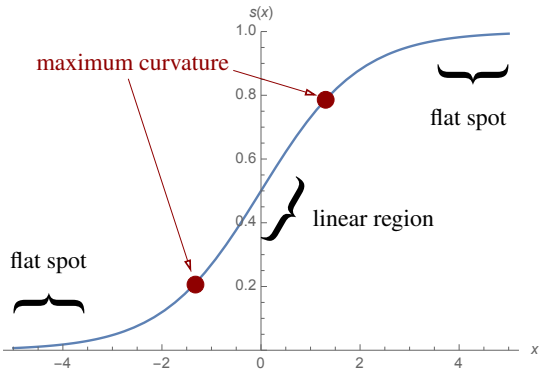
17 Vanishing Gradients; ReLUs; Output Units and Losses; Neurobiology

THE VANISHING GRADIENT PROBLEM; ReLUs

[Last lecture, we put a logistic function at the output of every unit except the input units. These units are called sigmoid units. But in practice, sigmoid units are usually a poor choice for hidden layers.]

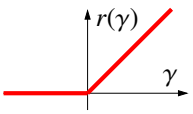
Problem: When unit output s is close to 0 or 1 for most training points, $s' = s(1-s) \approx 0$, so gradient descent changes s very slowly. Unit is "stuck." Slow training.



logistic.pdf [Draw flat spots, "linear" region, & maximum curvature points (at $s(\lambda) \doteq 0.21$ and $s(\lambda) \doteq 0.79$) of the sigmoid function. Ideally, we would stay away from the flat spots.]

[This is called the vanishing gradient problem. The more layers your network has, the more problematic this problem becomes. Most of the early attempts to train deep, many-layered neural nets failed.]

Solution: Replace sigmoids with ReLUs: rectified linear units.
ramp fn: $r(\gamma) = \max\{0, \gamma\}$.



$$r'(\gamma) = \begin{cases} 1, & \gamma \geq 0, \\ 0, & \gamma < 0. \end{cases}$$

[The derivative is not defined at zero, but we just pretend it is for the sake of gradient descent.]

Most neural networks today use ReLUs for the hidden units.
[However, it is still common to use sigmoids for the output units in classification problems.]

[ReLUs are preferred over sigmoids as hidden units because in practice, they're much less likely to get stuck. But the derivative r' is sometimes zero, so you might wonder if ReLUs can get stuck too. Fortunately, it's rare for a ReLU's gradient to be zero for *all* the training data; it's usually zero for just some training points. But yes, ReLUs sometimes get stuck too; just not as often as sigmoids.]

[The output of a ReLU can be arbitrarily large; the fact that ReLUs don't saturate like sigmoids do leaves them vulnerable to a related problem called the "exploding gradient problem." It is not a big problem in shallow networks, but it becomes a big problem in deep or recurrent networks.]

[Even though ReLUs are linear in each half of their range, they're still nonlinear enough to easily compute functions like XOR. Of course, if you replace sigmoids with ReLUs, you have to change the derivation of backprop to reflect the changes in the gradients. We'll do that later in this lecture.]

OUTPUT UNITS

[Many neural networks use ReLUs for all or most of the hidden units, but ReLUs are rarely used as output units. The output units are chosen to fit the application, and there are three common choices.]

Most output units are linear units (regression) or sigmoid/logistic or softmax units (classification).

[When you train a neural network with these output units by gradient descent, the last layer of edges of the network is solving a problem in linear regression, logistic regression, or softmax regression by gradient descent. Or maybe all three!]

(1) Linear units for regression.
Given vector h of unit values in last hidden layer, output layer computes $\hat{y} = Wh$.
Activation fn is the identity fn. [You could say there is no activation function.]
Then the final layer of edges is doing linear regression (on values of h & y)!
Usually trained with squared-error loss. If so, it's least-squares linear regression.

[When we train a neural network by gradient descent, each linear output unit finds the solution of a linear regression problem by gradient descent. In principle we could find the weights entering that unit by solving the normal equations. But we don't, because the hidden unit values keep changing during training.]

(2) Sigmoid units [aka logistic units] for [two-class] classification.
Let $y \in \mathbb{R}^k$ be vector of labels; $y_i \in [0, 1]$.
Given hidden layer h, output layer computes pre-activation $a = Wh \in \mathbb{R}^k$ and applies sigmoid activations to obtain prediction $\hat{y} = s(a)$.

[Here, s is the logistic function applied component-wise to the vector a. The labels y_i are usually 0's and 1's, but $\hat{y}_i$ can never be exactly 0 or 1. So it might be better to choose target labels like $y_i = 0.05$ or $y_i = 0.95$, because then a neural network with enough weights and sufficiently wide layers can achieve $\hat{y} = y$ exactly for every training point! Unless there are co-located training points with different labels.]

