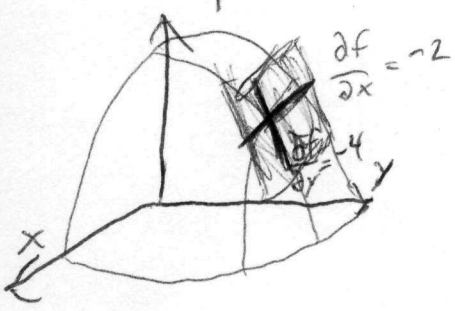
To use an example from the book, if $z = 14 - x^2 - y^2$ and we want the tangent plane at $(1, 2, 9)$,

$\frac{\partial f}{\partial x} = -2x$ $\quad$ $\frac{\partial f}{\partial y} = -2y$

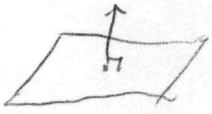
↳ -2 at point $\quad$ ↳ -4 at point

$\rightarrow z - 9 = -2(x-1) - 4(y-2)$

$\Rightarrow 2x + 4y + z = 19$



We might also be interested in a vector that is orthogonal to this plane — the "normal vector."



Its equation is $\begin{bmatrix} \left(\frac{\partial f}{\partial x}\right)_0 \\ \left(\frac{\partial f}{\partial y}\right)_0 \\ -1 \end{bmatrix}$, which we can check against the equation of the plane to see that the dot product with any vector from (x_0, y_0, z_0) to another point in the plane will be 0:

$$\begin{bmatrix} \Delta x \\ \Delta y \\ \left(\frac{\partial f}{\partial x}\right)_0 \Delta x + \left(\frac{\partial f}{\partial y}\right)_0 \Delta y \end{bmatrix} \cdot \begin{bmatrix} \left(\frac{\partial f}{\partial x}\right)_0 \\ \left(\frac{\partial f}{\partial y}\right)_0 \\ -1 \end{bmatrix} = \begin{array}{l} \left(\frac{\partial f}{\partial x}\right)_0 \Delta x - \left(\frac{\partial f}{\partial x}\right)_0 \Delta x \\ + \left(\frac{\partial f}{\partial y}\right)_0 \Delta y - \left(\frac{\partial f}{\partial y}\right)_0 \Delta y \end{array} = 0$$

These equations can be extended naturally to higher dimensions, too, defining hyperplanes and their normals.

Directional derivatives

$\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ give us two very specific slopes — one along the x-axis, and one along the y-axis. But there's no particular reason we should only be able to take derivatives in these two directions; we should be able to find the slope of a function in a direction like 45° between these two directions.

Given $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$, we can actually find the derivative in any direction in the plane.

The derivative in the direction of a unit vector $\vec{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$ is $D_u f = \frac{\partial f}{\partial x} u_1 + \frac{\partial f}{\partial y} u_2$. That is, we scale each component by the derivative in that direction. So, for example, if $f = xy$, then $\frac{\partial f}{\partial x} = y$, $\frac{\partial f}{\partial y} = x$, and a derivative in the direction $\begin{bmatrix} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{bmatrix}$ (a 45° angle) is $y\left(\frac{\sqrt{2}}{2}\right) + x\left(\frac{\sqrt{2}}{2}\right)$. (The derivative in the direction of $\vec{v}$ still depends on the exact coordinates; so the slope in the direction of $\begin{bmatrix} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{bmatrix}$ at $(3,1)$ would be $\frac{3\sqrt{2}}{2} + \frac{\sqrt{2}}{2} = \frac{4\sqrt{2}}{2} = 2\sqrt{2}$.) Notice that the directional derivative is a scalar; we already set the direction when we asked for it.

To see why this should be the formula, consider that a natural generalization of the derivative to multiple dimensions is $D_u f = \lim_{\Delta s \to 0} \frac{\Delta f}{\Delta s}$ for a "step" Δs of size in the direction u. If $\Delta f = \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y = \frac{\partial f}{\partial x}u_1 \Delta s + \frac{\partial f}{\partial y}u_2 \Delta s$, then we get our formula with $\frac{\Delta f}{\Delta s}$.

