

17 Neural Networks

NEURAL NETWORKS

Can do both classification & regression.

[They tie together several ideas from the course: perceptrons, logistic regression, ensembles of learners, and stochastic gradient descent. They also tie in the idea of lifting sample points to a higher-dimensional feature space, but with a new twist: neural nets can learn features themselves.]

[I want to begin by reminding you of the story I told you at the beginning of the semester, about Frank Rosenblatt's invention of perceptrons in 1957. Remember that he held a press conference where he predicted that perceptrons would be "the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence."]

[Perceptron research continued until something monumental happened in 1969. Marvin Minsky, one of the founding fathers of AI, and Seymour Papert published a book called "Perceptrons." Sounds good, right? Well, part of the book was devoted to things perceptrons can't do. And one of those things is XOR.]

		x_1	
	XOR	0	1
x_2	0	0	1
	1	1	0

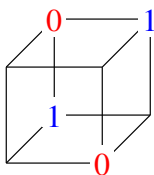
[Think of the four outputs here as sample points in two-dimensional space. Two of them are in class 1, and two of them are in class 0. We want to find a linear classifier that separates the 1's from the 0's. Can we do it? No.]

[So Minsky and Papert were basically saying, "Frank. You're telling us this machine is going to be conscious of its own existence but it can't do XOR?"]

[The book had a devastating effect on the field. After its publication, almost no research was done on neural net-like ideas for a decade, a time we now call the "AI Winter." Shortly after the book was published, Frank Rosenblatt died.]

[One thing I don't understand is why the book was so fatal when there are several almost obvious ways to get around the XOR problem. Here's the easiest.]

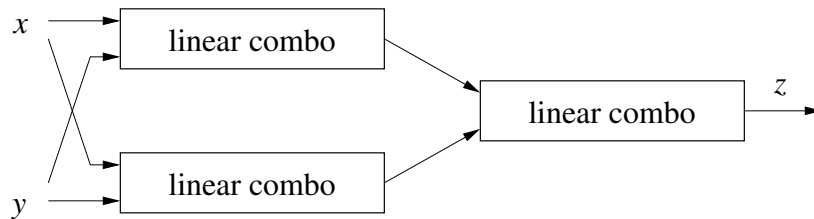
If you add one new quadratic feature, $x_1 x_2$, XOR is linearly separable in 3D.



[Draw this by hand. [xorcube.pdf](#)]

[Now we can find a plane that cuts through the cube obliquely and separates the 0's from the 1's.]

[However, there's an even more powerful way to do XOR. The idea is to design linear classifiers whose output is the input to other linear classifiers. That way, you should be able to emulate arbitrary logical circuits. Suppose I put together some linear decision functions like this.]



[Draw this by hand. [lincombo.pdf](#)]

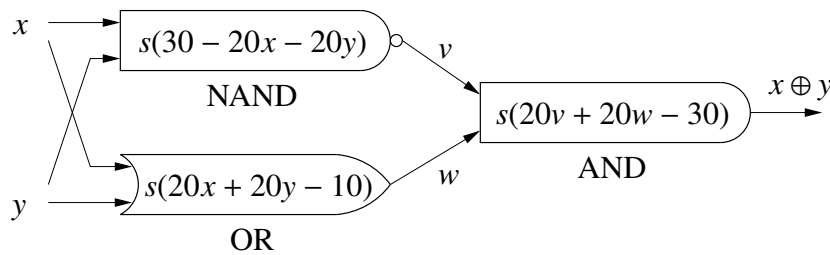
[If I interpret the output as 1 if z is positive or 0 if z is negative, can I do XOR with this?]

A linear combo of a linear combo is a linear combo ... only works for linearly separable points.

[We need one more idea to make neural nets. We need to add some sort of nonlinearity between the linear combinations. Let's call these boxes that compute linear combinations "neurons." If a neuron runs the linear combination it computes through some nonlinear function before sending it on to other neurons, then the neurons can act somewhat like logic gates. The nonlinearity could be as simple as clamping the output so it can't go below zero. And that's what people usually use in practice these days.]

[However, the traditional choice has been to use the logistic function. The logistic function can't go below zero or above one, which is nice because it can't ever get huge and oversaturate the other neurons it's sending information to. The logistic function is also smooth, which means it has well-defined gradients and Hessians we can use for optimization.]

[With logistic functions between the linear combinations, here's a two-level perceptron that computes the XOR function.]



