

Tools & Models for Data Science

Deep Neural Networks (2): Learning

Chris Jermaine & Risa Myers

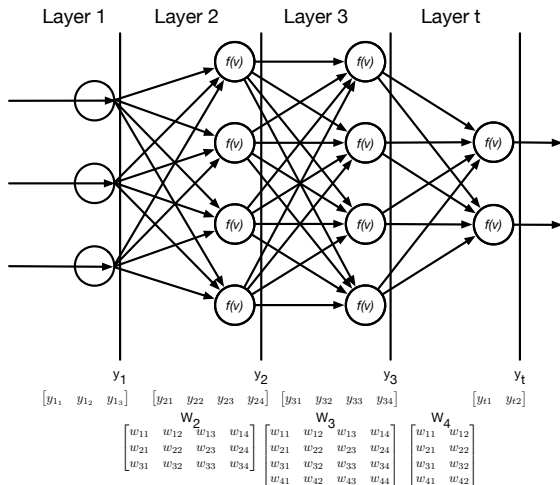
Rice University



Learning Is Accomplished Via GD

- GD in the context of a NN gives rise to the “back-propagation” algorithm
 - First described in 1975
 - Recall params to deep network are all of the weight matrices: $W = W_1, W_2, \dots$
 - “Learning” involves tweaking values of those matrices
 - ... to minimize a loss function

NN Variables



- $w_{i,j,k}$:
- i = layer
- j = input neuron
- k = output neuron

- Just like all GD algorithms, it is iterative
 - Assume loss function $L(W)$
 - with learning rate, λ
 - and where W is a tensor of 2D weight matrices
 - Then we have

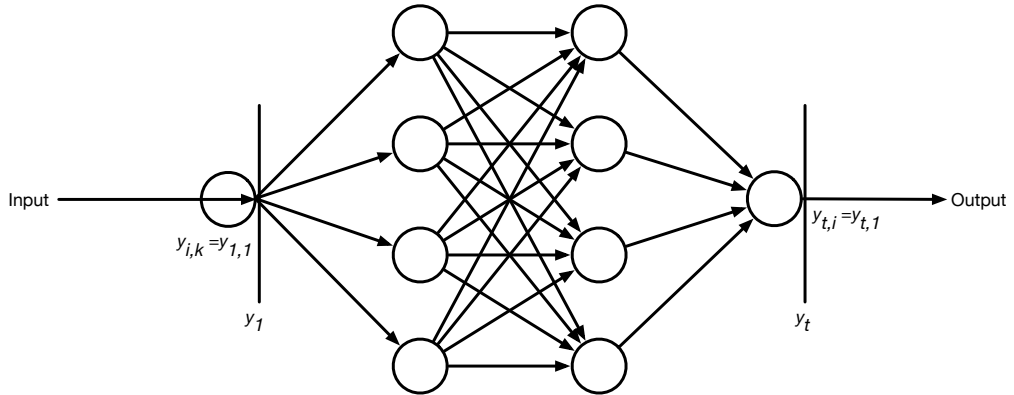
Make a non-stupid guess for each W_i ;

```
repeat {  
     $W \leftarrow W - \lambda \nabla L(W)$ ;  
} while (change in loss  $> \epsilon$ )
```

Choosing a Loss Function

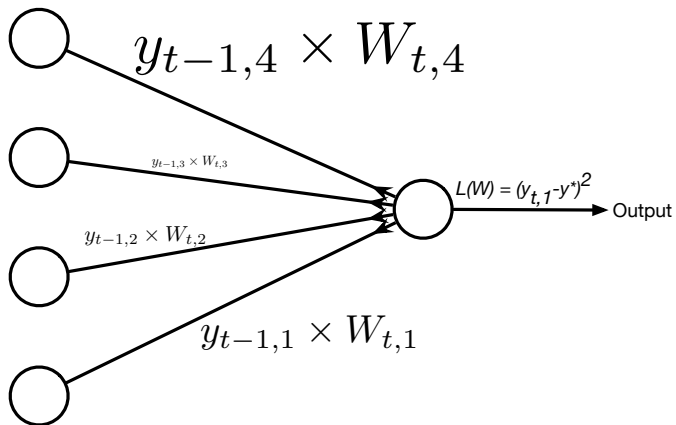
- There are many possible loss functions
- Let's assume one output neuron, one input data point;
 - Use $y_{i,k}$ to denote output from layer i , neuron k
 - So in layer t ("top") just have $y_{t,1}$
 - Hoped-for value is y^*
 - Let's use squared loss: $L(W) = (y_{t,1} - y^*)^2$
- Extension to many data points, many outputs straightforward.