The directional derivative is analogous for more dimensions — take a unit vector in the desired dimensions and multiply its components by the appropriate partial derivatives. Actually though, we're mostly setting up the idea of the "gradient" here — a vector in the dir. of the biggest directional derivative.

The gradient

When doing machine learning, we often want to "turn the knobs" of some function in a way that will reduce error the most, or improve performance the most. To do this, we need a gradient.

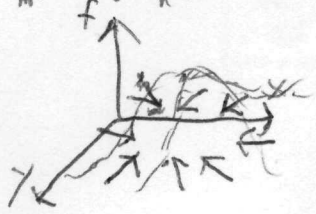
The gradient ∇f is a vector in the direction $\vec{u}$ that maximizes the directional derivative $D_u f$. In a function with two inputs, $f(x, y)$, the formula is

$$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$$

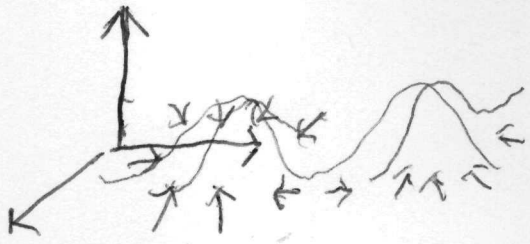
That gives the direction, and the length of the gradient vector is the value of the derivative (so, $D_{\nabla f} = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$)

The most salient parts about the gradient are

- It points in the direction of greatest change (specifically, increase - "uphill")
- It has a larger or smaller magnitude depending on the magnitude of the slope in that direction.
- It doesn't actually have a component in the direction of f. In the case of $f(x,y)$, it lives in the xy plane, pointing in the direction to tune x and y in order to increase f.



Gradients live in the space of inputs, pointing toward the inputs that would increase f.



But they're still only local info about slope - they don't "know" the right direction to go globally from any particular point.

How do we know our vector for the gradient is steepest? We can $D_u f = \frac{\partial f}{\partial x} u_1 + \frac{\partial f}{\partial y} u_2$ is the dot product of the gradient and the unit vector: $D_u f = \nabla f \cdot \vec{u}$. We can then ask, what value for $\vec{u}$ maximizes this dot product? Using the alternate formula for the dot product, $\nabla f \cdot \vec{u} = |\nabla f||\vec{u}| \cos\theta$, we can see the value is maximized when $\theta = 0$ and $\vec{u}$ is in the direction of ∇f.

Like the directional derivative, the gradient generalizes to more dimensions, $x_1, x_2, \ldots, x_n$.

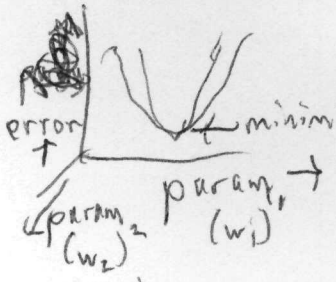
$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

The logic for its being the direction with the biggest directional derivative remains the same; $\nabla f \cdot \vec{u}$ is maximized when the angle between them is 0.

Gradient descent

Let's recall the perceptron learning rule, $w_i' = w_i + \alpha(\text{err})x_i$. This was a nudge of the perceptron function in a direction that would reduce error.

This rule can actually be derived from taking partial derivatives with respect to the error. We can think of our function for the perceptron as generating some amount of error, based on the parameters,



Just like in basic calculus, we can't take derivatives of non-continuous functions — so that ⌐ step function is a non-starter for calculus. But, if our perceptron says TRUE, then we could just consider the total amount we're above 0 to be error.

$$Error = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b$$

(if we said >0 and wanted <0)

*We're simplifying a little because we haven't covered the (multivar) chain rule yet.