[Draw this by hand. [xorgates.pdf](#)]

[A natural question is: can an algorithm learn a function like this?]

Network with 1 Hidden Layer

Input layer: $x_1, \dots, x_d; x_{d+1} = 1$

Hidden units: $h_1, \dots, h_m; h_{m+1} = 1$

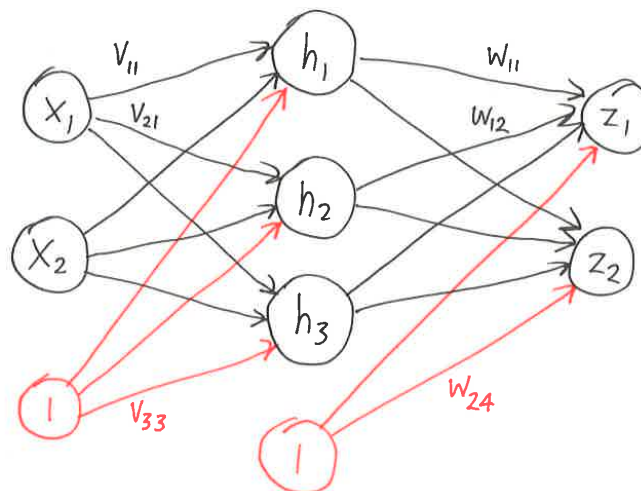
Output layer: z_1, \dots, z_k

Layer 1 weights: $m \times (d + 1)$ matrix V V_i^\top is row i

Layer 2 weights: $k \times (m + 1)$ matrix W W_i^\top is row i

Recall [logistic function] $s(\gamma) = \frac{1}{1+e^{-\gamma}}$. Other nonlinear fns can be used.

For vector v , $s(v) = \begin{bmatrix} s(v_1) \\ s(v_2) \\ \vdots \end{bmatrix}$, $s_1(v) = \begin{bmatrix} s(v_1) \\ s(v_2) \\ \vdots \\ 1 \end{bmatrix}$ [We apply s to a vector component-wise.]



[Draw this by hand. [1hiddenlayer.pdf](#)]

