Gradient Descent Solution

```
def gradient_descent_update(x, gradx, learning_rate):
    """
    Performs a gradient descent update.
    """
    x = x - learning_rate * gradx
    # Return the new value for x
    return x
```

We adjust the old `x` pushing it in the *direction* of `gradx` with the *force*`learning_rate`. Subtracting `learning_rate * gradx`. Remember the gradient is initially in the direction of **steepest ascent** so subtracting `learning_rate * gradx` from `x` turns it into **steepest descent**. You can make sure of this yourself by replacing the subtraction with an addition.

The Gradient & Backpropagation

As promised, we'll now discuss the gradient in more depth. Specifically we'll focus on the following insight:

> *In order to figure out how we should alter a parameter to minimize the cost, we must first find out what effect that parameter has on the cost.*

That makes sense. After all, we can't just blindly change parameter values and hope to get meaningful results. The gradient takes into account the effect each parameter has on the cost, so that's how we find the direction of steepest ascent.

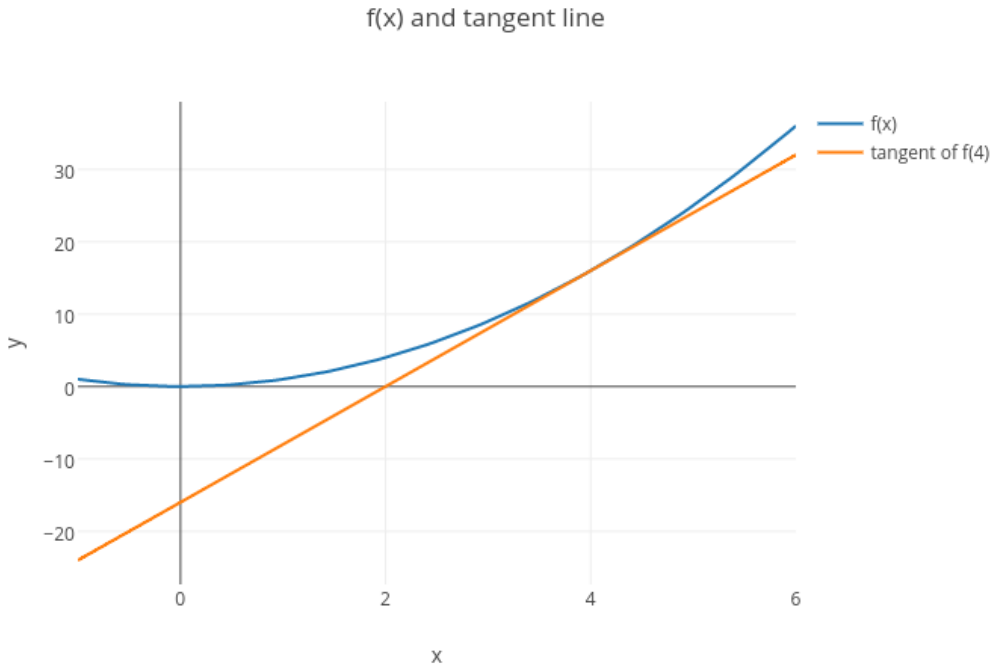
How do we determine the effect a parameter has on the cost? This technique is famously known as **backpropagation** or **reverse-mode differentiation**. Those names might sound intimidating, but behind it all, it's just a clever application of the **chain rule**. Before we get into the chain rule let's revisit plain old derivatives.

Derivatives

In calculus, the derivative tells us how something changes with respect to something else. Or, put differently, how *sensitive* something is to something else.

Let's take the function $f(x) = x^2$ as an example. In this case, the derivative of $f(x)$ is $2x$. Another way to state this is, "the derivative of $f(x)$ with respect to x is $2x$".

Using the derivative, we can say *how much* a change in x effects $f(x)$. For example, when x is 4, the derivative is 8 ($2x = 2 * 4 = 8$). This means that if x is increased or decreased by 1 unit, then $f(x)$ will increase or decrease by 8.



f(x) and the tangent line of f(x) when x = 4.