Now we can find the gradient of this function, the direction in parameter space that would maximize the error. That's the vector consisting of derivatives

$$\nabla Error = \left[\frac{\partial Error}{\partial w_1}, \frac{\partial Error}{\partial w_2}, \cdots, \frac{\partial Error}{\partial w_n}, \frac{\partial Error}{\partial b} \right]$$

Since this is a linear function each of these just zeroes out everything but its own coefficient.

$$\nabla Error = [x_1, x_2, \cdots, x_n, 1]$$

This gives us a vector pointing in the most "uphill" direction of the error surface. To minimize error, we'd want to move in the opposite direction.

$$\text{Dir to adjust} = [-x_1, -x_2, -x_3, \cdots, -x_n, -1]$$

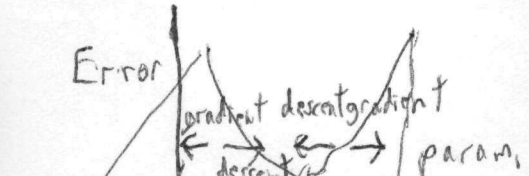
Of course, we have the same issue with this method that we noted while talking about perceptrons: this vector could overshoot where we really want to be for the parameters. So we scale by a learning rate α...

... and we get the original perceptron rule, $w_i = w_i - \alpha x_i$ if the error was a false positive.

(We'd come to a similar conclusion for a false negative, introducing the +1/0/-1 "Err" term used the first time.)

Thus perceptron learning is a specific example of a more general strategy called "gradient descent":

1. Describe the error on a particular example as a function of the learner's parameters.
2. Find the gradient of that function.
3. Adjust the parameters in the opposite direction.
4. Repeat on different examples until the error is no longer changing.



We'll see that multilayer neural networks are really a kind of gradient descent with a lot of parameters and a complex function.

We can see each component of the gradient as assigning "blame" to that parameter. If $\frac{\partial \text{Error}}{\partial w_i}$ is big, increasing w_i increases the error a lot. If $\frac{\partial \text{Error}}{\partial w_i}$ is very negative, increasing the parameter is a good way to decrease error. If $\frac{\partial \text{Error}}{\partial w_i}$ is 0, wiggling this parameter doesn't do much to this classification.

Gradient descent methods come in a few varieties. "Stochastic gradient descent" is probably more popular than "batch" gradient descent. The former picks a data point at random to learn from at each time, while the latter sums error across many points before adjusting. It's also possible you don't really know what function you're manipulating - "empirical gradient descent" just plays with the parameters to estimate the local gradient.

So, one way to bump the expressiveness of our perceptron would be to pick a different parameterized function, like

$$w_1 x_1^2 + w_2 x_1 + w_3 x_2^2 + w_4 x_2 + \ldots + b$$

and then do gradient descent with that instead.

Gradient is

$$\nabla Err = [2x_1, x_1, 2x_2, x_2, \ldots, 1]$$

and so our weights would be adjusted

$$w_i' -= \alpha 2x_i \text{ or } w_i -= \alpha x_i$$ depending on whether i is odd (x^2 coef) or even (x coef)

(α: scaling the gradient)

or whatever, adjusting opposite the gradient. In practice, there are just a few kinds of functions that are "one-size-fits-all" that people use, including the multilayer neural networks we'll get to soon.

(end wk 6)

Stochastic gradient descent is well-known enough that you can readily find implementations (in, for example, python's scikit-learn). SGDRegressor does least-squares-like linear regression while SGDClassifier does perceptron-like linear classification. But there are two parameters to the constructor that go a little beyond what those methods do.

For one, you can change the loss — the function that we are actually doing gradient descent on doesn't have to be the squared error. The loss is what we are trying to minimize, and it could be all-or-nothing depending on whether we're right, or ~~[illegible]~~ increase more gradually than the square, or it could be based on something else entirely. (This usually doesn't matter all that much, though there are an intimidating wealth of options.)

Another thing we could adjust is the penalty for having nonzero parameters, we should prefer zeros to small parameters since features that shouldn't do anything are common. These penalties often take the form of a norm on the vector of parameters: L1 is the sum of their absolute values and L2 is the sum of squares. We can do gradient descent on Loss + $\alpha \|\vec{w}\|$ instead of just the loss, where α is some small

regularization

weight, [We really do care much more about error than the sum of weights, so α should be small,)
The penalty is called a regularization penalty and "L1 regularization" is particularly good at driving weights to 0.

