16 Neural Networks

NEURAL NETWORKS

Can do both classification & regression.

[They tie together many ideas from the course: perceptrons, linear regression, 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 unfortunate happened in 1969. Marvin Minsky, one of the founding fathers of AI, and Seymour Papert published a book called "Perceptrons." Sounds promising? Well, part of the book was devoted to things perceptrons can't do. One of those things is XOR.]

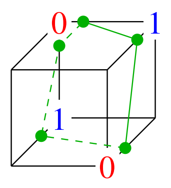
		x_1	
	XOR	0	1
x_2	0	0	1
	1	1	0

[Think of the four outputs here as training 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.]

[The XOR problem is also called parity, especially when you have more features: the input is a bunch of bits and you answer whether the number of 1's is even or odd. It was known even then that you could solve parity problems by adding extra layers of perceptrons, but Minsky and Papert gave technical proofs about some circumstances where this can't be done, and those limitations were misinterpreted. 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 first "AI Winter." Shortly after the book was published, Frank Rosenblatt died in a boating accident.]

[There are several almost obvious ways to get around the XOR problem. Here's the easiest.]

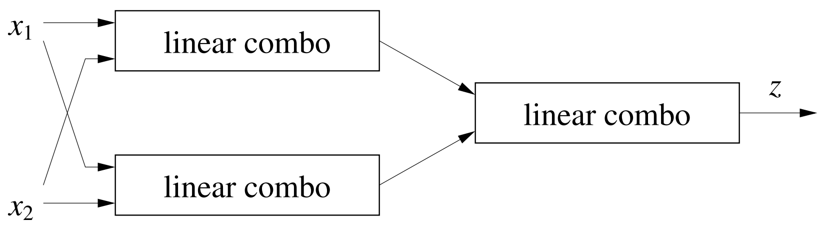
If you add one new quadratic feature, $x_1 x_2$, XOR is linearly separable in 3D.



[Draw this by hand. $\boxed{\text{xorcube.pdf}}$]

[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 logic circuits. Suppose I put together some linear decision functions like this.]



[Draw this by hand. $\boxed{\text{lincombo.pdf}}$]

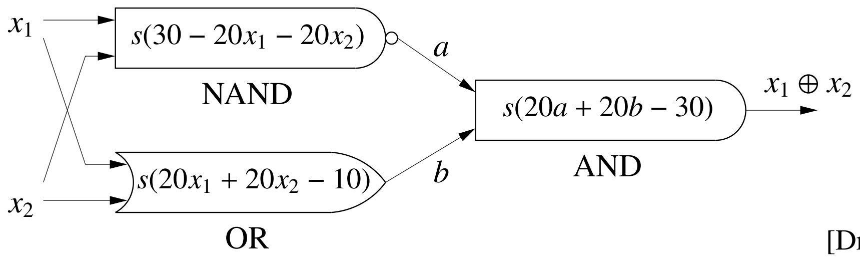
[Interpret the output as true if z is greater than one-half or false if z is less than one-half. Can I do XOR with this?]

A linear combo of linear combos 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 sends 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. That's what people usually use in practice these days.]

[However, the traditional choice was 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. And we know that the logistic is often a good model for posterior probabilities.]

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



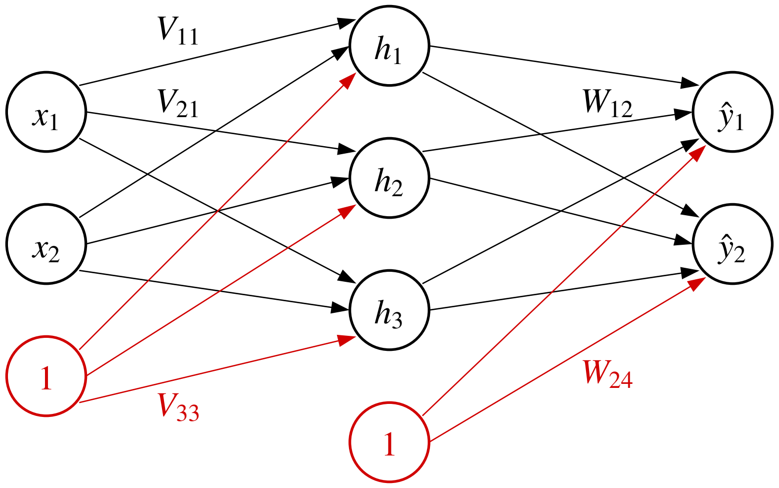
[Draw this by hand. $\boxed{\text{xorgates.pdf}}$]

[The big question is: can an algorithm *learn* a function like this?]

