Backpropagation

The goals of backpropagation are straightforward: adjust each weight in the network in proportion to how much it contributes to overall error. If we iteratively reduce each weight's error, eventually we'll have a series of weights the produce good predictions.

Chain rule refresher

As seen above, foward propagation can be viewed as a long series of nested equations. If you think of feed forward this way, then backpropagation is merely an application the Chain rule to find the Derivatives of cost with respect to any variable in the nested equation. Given a forward propagation function:

$$f(x) = A(B(C(x)))$$

A, B, and C are activation functions at different layers. Using the chain rule we easily calculate the derivative of $f(x)$ with respect to x:

$$f'(x) = f'(A) \cdot A'(B) \cdot B'(C) \cdot C'(x)$$

How about the derivative with respect to B? To find the derivative with respect to B you can pretend $B(C(x))$ is a constant, replace it with a placeholder variable B, and proceed to find the derivative normally with respect to B.

$$f'(B) = f'(A) \cdot A'(B)$$

This simple technique extends to any variable within a function and allows us to precisely pinpoint the exact impact each variable has on the total output.

Applying the chain rule

Let's use the chain rule to calculate the derivative of cost with respect to any weight in the network. The chain rule will help us identify how much each weight contributes to our overall error and the direction to update each weight to reduce our error. Here are the equations we need to make a prediction and calculate total error, or cost:

Function	Formula	Derivative
Weighted input	$Z = XW$	$Z'(X) = W$ $Z'(W) = X$
ReLU activation	$R = max(0, Z)$	$R'(Z) = \begin{cases} 0 & Z < 0 \\ 1 & Z > 0 \end{cases}$
Cost function	$C = \frac{1}{2}(\hat{y} - y)^2$	$C'(\hat{y}) = (\hat{y} - y)$

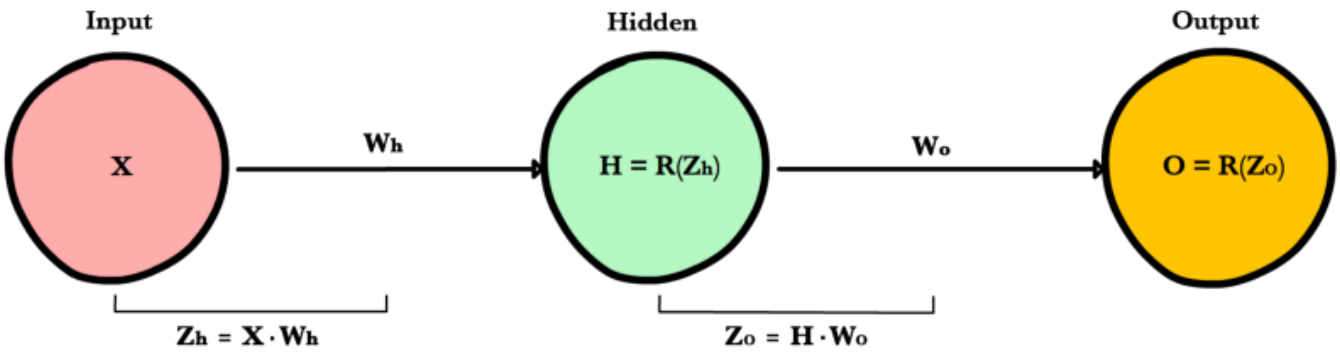
Given a network consisting of a single neuron, total cost could be calculated as:

$$Cost = C(R(Z(XW)))$$

Using the chain rule we can easily find the derivative of Cost with respect to weight W.

$$\begin{aligned} C'(W) &= C'(R) \cdot R'(Z) \cdot Z'(W) \\ &= (\hat{y} - y) \cdot R'(Z) \cdot X \end{aligned}$$

Now that we have an equation to calculate the derivative of cost with respect to any weight, let's go back to our toy neural network example above



What is the derivative of cost with respect to W_o?

$$\begin{aligned} C'(W_O) &= C'(\hat{y}) \cdot \hat{y}'(Z_O) \cdot Z_O'(W_O) \\ &= (\hat{y} - y) \cdot R'(Z_O) \cdot H \end{aligned}$$

And how about with respect to W_h? To find out we just keep going further back in our function applying the chain rule recursively until we get to the function that has the Wh term.