Introducing a function on top of the error, even if it's squaring it, embeds a function within a function — so we'll cover next how to apply the chain rule in a multivariable setting.

wk 7 41

The Chain Rule

The chain rule is particularly of interest to us when we want to do gradient descent on complicated functions that pass the input through a few steps of processing. Neural networks, as we'll see, are essentially a way to do gradient descent on a fairly complicated input→output function. The chain rule helps take error at the output level and attribute it to contributing factors earlier in a computation.

Recall from calculus that the basic chain rule is if $h(x) = f(g(x))$, then $h'(x) = f'(g(x))g'(x)$. $\frac{df}{dg}\frac{dg}{dx}$

For example, if $h(x) = \sin x^2$, this is like $f(x) = \sin x$ and $g(x) = x^2$, and since $f'(x) = \cos x$, then $h'(x) = \cos x^2 \cdot 2x = 2x \cos x^2$.

There are two main patterns for extending this to the multivariable setting. The first (probably more common in CS) is that we want to apply the chain rule to a function of the form $f(g(x,y))$, finding either $\frac{\partial f}{\partial x}$ or $\frac{\partial f}{\partial y}$. In this case, the rule looks as you might expect, which is to say do as we did with other multivariable derivatives and treat other variables as constants.

$$\frac{\partial f}{\partial x} = \frac{df}{dg}\cdot\frac{\partial g}{\partial x}$$

So, if $f(x,y) = \sin(2x+y)$, $\frac{\partial f}{\partial x} = (\cos(2x+y)) \cdot 2$ and $\frac{\partial f}{\partial y} = (\cos(2x+y)) \cdot 1$,

A second possibility is that we're differentiating a function with respect to an underlying variable that drives two intermediate variables, $f(x(t), y(t))$ where we want df/dt. In this case, the chain rule is applied to both intermediate variables and summed: $\frac{df}{dt} = \frac{\partial f}{\partial x} \cdot \frac{dx}{dt} + \frac{\partial f}{\partial y} \cdot \frac{dy}{dt}$.

For example, if $f(x,y) = (x^2 + y^2)$ and $x = 2t$, $y = 3t$, we could compute $\frac{df}{dt}$ as $\frac{\partial f}{\partial x} \cdot \frac{dx}{dt} + \frac{\partial f}{\partial y} \cdot \frac{dy}{dt}$

$= 2x \cdot 2 + 2y \cdot 3$

$= 4x + 6y$

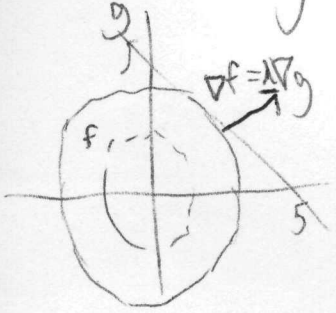
$= 8t + 18t = 26t$

Which is what we'd have gotten if we'd just substituted first ($(2t)^2 + (3t)^2 = 13t^2 \Rightarrow 26t$) but shows that when an underlying variable has multiple paths to affecting the outcome, we just need to sum those effects.

Lagrange multipliers & optimization

There's an additional trick that can be done in optimization problems with calculus beyond just looking for minima or maxima of the function to be optimized (i.e. places where the gradient is $\vec{0}$). We may want to find minima and maxima of a function subject to some additional constraint, like our variables needing to sum to something, or sitting on a particular circle. For example, we may want the closest point to the origin (minimize x^2+y^2) subject to being on the line $y=5-x$ (we could solve this with a projection but it's easy to visualize this one; the equation need not be linear).

The main trick to this is the following insight: if we really have the best value of f, we should not be able to improve by walking along the line $x+y=5$. This should be a "level curve" for f at our "k" solution. And that means it's perpendicular to the gradient of f. Further, the gradient of $g=x+y$ is also perpendicular to the curve of g — if we loosened its belt it would expand in a direction perpendicular to its curve.



(by increasing k from 5)

So for these "min/max f subject to $g=k$" sorts of problems, we have the method of Lagrange multipliers, wherein we take advantage of the fact $\nabla f = \lambda \nabla g$ at the solution point (or else walking along g could improve f). λ is the typical symbol for the Lagrange multiplier, which exists because we only know the two gradients point in the same direction.