Network with 1 Hidden Layer

Input layer:	$x_1, \ldots, x_d$; $x_{d+1} = 1$	[Index $d + 1$ is the fictitious dimension.]
Hidden units:	$h_1, \ldots, h_m$; $h_{m+1} = 1$	
Output layer:	$\hat{y}_1, \ldots, \hat{y}_k$	

Layer 1 weights:	$m \times (d + 1)$ matrix V	$V_i^\top$ is row i: weights into h_i
Layer 2 weights:	$k \times (m + 1)$ matrix W	$W_i^\top$ is row i: weights into $\hat{y}_i$



[Draw this by hand. $\boxed{\text{neuralnetwork.pdf}}$]

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

For vector u, $s(u) = \begin{bmatrix} s(u_1) \\ s(u_2) \\ \vdots \end{bmatrix}$, $s_1(u) = \begin{bmatrix} s(u_1) \\ s(u_2) \\ \vdots \\ 1 \end{bmatrix}$ [We apply s to a vector component-wise.]

$$\begin{aligned} h &= s_1(Vx) && \ldots\text{that is, } h_i = s(V_i \cdot x) \\ \hat{y} &= s(Wh) = s(W s_1(Vx)) \end{aligned}$$

[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 6 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.]

Training

Usually stochastic or batch gradient descent.

Pick loss fn $L(\underset{\substack{\uparrow \\ \text{predictions}}}{\hat{y}}, \underset{\substack{\uparrow \\ \text{true labels}}}{y})$ (could be vectors) e.g., $L(\hat{y}, y) = \|\hat{y} - y\|^2$.

$$\boxed{\text{Find } V \text{ and } W \text{ that minimize the cost fn } J(V, W) = \frac{1}{n} \sum_{i=1}^{n} L(\hat{y}(X_i), Y_i).}$$

[I'm using a capital Y here because Y is a matrix with one row for each training point and one column for each output unit of the neural net. Each training point has a whole vector of labels Y_i, stored as a row of Y.]

Usually there are many local minima!
[The cost function for a neural net is, generally, not even close to convex. Sometimes, it's possible to wind up in a bad minimum. Usually, you can avoid bad minima by having lots of units in each layer.]

[Now let me ask you this. Suppose we start by setting all the weights to zero, and then we do gradient descent on the weights. What will go wrong?]

[This neural network has a symmetry: there's really no difference between one hidden unit and any other hidden unit. The gradient descent algorithm has no way to break the symmetry between hidden units. You can get stuck in a situation where all the weights out of an input unit have the same value, and all the weights into an output unit have the same value, and they have no way to become different from each other. To avoid this problem, and in the hopes of finding a better local minimum, we start with random weights.]

Let w be a vector containing all the weights in V & W. Batch gradient descent:

```
$w \leftarrow$ vector of random weights
repeat
    $w \leftarrow w - \epsilon \nabla J(w)$
```

[We've just rewritten all the weights as a vector for notational convenience. When you actually write the code, for the sake of speed, you should probably operate directly on the weight matrices V and W.]

[It's important to make sure the random weights aren't too big, because if a unit's output gets too close to zero or one, it can get "stuck," meaning that a modest change in the input values causes barely any change in the output value. Stuck units tend to stay stuck. I'll say more about that next lecture.]