Our 1 I/O NN



- 1 Make a non-stupid guess for the weights
 - 2 Compute the value of the Loss function
 - 3 Redistribute the error from the Loss function among the weights
- Weights that cause the error to increase will have larger activation values (y)
 - And these will have larger values for $\frac{\partial L}{\partial W}$

Push Back Loss



From Inputs to Outputs to Weights

- Each input data point contributes to the cost and to the gradient
- Each “pulls” gradient descent in a certain direction
- Sum together the contribution to determine the overall direction of the gradient
- Move the weights that way

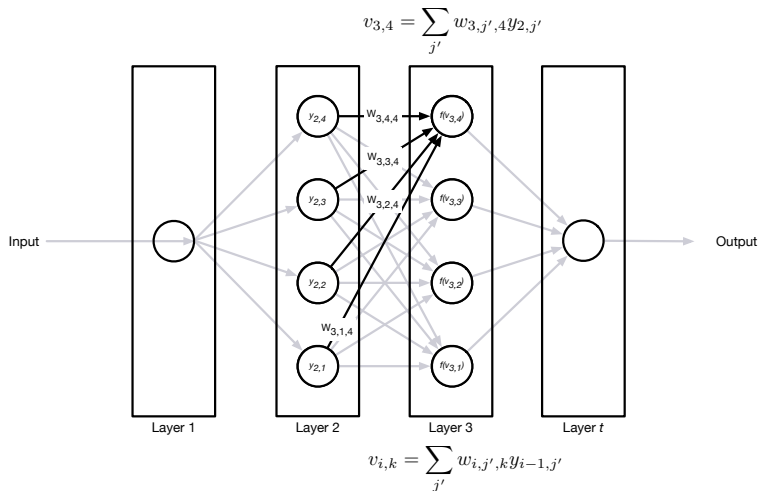
Computing the Gradient

- Need to be able to differentiate L wrt each weight
- Done by applying the chain rule:

$$\frac{\partial L}{\partial w_{i,j,k}} = \frac{\partial L}{\partial y_{i,k}} \frac{\partial y_{i,k}}{\partial v_{i,k}} \frac{\partial v_{i,k}}{\partial w_{i,j,k}}$$

- $v_{i,k}$ is the weighted sum of activations sent into layer i , neuron k
- $v_{i,k} = \sum_{j'} w_{i,j',k} y_{i-1,j'}$

Computing the Neuron Input



Computing the Gradient

- Need to be able to differentiate L wrt each weight
- Done by applying the chain rule:

$$\frac{\partial L}{\partial w_{i,j,k}} = \frac{\partial L}{\partial y_{i,k}} \frac{\partial y_{i,k}}{\partial v_{i,k}} \frac{\partial v_{i,k}}{\partial w_{i,j,k}}$$
$$= \frac{\text{Loss}}{\text{Neuron Output}} \frac{\text{Neuron Output}}{\text{Neuron input}} \frac{\text{Neuron Input}}{\text{Weights}}$$

- Application of chain rule comes from chain of dependencies:
 - neuron output $y_{i,k}$ used to compute the loss
 - neuron input $v_{i,k}$ used to compute neuron output $y_{i,k}$
 - weight $w_{i,j,k}$ used to compute neuron input $v_{i,k}$
- Note: No one actually does the math!