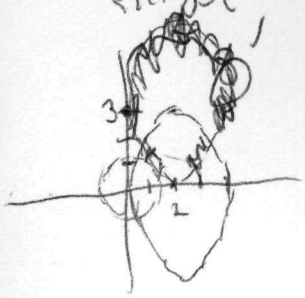
With two variables x, y, this typically gives us 3 constraints: $f_x = \lambda g_x$, $f_y = \lambda g_y$, and $g(x,y) = k$.

For our particular example, we have

$$2x = \lambda \qquad 2y = \lambda \qquad x + y = 5$$

Now we have 3 equations and 3 unknowns, and can solve for $\lambda = 5$ $x = 5/2$ $y = 5/2$. The solution here, unsurprisingly, is $(2.5, 2.5)$.

What if we constrain the solution to be on an ellipse, $\frac{(x-2)^2}{4} + \frac{(y-2)^2}{9} = 1$?



$f_x = \lambda g_x$: $2x = \lambda \frac{(x-2)}{2} \Rightarrow 4x = \lambda x - 2\lambda \Rightarrow x = \frac{-2\lambda}{4-\lambda}$

$f_y = \lambda g_y$: $2y = \lambda \frac{2(y-2)}{9}$

From the second equation, $\lambda = 0$ or 9, making x 0 or $\frac{-18}{5} = 3\frac{3}{5}$ and y 0 or $\frac{9}{5}$ (if we solve using the ellipse equation). We can infer $(0,0)$ is a minimum and $(3\frac{3}{5}, 1\frac{4}{5})$ is a maximum (last stop for expanding circle to touch ellipse).

$\frac{16}{25} + \frac{y^2}{9} = 1$

$\frac{y^2}{9} = \frac{9}{25}$

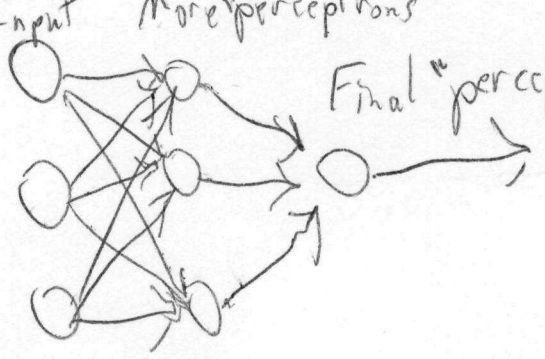
We set up that equation to be easy to solve with the $\lambda = 0$ case, in general, there isn't necessarily a set method for solving the system of equations unless it's linear. (Though we can numerically approximate things)

Optimizing subject to lots of inequality constraints is called "linear programming" (if the constraints are linear) and generally uses some more advanced algorithms. But you can sometimes see Lagrange multipliers crop up in CS applications when something is being minimized subject to some known constraints. We only touch on it here so you'll recognize the method when you see it.

Neural Networks

It's been a while since we covered the perceptron. But with the chain rule in hand, we can now begin to explain how neural networks work.

The main restriction of the perceptron was that it wasn't expressive enough, it could just learn linear functions. But we could start to learn arbitrary functions if we can chain perceptrons together. Recall that AND, OR, and NOT are linear functions individually. Combined in the right way, they can make arbitrary functions of digital input. Similarly, networks of perceptrons can represent and compute arbitrary functions.



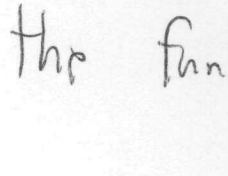
We can hook up inputs to multiple perceptrons, then have those outputs feed more perceptrons, and so on until a final perceptron makes sense of this. At this point, we wouldn't refer to these nodes as perceptrons at all - they're neurons in a neural network.