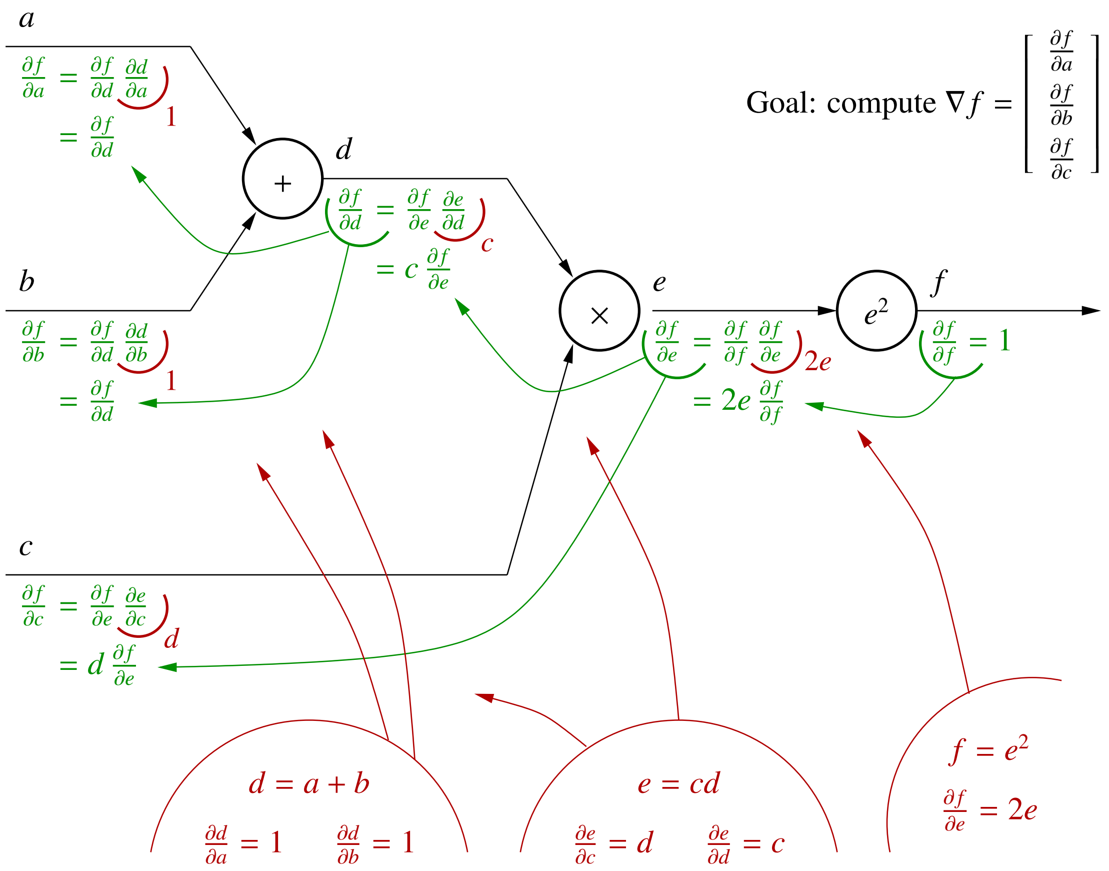
[Instead of batch gradient descent, we can use stochastic gradient descent, which means we use the gradient of one training point's loss function at each step. Typically, we shuffle the points in a random order, or just pick one randomly at each step. I'll say more about that next week.]

[The hard part of this algorithm is computing the gradient. If you simply derive one derivative for each weight, you'll find that for a network with many layers of hidden units, it takes time linear in the number of edges in the neural network to compute a derivative for one weight. Multiply that by the number of weights. We'll spend the rest of this lecture learning to improve the running time to linear in the number of edges.]

Naive gradient computation: $O(\text{edges}^2)$ time
Backpropagation: $O(\text{edges})$ time

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.]



a
$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial d}\frac{\partial d}{\partial a}$ 1
$= \frac{\partial f}{\partial d}$
d
$\frac{\partial f}{\partial d} = \frac{\partial f}{\partial e}\frac{\partial e}{\partial d}$ c
$= c\,\frac{\partial f}{\partial e}$
b
$\frac{\partial f}{\partial b} = \frac{\partial f}{\partial d}\frac{\partial d}{\partial b}$ 1
$= \frac{\partial f}{\partial d}$
$+$
$\times$
e
$\frac{\partial f}{\partial e} = \frac{\partial f}{\partial f}\frac{\partial f}{\partial e}$ $2e$
$= 2e\,\frac{\partial f}{\partial f}$
e^2
f
$\frac{\partial f}{\partial f} = 1$
Goal: compute $\nabla f = \begin{bmatrix} \frac{\partial f}{\partial a} \\ \frac{\partial f}{\partial b} \\ \frac{\partial f}{\partial c} \end{bmatrix}$
c
$\frac{\partial f}{\partial c} = \frac{\partial f}{\partial e}\frac{\partial e}{\partial c}$ d
$= d\,\frac{\partial f}{\partial e}$
$d = a + b$
$\frac{\partial d}{\partial a} = 1$ $\frac{\partial d}{\partial b} = 1$
$e = cd$
$\frac{\partial e}{\partial c} = d$ $\frac{\partial e}{\partial d} = c$
$f = e^2$
$\frac{\partial f}{\partial e} = 2e$

Each value z gives partial derivative of the form

$$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial n}\,\frac{\partial n}{\partial z}$$

computed during forward pass

where z is an input to n.