$$h = s_1(Vx) \quad \dots \text{ that is, } h_i = s(V_i^\top x)$$

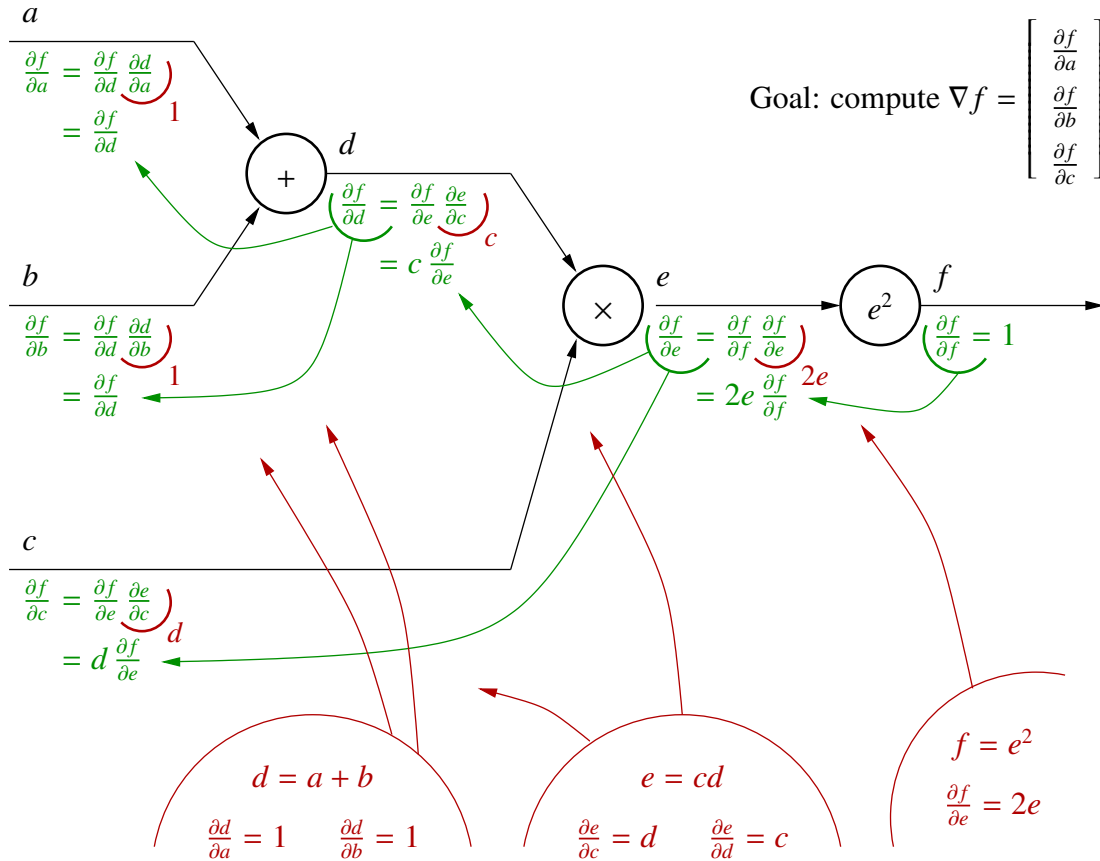
$$z = s(W h) = s(W s_1(Vx))$$

[Neural networks often have more than one output. This allows us to build multiple classifiers that share hidden units. One of the interesting advantages of neural nets is that if you train multiple classifiers simultaneously, sometimes some of them come out better because they can take advantage of particularly useful hidden units that first emerged to support one of the other classifiers.]

[We can add more hidden layers, and for image recognition tasks it's common to have 8 to 200 hidden layers. There are many variations you can experiment with—for instance, you can have connections that go forward more than one layer.]

Computing Gradients for Arithmetic Expressions

[Let's see what it takes to compute the gradient of an arithmetic expression. It turns into repeated applications of the chain rule from calculus.]



Each value z gives partial derivative of the form

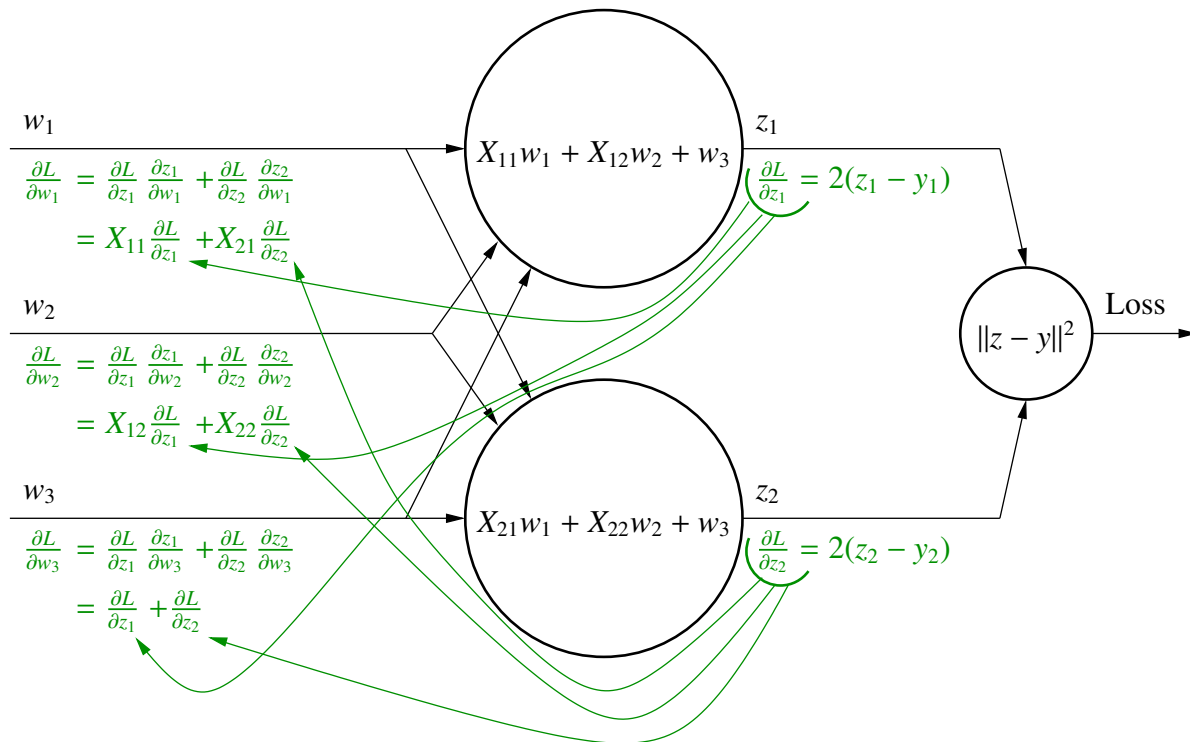
$$\frac{\partial f}{\partial z} = \left(\frac{\partial f}{\partial n} \frac{\partial n}{\partial z} \right)$$

where z is an input to n .

computed during forward pass
 computed during backward pass after forward pass
 "backpropagation"

[Draw this by hand. [gradients.pdf](#)] Draw the black diagram first. Then the goal (upper right). Then the green and red expressions, from left to right, leaving out the green arrows. Then the green arrows, starting at the right side of the page and moving left. Lastly, write the text at the bottom. (Use the same procedure for the next two figures.)]

