 times, once for a, once for b and once for e, to see exactly how it works at different points in the array.

If we're differentiating with respect to a, we can ignore any of the brackets in our new score equation that don't involve a (remember, we can treat them like constants, and constants differentiate to 0). That leaves us with just the first bracket to worry about:

$$\frac{d}{da}\text{score} = \frac{d}{da}(3 - ab)^2$$

$$= \frac{d}{da}(9 - 6ab + a^2b^2)$$

Remembering that, in this case, b is just a constant, we can continue:

$$\frac{d}{da}\text{score} = -6b + 2b^2a$$

Knowing a and b (which of course, we do, since they're part of the output list which we create), we can find the gradient that we're looking for. This will tell us how to change the value of a to get a better score. Note that if the gradient is 0, we should keep a the same. I'll explain later how exactly we could make the changes.

Let's do it again for b. This time there are *two* brackets involving b, and it'll be the same thing for c and d. Here's how we could handle that:

$$\frac{d}{db}\text{score} = \frac{d}{db}[(3 - ab)^2 + (5 - bc)^2]$$

$$= \frac{d}{db}[(9 - 6ab + a^2b^2) + (25 - 10bc + b^2c^2)]$$

$$= (-6a + 2a^2b) + (-10c + 2c^2b)$$

It looks pretty maths-y for a computer science problem, but we've actually done the majority of the work now. Think about it - most of the values in the list are middle values like b. We've just found a formula that takes the values of our output list and gives us the gradient. We can just hard-code this into our program and it will *just work*. Here's the gradient for e, the last value in the list:

$$\frac{d}{de}\text{score} = \frac{d}{de}(6 - de)^2 = \frac{d}{de}36 - 12de + d^2e^2 = -12d + 2d^2e$$

It's pretty repetitive actually. There are a few things we haven't addressed. Firstly, there's an *extra complication* because so far we've assumed that our list is just $[3, 5, 9, 6]$, but we need a general solution for any list.

Let's generalise our input list to $[i_1, i_2, i_3, i_4]$ and our output list to $[o_1, o_2, o_3, o_4, o_5]$. We're going to do the same differentiation with these generalised lists. Let's do the gradients for o_1 and o_2:

$$\begin{aligned}\frac{d}{do_1}\text{score} &= \frac{d}{do_1}(i_1 - o_1 o_2)^2 \\ &= \frac{d}{do_1}[(i_1)^2 - 2i_1 o_1 o_2 + (o_1)^2(o_2)^2] \\ &= -2i_1 o_2 + 2(o_2)^2 o_1\end{aligned}$$

$$\begin{aligned}\frac{d}{do_2}\text{score} &= \frac{d}{do_2}[(i_1 - o_1 o_2)^2 + (i_2 - o_2 o_3)^2] \\ &= \frac{d}{do_2}\left[(i_1)^2 - 2i_1 o_1 o_2 + (o_1)^2(o_2)^2 + (i_2)^2 - 2i_2 o_2 o_3 + (o_2)^2(o_3)^2\right] \\ &= -2i_1 o_1 + 2(o_1)^2 o_2 - 2i_2 o_3 + 2(o_3)^2 o_2\end{aligned}$$

That's exactly the same as what we did before, just with more letters. It looks pretty ugly but the code I wrote to generate these maths things is even uglier (it takes ages too). But wait, we're done now - almost, anyway. I'll write one more of these horrible maths things, which is going to explain exactly how we generalise this to *any size of list*; in other words, how we find the gradient of the score with respect to any value in the output list (written as o_k, where k is an index of the list). I'm going to skip the steps in between and jump straight to the answer, because we've seen how to get there enough times now:

$$\frac{d}{do_k} = (-2i_{k-1}o_{k-1} + 2(o_{k-1})^2 o_k) + (-2i_k o_{k+1} + 2(o_{k+1})^2 o_k)$$

So that's the master equation that's going to find us any gradient we want. The only thing to be careful about is that $k-1$ and $k+1$ are valid indices of the list. So for the first value in the list, we just miss out the first bracket from that equation. For the last value in the list, we miss out the last bracket. Apart from that check, we can just translate this into Python and it will work.

So let's say we have calculated our gradients for our original output list of $[a, b, c, d, e]$. Since we get one gradient for each value in the list, we'd get a list that looks like this:

$$\left[\frac{d}{da}\text{score}, \frac{d}{db}\text{score}, ..., \frac{d}{de}\text{score}\right]$$

I'm missing out values now because this whole maths-writing thing is pretty tedious. Remember how we need to **subtract** the gradients from our list to get a better solution? Well that's only kind of correct. The gradients are way too large, and that would cause our program to be really unstable, and we'd never converge on a solution. But it's not the size of the gradient that matters - it's the sign. We just need to know whether to increase or decrease. So we can just subtract a small number with the same *sign* as the gradient. For example, if $\frac{d}{da}\text{score}$ is negative, our new value of a would be:

$$a_{new} = a_{prev} - (-0.5)$$
$$a_{new} = a_{prev} + 0.5$$

So we add 0.5 to the value if the gradient is negative, and we subtract 0.5 from the value if the gradient is positive. We don't have to use 0.5, but we can experiment to see what works best on the day. This is generally known as a hyperparameter in machine learning - more specifically, this is the *learning rate*. We can't set it too high, otherwise the algorithm would never converge on a solution. We can't set it too low, otherwise it would take ages to converge on a solution.

There's a big flaw with all of this. Can you see what it is? Using a learning rate like 0.5 won't always give us whole numbers - that means that it might tell us that the optimal value for a is 3.5 or something like that. We can't have that - we need whole numbers for our solution. The best fix for this that I could think of is just to round the number after the gradient descent process is done. ¯_(ツ)_/¯

This probably looks way too complicated to do in 60 minutes. Here's the thing though - the hard part about this is understanding how it works, not writing the code. Like I said, we already have a one-size-fits-all equation to do this for us, which we can just directly translate to python. The rest of it is really basic stuff. I've used gradient descent in one of these programming competitions before - it works, and it works quite well.

I guess it would be possible to try and code *both* the extended greedy solution and the gradient descent solution, and I would highly recommend it. For certain inputs, the greedy one will work better. For others, the gradient descent will work better. If we have two solutions that have different

strengths, our overall score will be the combination of the best results from each. Again, I've done this kind of thing before (using a simple and advanced solution) and my team won that round.

But just because I think it could work doesn't mean we need to use it. And don't be pressured to use it just because I made this document thing - this is just something you could learn the process of gradient descent from. I'd genuinely much rather use a creative idea from one of you.

Gradient descent is the driving algorithm behind many machine learning techniques - most notably, neural networks. If you can understand this, then all you need to learn is the chain rule and matrices, and at that point you'll be able to code a neural network without any external libraries (this is actually part of my Computing NEA).

P.S. apologies for any mistakes, I made this document in a bit of a hurry