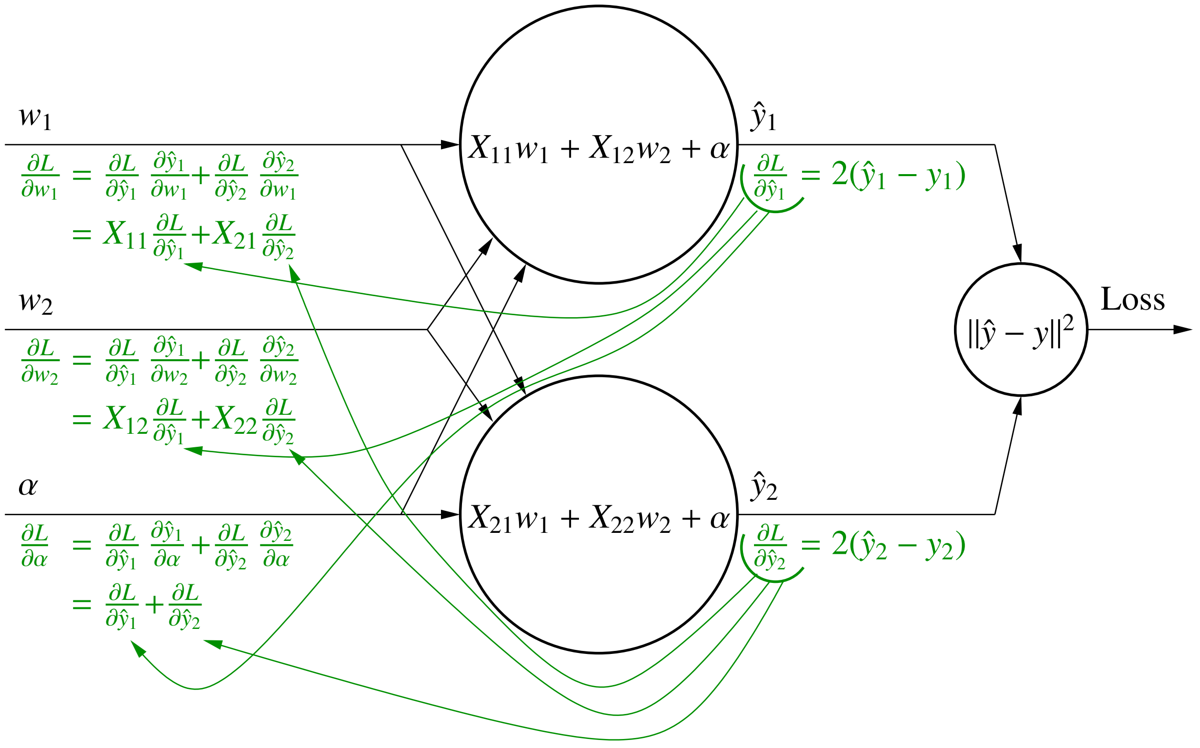
computed during backward pass after forward pass
"backpropagation"

[Draw this by hand. $\boxed{\text{gradientsarith.pdf}}$ 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. This is a standard rule of multivariate calculus:]

$$\frac{\partial}{\partial \alpha} L(y_1(\alpha), y_2(\alpha)) = \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial \alpha} + \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial \alpha} = \nabla_y L \cdot \frac{\partial}{\partial \alpha} y$$

[With this rule, let's compute gradients for an expression from least-squares linear regression.]



w_1
$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}_1}\frac{\partial \hat{y}_1}{\partial w_1} + \frac{\partial L}{\partial \hat{y}_2}\frac{\partial \hat{y}_2}{\partial w_1}$
$= X_{11}\frac{\partial L}{\partial \hat{y}_1} + X_{21}\frac{\partial L}{\partial \hat{y}_2}$
w_2
$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \hat{y}_1}\frac{\partial \hat{y}_1}{\partial w_2} + \frac{\partial L}{\partial \hat{y}_2}\frac{\partial \hat{y}_2}{\partial w_2}$
$= X_{12}\frac{\partial L}{\partial \hat{y}_1} + X_{22}\frac{\partial L}{\partial \hat{y}_2}$
α
$\frac{\partial L}{\partial \alpha} = \frac{\partial L}{\partial \hat{y}_1}\frac{\partial \hat{y}_1}{\partial \alpha} + \frac{\partial L}{\partial \hat{y}_2}\frac{\partial \hat{y}_2}{\partial \alpha}$
$= \frac{\partial L}{\partial \hat{y}_1} + \frac{\partial L}{\partial \hat{y}_2}$
$X_{11}w_1 + X_{12}w_2 + \alpha$
$X_{21}w_1 + X_{22}w_2 + \alpha$
$\hat{y}_1$
$\frac{\partial L}{\partial \hat{y}_1} = 2(\hat{y}_1 - y_1)$
$\hat{y}_2$
$\frac{\partial L}{\partial \hat{y}_2} = 2(\hat{y}_2 - y_2)$
$\|\hat{y} - y\|^2$
Loss

[Draw this by hand. $\boxed{\text{gradientsmulti.pdf}}$]

[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.]