Now We Look At the Different Parts

■ Explain

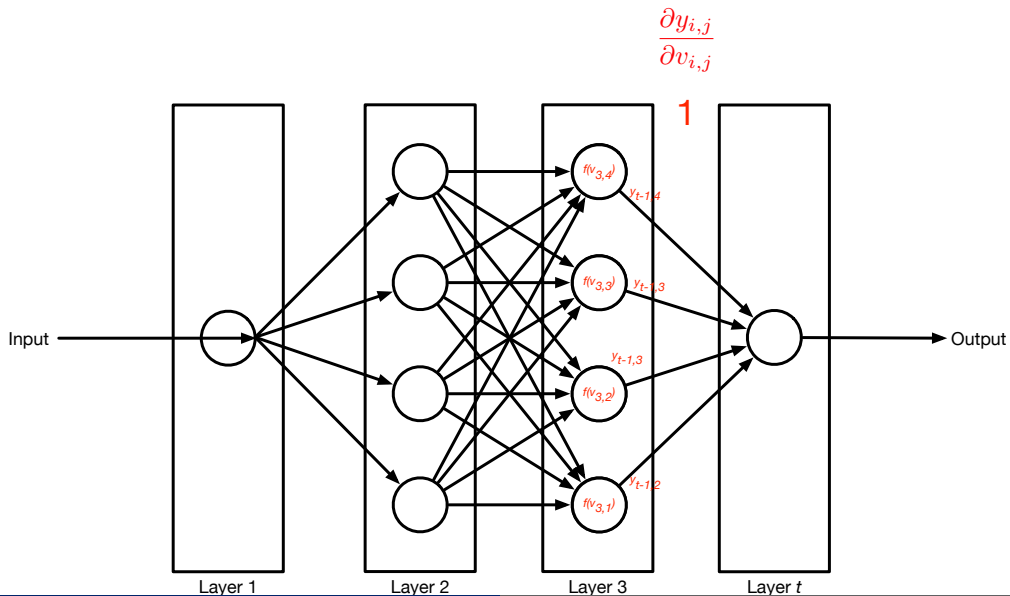
$$\frac{\partial L}{\partial w_{i,j,k}} = \frac{\partial L}{\partial y_{i,k}} \frac{\partial y_{i,k}}{\partial v_{i,k}} \frac{\partial v_{i,k}}{\partial w_{i,j,k}}$$
$$\frac{\partial L}{\partial w_{i,j,k}} = 3 \times 1 \times 2$$

- 1 Start with differentiating neuron output
- 2 Then differentiating the neuron input
- 3 And finally differentiating the loss

- We start with neuron output
- That is,

$$\frac{\partial y_{i,j}}{\partial v_{i,j}}$$

Computing the Gradient - Neuron Output



Differentiating Neuron Output

- Assume activation function is the logistic function

$$f(v_{i,j}) = \frac{1}{1 + e^{-v_{i,j}}}$$

- Popular because result is magically simple
- Will skip algebra, but differentiating the logistic function gives us:

$$\frac{df}{dv_{i,j}} = f(v_{i,j})(1 - f(v_{i,j}))$$

- So, since $y_{i,j} = f(v_{i,j})$,

$$\frac{\partial y_{i,j}}{\partial v_{i,j}} = \frac{df}{dv_{i,j}} = f(v_{i,j})(1 - f(v_{i,j}))$$

- Which is surprisingly nice for Logistic Regression

Differentiating Neuron Input

- Now time to deal with

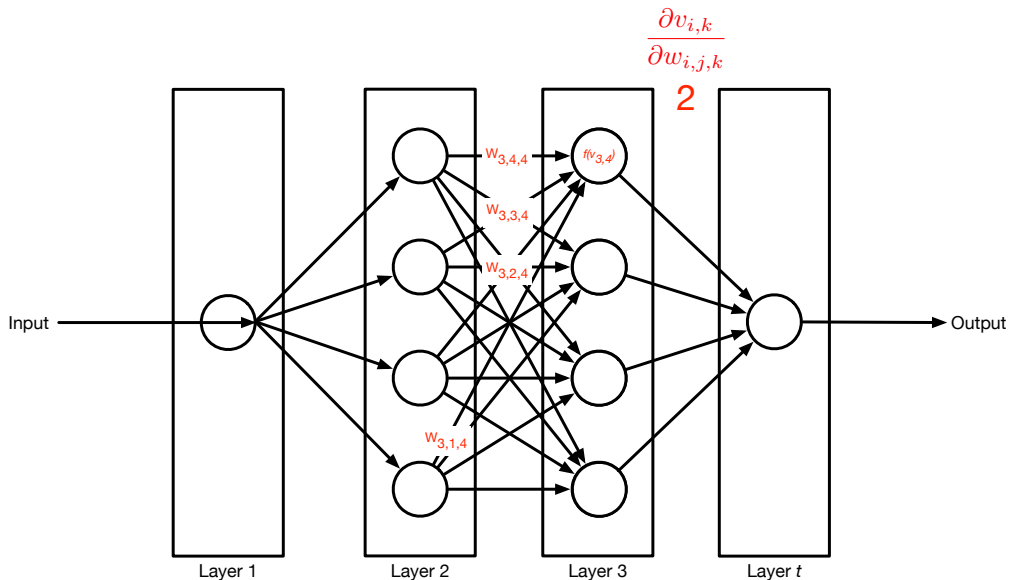
$$\frac{\partial v_{i,k}}{\partial w_{i,j,k}}$$