[What if a unit's output goes to more than one unit? Then we need to understand a more complicated version of the chain rule. Let's try it with an expression from least-squares linear regression.]



[Draw this by hand. [gradientsmulti.pdf](#)]

[Here we're using a standard rule of multivariate calculus:]

$$\frac{\partial}{\partial \tau} L(z_1(\tau), z_2(\tau)) = \frac{\partial L}{\partial z_1} \frac{\partial z_1}{\partial \tau} + \frac{\partial L}{\partial z_2} \frac{\partial z_2}{\partial \tau} = \nabla_z L \cdot \frac{\partial}{\partial \tau} z$$

[Observe that we're doing dynamic programming here. We're computing the solutions of subproblems, then using each solution to compute the solutions of several bigger problems.]

The Backpropagation Alg.

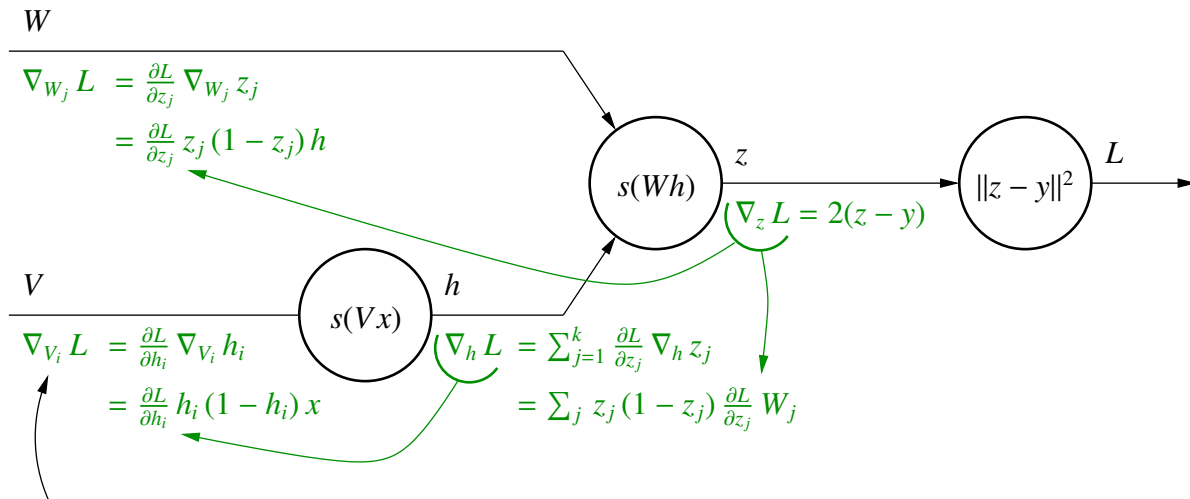
[Backpropagation is a dynamic programming algorithm for computing the gradients we need to do neural net stochastic gradient descent in time linear in the number of weights.]

V_i^T is row i of weight matrix V [and likewise for rows of W]

Recall $s'(\gamma) = s(\gamma)(1 - s(\gamma))$

$$\begin{aligned}
 h_i &= s(V_i \cdot x), \text{ so} & \nabla_{V_i} h_i &= s'(V_i \cdot x) x = h_i(1 - h_i) x \\
 z_j &= s(W_j \cdot h), \text{ so} & \nabla_{W_j} z_j &= s'(W_j \cdot h) h = z_j(1 - z_j) h \\
 & & \nabla_h z_j &= z_j(1 - z_j) W_j
 \end{aligned}$$

[Here is the arithmetic expression for the same neural network I drew for you three illustrations ago. It looks very different when you depict it like this, but don't be fooled; it's exactly the same network I started with. But now we treat the weights V and W as the inputs, rather than the point x .]



Compute $\nabla_V L, \nabla_W L$ one row at a time.

[Draw this by hand. [backpropalg.pdf](#)]