Notice that $f(4) = 16$ and $f(5) = 25$. 25 - 16 = 9, which isn't the same as 8.

But we just calculated that increasing x by 1 unit would change $f(x)$ by 8. What happened?

The answer is that the slope (or derivative) itself changes as x changes. If we calculate the derivative when x is 4.5, it's now 9, which matches the difference between $f(4) = 16$ and $f(5) = 25$.

Chain Rule

Let's return to neural networks and the original goal of figuring out what effect a parameter has on the cost.

We simply calculate the derivative of the cost with respect to each parameter in the network. The gradient is a vector of all these derivatives.

In reality, neural networks are a composition of functions, so computing the derivative of the cost w.r.t a parameter isn't quite as straightforward as calculating the derivative of a polynomial function like $f(x) = x^2$. This is where the chain rule comes into play.

I highly recommend checking out **Khan Academy's lessons on partial derivatives** and **gradients** if you need more of a refresher.

Say we have a new function $f \circ g(x) = f(g(x))$. We can calculate the derivative of $f \circ g$ w.r.t x, denoted $\frac{\partial f \circ g}{\partial x}$, by applying the chain rule.

$$\frac{\partial f \circ g}{\partial x} = \frac{\partial g}{\partial x} \frac{\partial f}{\partial g}$$

The way to think about this is:

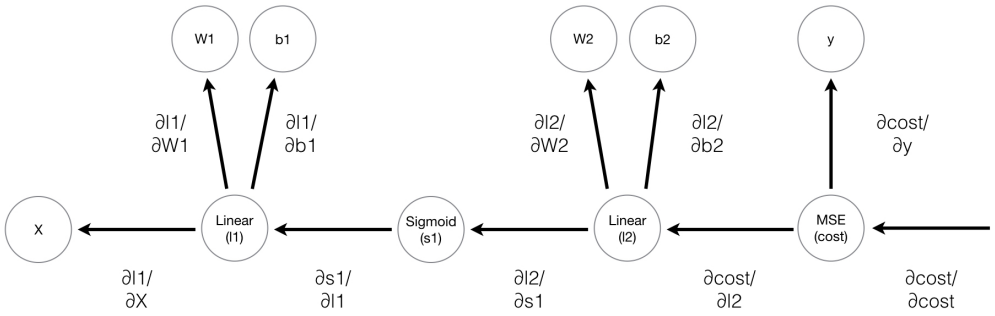
> In order to know the effect x has on f, we first need to know the effect x has on g, and then the effect g has on f.

Let's now look at a more complex example. Consider the following neural network in MiniFlow:

```
X, y = Input(), Input()
W1, b1 = Input(), Input()
W2, b2 = Input(), Input()

l1 = Linear(X, W1, b1)
s1 = Sigmoid(l1)
l2 = Linear(s1, W2, b2)
cost = MSE(l2, y)
```

This also can be written as a composition of functions `MSE(Linear(Sigmoid(Linear(X, W1, b1)), W2, b2), y)`. Our goal is to adjust the weights and biases represented by the `Input` nodes `W1, b1, W2, b2`, such that the cost is minimized.



Graph of the above neural network. The backward pass and gradients flowing through are illustrated.

In the upcoming quiz you'll implement the `backward` method for the `Sigmoid` node, so let's focus on that.