Loss fn: Use logistic loss instead of squared error. Fixes vanishing gradients at output!
[The logistic loss function prevents output units from suffering the vanishing gradient problem, but it can't solve the vanishing gradient problem for hidden units. So we don't use sigmoid hidden units.]

[When we train a neural network by gradient descent, each sigmoid output unit finds the solution of a logistic regression problem by gradient descent.]

(3) Softmax units for k-class classification.
[E.g., in the MNIST digit recognition problem, we would have $k = 10$ softmax output units, one for each digit class.]

Let $y \in \mathbb{R}^k$ be vector of labels for training pt x [indicating x's membership in the k classes].
[It is easy to design a neural network to solve more than one multi-class classification problem simultaneously, but for ease of notation let's suppose we're solving just one, so there are only k output units.]

Strongly recommended: choose training labels so $\sum_{i=1}^{k} y_i = 1$.

We commonly use a one-hot encoding: one label is 1, the others are 0.
[But one-hot encoding has a disadvantage we've already discussed for sigmoids: each softmax prediction $\hat{y}_i$ can never be exactly 1 or 0. It might be better to choose target labels such as 0.9, 0.05, and 0.05. Think of the labels as posterior probabilities, so they should sum to 1.]

Given hidden layer h, output layer computes pre-activation $a = Wh \in \mathbb{R}^k$ and applies softmax activation to obtain prediction $\hat{y} \in \mathbb{R}^k$.

Softmax output is $\hat{y}_i(a) = \dfrac{e^{a_i}}{\sum_{j=1}^{k} e^{a_j}}$. Each $\hat{y}_i \in (0, 1)$; $\sum_{i=1}^{k} \hat{y}_i = 1$.

[Interpret $\hat{y}_i$ as an estimate of the posterior probability that the input belongs to class i.]

Loss fn: Use cross-entropy. Fixes vanishing gradients at output.

For k-class softmax output, cross-entropy is $L(\hat{y}, y) = -\sum_{i=1}^{k} y_i \ln \hat{y}_i$.
↑ true labels
↑ prediction
} k-vectors

[When we train a neural network by gradient descent, if there are softmax output units, those units find the solution of a softmax regression problem by gradient descent.]

[Cross-entropy losses are only for softmax and sigmoid outputs. For linear or ReLU outputs, cross-entropy makes no sense, but squared error loss makes sense.]

Backpropagation for Outputs

For backprop, we need $\nabla_W L$ and $\nabla_h L$, where h is last hidden unit layer, W is weights of last edge layer.

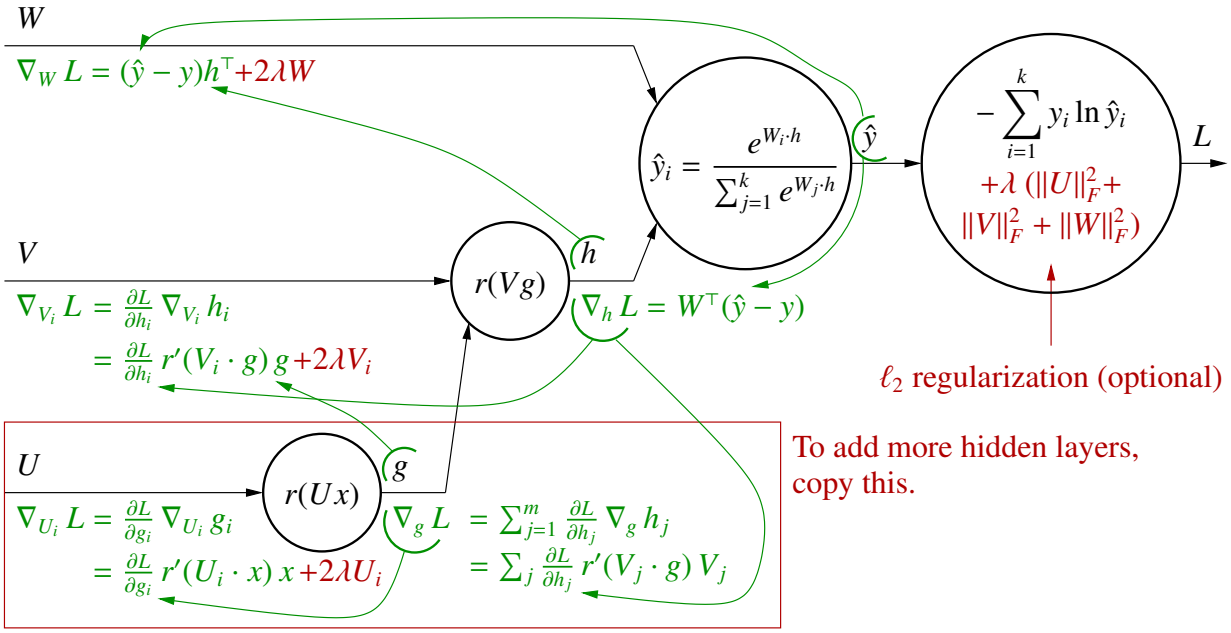
output + loss	linear + squared error	sigmoid + logistic loss	softmax + cross-entropy
$\hat{y} =$	Wh	$s(Wh)$	$\text{softmax}(Wh)$
$L(\hat{y}, y) =$	$\|\hat{y} - y\|^2$	$-\sum_i (y_i \ln \hat{y}_i + (1 - y_i) \ln(1 - \hat{y}_i))$	$-\sum_i y_i \ln \hat{y}_i$
$\nabla_W L =$	$2(\hat{y} - y)\, h^\top$	$(\hat{y} - y)\, h^\top$	$(\hat{y} - y)\, h^\top$
$\nabla_h L =$	$2W^\top(\hat{y} - y)$	$W^\top(\hat{y} - y)$	$W^\top(\hat{y} - y)$
			assuming $\sum_{i=1}^k y_i = 1$