$$\begin{aligned} C'(W_h) &= C'(\hat{y}) \cdot O'(Z_o) \cdot Z_o'(H) \cdot H'(Z_h) \cdot Z_h'(W_h) \\ &= (\hat{y} - y) \cdot R'(Z_o) \cdot W_o \cdot R'(Z_h) \cdot X \end{aligned}$$

And just for fun, what if our network had 10 hidden layers. What is the derivative of cost for the first weight w_1?

$$\begin{aligned} C'(w_1) = \frac{dC}{d\hat{y}} \cdot \frac{d\hat{y}}{dZ_{11}} \cdot \frac{dZ_{11}}{dH_{10}} \cdot \\ \frac{dH_{10}}{dZ_{10}} \cdot \frac{dZ_{10}}{dH_9} \cdot \frac{dH_9}{dZ_9} \cdot \frac{dZ_9}{dH_8} \cdot \frac{dH_8}{dZ_8} \cdot \frac{dZ_8}{dH_7} \cdot \frac{dH_7}{dZ_7} \cdot \\ \frac{dZ_7}{dH_6} \cdot \frac{dH_6}{dZ_6} \cdot \frac{dZ_6}{dH_5} \cdot \frac{dH_5}{dZ_5} \cdot \frac{dZ_5}{dH_4} \cdot \frac{dH_4}{dZ_4} \cdot \frac{dZ_4}{dH_3} \cdot \\ \frac{dH_3}{dZ_3} \cdot \frac{dZ_3}{dH_2} \cdot \frac{dH_2}{dZ_2} \cdot \frac{dZ_2}{dH_1} \cdot \frac{dH_1}{dZ_1} \cdot \frac{dZ_1}{dW_1} \end{aligned}$$

See the pattern? The number of calculations required to compute cost derivatives increases as our network grows deeper. Notice also the redundancy in our derivative calculations. Each layer's cost derivative appends two new terms to the terms that have already been calculated by the layers above it. What if there was a way to save our work somehow and avoid these duplicate calculations?

Saving work with memoization

Memoization is a computer science term which simply means: don't recompute the same thing over and over. In memoization we store previously computed results to avoid recalculating the same function. It's handy for speeding up recursive functions of which backpropagation is one. Notice the pattern in the derivative equations below.

$$\begin{aligned} C'(W_3) &= (O - y) \cdot R'(Z_3) \cdot H_2 \\ C'(W_2) &= (O - y) \cdot R'(Z_3) \cdot W_3 \cdot R'(Z_2) \cdot H_1 \\ C'(W_1) &= (O - y) \cdot R'(Z_3) \cdot W_3 \cdot R'(Z_2) \cdot W_2 \cdot R'(Z_1) \cdot H_0 \\ C'(W_0) &= (O - y) \cdot R'(Z_3) \cdot W_3 \cdot R'(Z_2) \cdot W_2 \cdot R'(Z_1) \cdot W_1 \cdot R'(Z_0) \cdot X \end{aligned}$$

Each of these layers is recomputing the same derivatives! Instead of writing out long derivative equations for every weight, we can use memoization to save our work as we backprop error through the network. To do this, we define 3 equations (below), which together encapsulate all

the calculations needed for backpropagation. The math is the same, but the equations provide a nice shorthand we can use to track which calculations we've already performed and save our work as we move backwards through the network.

Function	Formula	Derivative
Weighted input	$Z = XW$	$Z'(X) = W$ $Z'(W) = X$
ReLU activation	$R = max(0, Z)$	$R'(Z) = \begin{cases} 0 & Z < 0 \\ 1 & Z > 0 \end{cases}$
Cost function	$C = \frac{1}{2}(\hat{y} - y)^2$	$C'(\hat{y}) = (\hat{y} - y)$

We first calculate the output layer error and pass the result to the hidden layer before it. After calculating the hidden layer error, we pass its error value back to the previous hidden layer before it. And so on and so forth. As we move back through the network we apply the 3rd formula at every layer to calculate the derivative of cost with respect that layer's weights. This resulting derivative tells us in which direction to adjust our weights to reduce overall cost.

Note

The term *layer error* refers to the derivative of cost with respect to a layer's *input*. It answers the question: how does the cost function output change when the input to that layer changes?