The main problem that stumped people for a bit in the beginning was, how do you train such a thing? The answer is, by adjusting each weight w_i according to $-\frac{dError}{dw_i}$, moving in the opposite direction of the derivative of the error. For a single perceptron of the kind we've seen before, this means adjusting in the direction $-\frac{d}{dw_i}(x_1w_1 + x_2w_2 + \cdots + x_nw_n)$

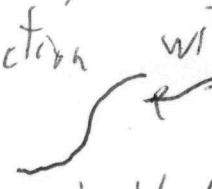
$= -x_i$. But, that's when our function to differentiate was simple, easy to take a derivative of. How do we take a derivative in a complex network?

- By using the chain rule. Each perceptron's output in the network is a function of the output of the neurons before it, which in turn are functions of

their inputs. These are composed functions, so the error is a composed function as well. When finding $\frac{dError}{dw_i}$ for a weight that is early in the network, we'll need to think of the error as the result of composing our summation and threshold functions.

This leads to the other insight we need to perform learning with neural networks. Our step function  is not differentiable, and worse, the slope on either side doesn't give a clue as to where we are in the function. 

This was a major problem that had to be solved in the early days.

The solution was to replace the function with a differentiable function, the sigmoid: $\frac{1}{1+e^{-x}}$ 

looks like a step but smoother; it's differentiable! This function allowed the chain rule to work through multiple levels.

For deep neural networks lately, another activation function is commonly used:  The rectifier is 0 before the threshold and a pass-through for input after that. It may not be differentiable everywhere, but we can fudge the discontinuity to the left or right if needed.

Backpropagation

Now we're ready to describe how such a network would adjust itself to accommodate error. If we think of its error as a function of both inputs and weights, we want to adjust each weight w_i in a direction & magnitude dictated by $\frac{\partial \text{Error}}{\partial w_i}$.

Because these terms depend on the error calculated in later layers of the network, the process of adjustment is called "backpropagation" as error terms and adjustments propagate back through the network.

We start at the very end of the network. If $f(x)$ is our activation function (like a sigmoid), then the error is a function of $f(x_1 w_1 + \dots + x_n w_n)$, where the x_i are the signals from the previous layer, that the activation function is differentiable, and the chain rule to determine $\frac{\partial \text{Error}}{\partial w_i}$:

$$\frac{\partial}{\partial w_i} f(x_1 w_1 + \dots + x_n w_n)$$

that is $f'(\text{in}) \cdot x_i$ where f' is the derivative of the activation function at the point corresponding to the particular weight's input. Thus the rule is $w_i = w_i$

We'll also consider ourselves to be minimizing the squared error, just like in least squares; this penalizes big errors more than many small ones, and also conveniently will result in the network adjusting more or less to bigger or smaller errors.

First, suppose we find the gradient for a single perceptron, only with a differentiable activation function g. Let the "loss" be $(y - p_w(x))^2$, where y is the true value and $p_w(x)$ is the perceptron's output, $p_w(x) = g(w \cdot x)$; we'll call it "loss" to make clear it's not just a simple difference $|y - p_w(x)|$. Suppose we want to find a weight's blame for some loss, this is

$$\frac{\partial \text{Loss}}{\partial w_i} = \frac{\partial}{\partial w_i}(y - p_w(x))^2$$

$$= 2(y - p_w(x)) \frac{\partial}{\partial w_i}(y - p_w(x)) \quad \text{(chain rule)}$$

$$= 2(y - p_w(x)) \cdot (-g'(w \cdot x)) \frac{\partial}{\partial w_i} w \cdot x \quad \text{(chain rule)}$$

$$= -2(y - p_w(x)) \cdot g'(w \cdot x) x_i \quad \text{(all other terms of } w \cdot x \text{ have } \frac{\partial}{\partial w_i} = 0\text{)}$$

$$= \alpha \, \text{Err} \cdot g'(in) x_i$$

This looks identical to the perceptron rule we had before, only now that the activation function is differentiable, we multiply by its derivative too. The -2 term left over from differentiation can just be absorbed into the learning rate, as long as we shift in the right direction.

For backprop through multiple layers, the layer closest to the output looks like that. But the earlier layers' error is best computed from some terms left over from adjusting the last layer.