[It's interesting that all three types of outputs produce essentially the same form of gradients for the final layer of the network, except that the predictions $\hat{y}$ are different in each case. This is true even though each linear or sigmoid output unit is independent, but the softmax outputs units are coupled with each other.]

[Observe that even for sigmoid and softmax units, both $\nabla_W L$ and $\nabla_h L$ are linear in the error $\hat{y} - y$. This is a nice outcome, when you consider the exponentials and logarithms we started with. It implies that sigmoid units with logistic loss do not get stuck when the sigmoid derivatives are small. This is related to the fact that the logistic loss goes to infinity as the predicted value $\hat{y}_i$ approaches zero or one. The vanishing gradient of the sigmoid unit is compensated for by the exploding gradient of the logistic loss.]

Note: we don't need to compute $\nabla_{\hat{y}} L$.
Instead, we eliminate $\hat{y}$ by substituting $\hat{y}(W, h)$ into $L(\hat{y}, y)$.
[. . . before taking derivatives. This makes it easier both to derive the math and to write the code.]

[Now I will show you how to perform backpropagation for two hidden layers of ReLU units, a k-class softmax output, the cross-entropy loss function, and ℓ_2 regularization—which may improve test accuracy.]



[Draw this by hand. backpropsoft2.pdf]

[Note that $r(Ux)$ is the ramp function applied component-wise to the vector Ux. The derivative $r'(U_i \cdot x)$ is always zero or one. Observe that we don't need to compute the loss L at all. We also don't compute $\nabla_{\hat{y}} L$, as we said above, but we do need to compute the value of $\hat{y}$ to compute gradients.]

Derivations

[I won't go over this page of derivations in lecture, but I include them here for completeness. Students who want to understand neural networks deeply should spend some time going through these.]

Linear output units ($\hat{y} = Wh$) with squared error loss:

$$\begin{aligned}
L(\hat{y}, y) &= \|\hat{y} - y\|^2 = \|Wh - y\|^2 = \sum_{i=1}^{k} (W_i \cdot h - y_i)^2, \\
\nabla_{W_i} L &= 2(W_i \cdot h - y_i)\, h = 2(\hat{y}_i - y_i)\, h, \\
\nabla_W L &= 2(\hat{y} - y)\, h^\top, \\
\nabla_h L &= 2W^\top (Wh - y) = 2W^\top (\hat{y} - y).
\end{aligned}$$

Sigmoid [logistic] output units ($\hat{y} = s(a) = s(Wh)$) with logistic loss:

$$\begin{aligned}
L(\hat{y}, y) &= -\sum_{i=1}^{k} (y_i \ln \hat{y}_i + (1 - y_i) \ln(1 - \hat{y}_i)) = -\sum_{i=1}^{k} \left(y_i \ln \frac{1}{1 + e^{-a_i}} + (1 - y_i) \ln \left(1 - \frac{1}{1 + e^{-a_i}} \right) \right) \\
&= \sum_{i=1}^{k} \left(y_i \ln(1 + e^{-a_i}) - (1 - y_i) \ln \frac{e^{-a_i}}{1 + e^{-a_i}} \right) \\
&= \sum_{i=1}^{k} \left(y_i \ln(1 + e^{-a_i}) - (1 - y_i)(-a_i - \ln(1 + e^{-a_i})) \right) = \sum_{i=1}^{k} \left((1 - y_i) a_i + \ln(1 + e^{-a_i}) \right), \\
\frac{\partial L}{\partial a_i} &= 1 - y_i - \frac{e^{-a_i}}{1 + e^{-a_i}} = \frac{1}{1 + e^{-a_i}} - y_i = \hat{y}_i - y_i, \\
a_i &= W_i \cdot h, \qquad \nabla_{W_i} a_i = h, \qquad \nabla_h a_i = W_i, \\
\nabla_{W_i} L &= \frac{\partial L}{\partial a_i} \nabla_{W_i} a_i = (\hat{y}_i - y_i)\, h, \\
\nabla_W L &= (\hat{y} - y)\, h^\top, \\
\nabla_h L &= \sum_{i=1}^{k} \frac{\partial L}{\partial a_i} \nabla_h a_i = \sum_{i=1}^{k} (\hat{y}_i - y_i) W_i = W^\top (\hat{y} - y).
\end{aligned}$$

Softmax output units ($\hat{y} = \text{softmax}(a) = \text{softmax}(Wh)$) with cross-entropy loss:

[This derivation uses the assumption that $\sum_{i=1}^{k} y_i = 1$ for each training point's labels.]

$$\begin{aligned}
L(\hat{y}, y) &= -\sum_{i=1}^{k} y_i \ln \hat{y}_i = -\sum_{i=1}^{k} y_i \left(a_i - \ln \sum_{j=1}^{k} e^{a_j} \right) = -\sum_{i=1}^{k} y_i a_i + \ln \sum_{j=1}^{k} e^{a_j}, \\
\frac{\partial L}{\partial a_i} &= -y_i + \left(e^{a_i} \Big/ \sum_{j=1}^{k} e^{a_j} \right) = \hat{y}_i - y_i.
\end{aligned}$$

From here, we repeat the last four lines of the sigmoid derivation.

NEUROBIOLOGY

[The field of artificial intelligence started with some wrong premises. The early AI researchers attacked problems like chess and theorem proving, because they thought those exemplified the essence of intelligence. They didn't pay much attention at first to problems like vision and speech understanding. Any four-year-old can do those things, and so researchers underestimated their difficulty.]

[Today, we know better. Computers can beat world chess champions, but they still can't play with toys well. We've come to realize that rule-based symbol manipulation is not the primary defining mark of intelligence. Even rats do computations that we're hard pressed to match with our computers. We've also come to realize that these are different classes of problems that require very different styles of computation. Brains and computers have very different strengths and weaknesses, which reflect their different computing styles.]

[Neural networks are partly inspired by the workings of actual brains. Let's take a look at a few things we know about biological neurons, and contrast them with both neural nets and traditional computation.]

- CPUs: largely sequential, nanosecond gates, fragile if gate fails
  superior for arithmetic, logical rules, perfect key-based memory
- Brains: very parallel, millisecond neurons, fault-tolerant

  [Neurons are continually dying. You've probably lost a few since this lecture started. But you probably didn't notice. And that's interesting, because it points out that our memories are stored in our brains in a diffuse representation. There is no one neuron whose death will make you forget that $2 + 2 = 4$. Artificial neural nets often share that resilience. Brains and neural nets seem to superpose memories on top of each other, all stored together in the same weights, sort of like a hologram.]

  [In the 1920's, the psychologist Karl Lashley conducted experiments to identify where in the brain memories are stored. He trained rats to run a maze, and then made lesions in different parts of the cerebral cortex, trying to erase the memory trace. Lashley failed; his rats could still find their way through the maze, no matter where he put lesions. He concluded that memories are not stored in any one area of the brain, but are distributed throughout it. Neural networks, properly trained, can duplicate this property.]

  superior for vision, speech, associative memory

  [By "associative memory," I mean noticing connections between things. One thing our brains are very good at is retrieving a pattern if we specify only a portion of the pattern.]

[It's impressive that even though a neuron needs a few milliseconds to transmit information to the next neurons downstream, we can perform very complex tasks like interpreting a visual scene in a tenth of a second. This is possible because neurons run in parallel, but also because of their computation style.]

[Neural nets try to emulate the parallel, associative thinking style of brains, and they are the best techniques we have for many fuzzy problems, including most problems in vision and speech. Not coincidentally, neural nets are also inferior at many traditional computer tasks such as multiplying 10-digit numbers or compiling source code.]