[In one sense, all we've done here is to rederive the fact that the gradient of the least-squares regression cost function is $\nabla_w L = 2X^\top(\hat{y} - y)$, where $\hat{y} = Xw$. But the way we've divided it into a forward pass and a backward pass gives us a way to generalize it by adding more layers of computations.]

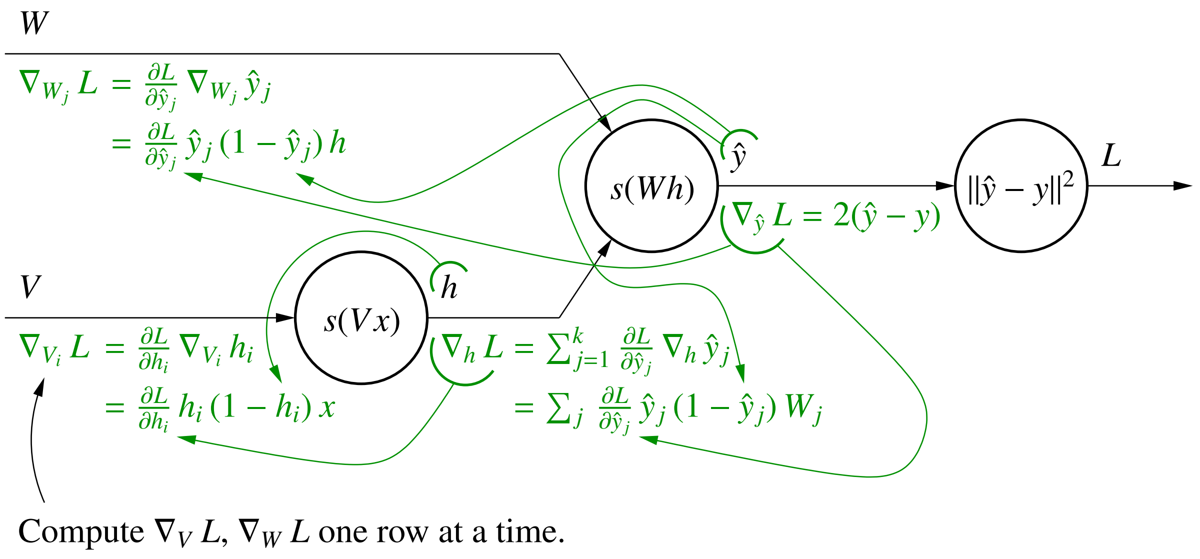
The Backpropagation Alg.

[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.]

Recall $s'(\gamma) = s(\gamma)\,(1 - s(\gamma))$; $V_i^\top$ is row i of weight matrix V [and likewise for rows of W]

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

[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.]



W
$\nabla_{W_j} L = \frac{\partial L}{\partial \hat{y}_j} \nabla_{W_j} \hat{y}_j$
$= \frac{\partial L}{\partial \hat{y}_j} \hat{y}_j\,(1 - \hat{y}_j)\, h$
$s(Wh)$
$\hat{y}$
$\|\hat{y} - y\|^2$
L
$\nabla_{\hat{y}} L = 2(\hat{y} - y)$
V
$s(Vx)$
h
$\nabla_{V_i} L = \frac{\partial L}{\partial h_i} \nabla_{V_i} h_i$
$= \frac{\partial L}{\partial h_i} h_i\,(1 - h_i)\, x$
$\nabla_h L = \sum_{j=1}^{k} \frac{\partial L}{\partial \hat{y}_j} \nabla_h \hat{y}_j$
$= \sum_j \frac{\partial L}{\partial \hat{y}_j} \hat{y}_j\,(1 - \hat{y}_j)\, W_j$
Compute $\nabla_V L$, $\nabla_W L$ one row at a time.

[Draw this by hand. $\boxed{\text{backalg.pdf}}$]

[Note that h and $\hat{y}$ are computed during the forward pass, and $\nabla_{\hat{y}} L$, $\nabla_h L$, $\nabla_W L$, and $\nabla_V L$ are computed during the backward pass. In particular, we can't compute $\nabla_V L$ until after we compute $\nabla_h L$, and we can't compute that until after we compute $\nabla_{\hat{y}} L$. The loss L doesn't need to be explicitly computed at all! We can compute all the gradients without it.]