Let w_{jk} be a weight connecting the second-to-last layer to the last layer, and let w_{ij} be a weight connecting the third-to-last to the second-to-last. (If we have a single-hidden-layer network, these are the only kinds of weights.) Also, let $\Delta_k = Err_k \, g'(in_k)$, the error at output layer k multiplied by the derivative of the activation function — having this term around will help as see similarity to the single-layer case. (And let a_k be the output of a neuron at layer k.)

$$\frac{\partial Loss_k}{\partial w_{ij}} = -2(y - a_k)\frac{\partial a_k}{\partial w_{ij}} = -2(y-a_k)\frac{\partial g(in_k)}{\partial w_{ij}}$$ } Chain rule

$$= -2(y-a_k)\, g'(in_k)\frac{\partial in_k}{\partial w_{ij}} = -2\Delta_k \frac{\partial}{\partial w_{ij}}\left(\sum_j w_{jk} a_j\right)$$

(From Russell & Norvig 3rd ed p. 735)

$$= -2\Delta_k w_{jk}\frac{\partial a_j}{\partial w_{ij}}$$ (Call other deriv terms are zero)

$$= -2\Delta_k w_{jk}\frac{\partial g(in_j)}{\partial w_{ij}} = -2\Delta_k w_{jk}\, g'(in_j)\frac{\partial in_j}{\partial w_{ij}}$$ More chain rule.

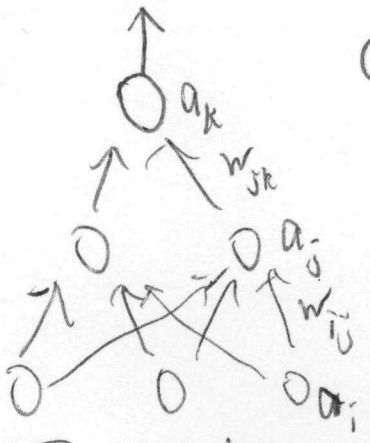
$$= -2\Delta_k w_{jk}\, g'(in_j)\frac{\partial}{\partial w_{ij}}\left(\sum_i w_{ij} a_i\right)$$

$$= -2\Delta_k w_{jk}\, g'(in_j)\, a_i$$ Which makes the total update ... over all output nodes k ...

$\propto a_i \Delta_j$ if we define $\Delta_j = g'(in_j)\sum_k w_{jk}\Delta_k$

leading to a nice-looking update rule $w_{ij} \mathrel{+}= \alpha a_i \Delta_j$.

So, backprop consists of calculating the adjusted weights for each previous layer, using error terms calculated for the layer above.



① Compute activations a_j of layer after input layer: $a_j = g(\sum_i w_{ij} x_i)$ for each j and repeat for each layer after that:

$$a_k = g(\sum_j w_{jk} a_j)$$

② Get the error/loss by comparing to the desired output.

$$Err_k = [\text{true value}]_k - a_k$$

③ Find the adjustments to be made in the last layer's input weights, $w_{jk} \mathrel{+}= \alpha Err_k \, g'(in_k) a_j$ or $w_{jk} \mathrel{+}= \alpha \Delta_k a_j$

④ Calculate new error terms to propagate back

$$\Delta_j = g'(in_j) \sum_k w_{jk} \Delta_k$$

⑤ Update the previous level's weights

$$w_{ij} \mathrel{+}= \alpha \Delta_j a_i$$

⑥ Repeat for any previous layers if this is a deep network.

Neural networks can be used for:

- Classification. For each class, have an output that is 1 if that is the class and 0 if not. In testing, use the output that is most active (closest to 1).
- Learning "evaluation functions" - as part of another learning algorithm; these can learn to output evaluations of game positions, ranging from +1 (really good) to -1 (really bad). These can help a game-playing AI make good moves.
- Speech recognition & understanding. Alexa uses a trained neural network to choose which task is being selected, and take the right "arguments" to the command.
- Machine translation. Google uses a neural network to perform translations in Google translate - it's trained on "aligned corpora" of text and learns to produce the right words and phrases in the target language.

There are many varieties of neural networks that have structures more or less suited to their tasks - but they generally use the backprop we described here.

- end week 7