First, we unwrap the derivative of `cost` w.r.t. `l1` (the input to the `Sigmoid` node). Once again, apply the chain rule:

$$\frac{\partial cost}{\partial l1} = \frac{\partial s1}{\partial l1} \frac{\partial cost}{\partial s1}$$

We can unwrap $\frac{\partial cost}{\partial s1}$ further:

$$\frac{\partial cost}{\partial s1} = \frac{\partial l2}{\partial s1} \frac{\partial cost}{\partial l2}$$

Finally:

$$\frac{\partial cost}{\partial l1} = \frac{\partial s1}{\partial l1} \frac{\partial l2}{\partial s1} \frac{\partial cost}{\partial l2}$$

In order to calculate the derivative of `cost` w.r.t `l1` we need to figure out these 3 values:

- $\frac{\partial s1}{\partial l1}$
- $\frac{\partial l2}{\partial s1}$
- $\frac{\partial cost}{\partial l2}$

Backpropagation makes computing these values convenient.

During backpropagation, the derivatives of nodes in the graph are computed back to front. In the case of the 3 above values, $\frac{\partial cost}{\partial l2}$ would be computed first, followed by $\frac{\partial l2}{\partial s1}$ and $\frac{\partial s1}{\partial l1}$. Thus,

$\frac{\partial s1}{}$

if we compute $\frac{\partial s1}{\partial l1}$, then we can also *assume* the 2 other values have already been computed!

This insight makes backpropagation much easier to implement. When computing the backward pass for a `Node` we only need to concern ourselves with the computation of that node w.r.t its inputs.

NOTE: As a quick aside, you might be wondering why we're even computing the derivative of `cost` w.r.t `l1`. After all, `l1` is not a parameter of the neural network. While that is true, `W1` and `b1` are parameters of the network, and if you write the expression for `cost`w.r.t `W1` or `b1`, you'll notice that the derivative of `s1` w.r.t `l1` is a term in that expression, and so for the chain rule, we'll need to calculate the derivative of `cost` w.r.t `l1` at some point.

Additional Resources

- Yes you should understand backprop by Andrej Karpathy
- Vector, Matrix, and Tensor Derivatives by Erik Learned-Miller.

Alright, time for that quiz!

New Code

There have been a couple of changes to MiniFlow since we last took it for a spin:

The first being the `Node` class now has a `backward` method, as well as a new attribute `self.gradients`, which is used to store and cache gradients during the backward pass.

```
class Node(object):
    """
    Base class for nodes in the network.

    Arguments:


        `inbound_nodes`: A list of nodes with edges into this node.
    """
    def __init__(self, inbound_nodes=[]):
        """
        Node's constructor (runs when the object is instantiated). Sets
        properties that all nodes need.
```

```
        """
        # A list of nodes with edges into this node.
        self.inbound_nodes = inbound_nodes
        # The eventual value of this node. Set by running
        # the forward() method.
        self.value = None
        # A list of nodes that this node outputs to.
        self.outbound_nodes = []
        # New property! Keys are the inputs to this node and
        # their values are the partials of this node with
        # respect to that input.
        self.gradients = {}
        # Sets this node as an outbound node for all of
        # this node's inputs.
        for node in inbound_nodes:
            node.outbound_nodes.append(self)

    def forward(self):
        """
        Every node that uses this class as a base class will
        need to define its own `forward` method.
        """
        raise NotImplementedError

    def backward(self):
        """
        Every node that uses this class as a base class will
        need to define its own `backward` method.
        """
        raise NotImplementedError
```

The second change is to the helper function `forward_pass()`. That function has been replaced with `forward_and_backward()`.

```
def forward_and_backward(graph):
    """
    Performs a forward pass and a backward pass through a list of sorted no
```

```
    Arguments:

        `graph`: The result of calling `topological_sort`.
    """
    # Forward pass
    for n in graph:
        n.forward()

    # Backward pass
    # see: https://docs.python.org/2.3/whatsnew/section-slices.html
    for n in graph[::-1]:
        n.backward()
```

Setup

Here's the derivative of the $sigmoid$ function w.r.t x:

$$sigmoid(x) = 1/(1 + exp(-x))$$

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

- Complete the implementation of backpropagation for the `Sigmoid` node by finishing the `backward` method in `miniflow.py`.
- The `backward` methods for all other nodes have already been implemented. Taking a look at them might be helpful.