Output layer error

To calculate output layer error we need to find the derivative of cost with respect to the output layer input, Z_o. It answers the question—how are the final layer's weights impacting overall error in the network? The derivative is then:

$$C'(Z_o) = (\hat{y} - y) \cdot R'(Z_o)$$

To simplify notation, ml practitioners typically replace the $(\hat{y} - y) * R'(Zo)$ sequence with the term E_o. So our formula for output layer error equals:

$$E_o = (\hat{y} - y) \cdot R'(Z_o)$$

Hidden layer error

To calculate hidden layer error we need to find the derivative of cost with respect to the hidden layer input, Zh.

$$C'(Z_h) = (\hat{y} - y) \cdot R'(Z_o) \cdot W_o \cdot R'(Z_h)$$

Next we can swap in the E_o term above to avoid duplication and create a new simplified equation for Hidden layer error:

$$E_h = E_o \cdot W_o \cdot R'(Z_h)$$

This formula is at the core of backpropagation. We calculate the current layer's error, and pass the weighted error back to the previous layer, continuing the process until we arrive at our first hidden layer. Along the way we update the weights using the derivative of cost with respect to each weight.

Derivative of cost with respect to any weight

Let's return to our formula for the derivative of cost with respect to the output layer weight W_o.

$$C'(W_O) = (\hat{y} - y) \cdot R'(Z_O) \cdot H$$

We know we can replace the first part with our equation for output layer error E_h. H represents the hidden layer activation.

$$C'(W_o) = E_o \cdot H$$

So to find the derivative of cost with respect to any weight in our network, we simply multiply the corresponding layer's error times its input (the previous layer's output).

$$C'(w) = CurrentLayerError \cdot CurrentLayerInput$$

Note

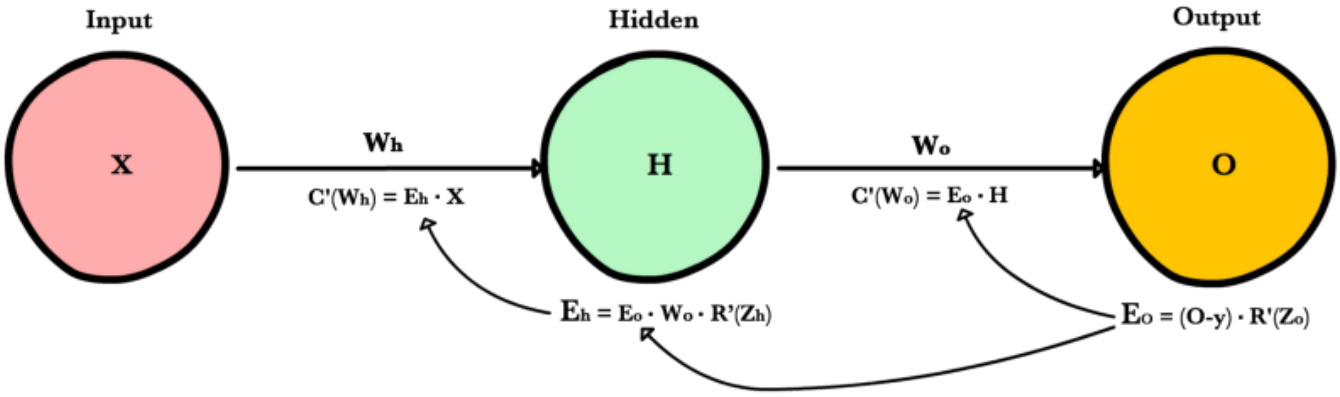
Input refers to the activation from the previous layer, not the weighted input, Z.

Summary

Here are the final 3 equations that together form the foundation of backpropagation.

Output Layer Error	$E_o = (O - y) \cdot R'(Z_o)$
Hidden Layer Error	$E_h = E_o \cdot W_o \cdot R'(Z_h)$
Cost-Weights Deriv	$LayerError \cdot LayerInput$

Here is the process visualized using our toy neural network example above.



Code example

```
def relu_prime(z):
    if z > 0:
        return 1
    return 0

def cost(yHat, y):
    return 0.5 * (yHat - y)**2

def cost_prime(yHat, y):
    return yHat - y

def backprop(x, y, Wh, Wo, lr):
    yHat = feed_forward(x, Wh, Wo)

    # Layer Error
    Eo = (yHat - y) * relu_prime(Zo)
    Eh = Eo * Wo * relu_prime(Zh)

    # Cost derivative for weights
    dWo = Eo * H
    dWh = Eh * x

    # Update weights
    Wh -= lr * dWh
    Wo -= lr * dWo
```

References

[1] Example