- Note that $v_{i,k}$ computes a dot product, so:

$$\frac{\partial v_{i,k}}{\partial w_{i,j,k}} = \frac{\partial}{\partial w_{i,j,k}} \sum_{j'} w_{i,j',k} y_{i-1,j'}$$

- This is just $y_{i-1,j}$, since all the j' 's drop out, except for the case where $j' = j$
- We are summing over the prior layer's neurons
- Multiplying those outputs by the weights leading to the current layer
- For each neuron input, we only care about weights that go INTO this neuron
- Can ignore all the other weights in this layer

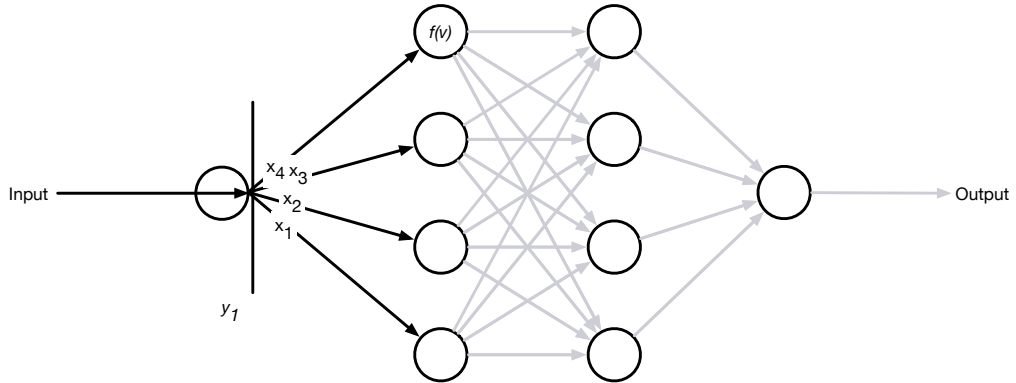
Computing the Gradient - Neuron Input



Differentiating Neuron Input from the Input Layer

- Note that if $i - 1$ is the input layer, then $y_{i-1,j} = x_j$
 - Where x_j is the j th entry in the input vector

Computing the Gradient - Neuron Input



Differentiating the Loss

- Next, need to be able to deal with differentiating the loss

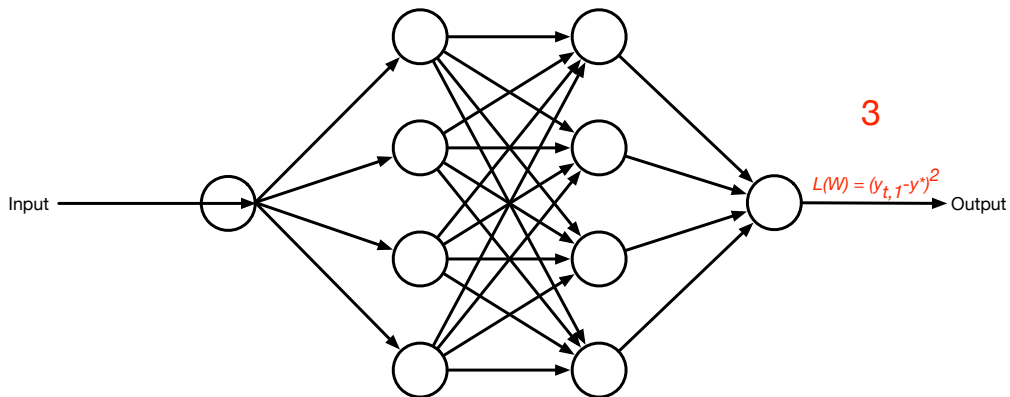
$$\frac{\partial L}{\partial y_{i,j}}$$

- If this is the output layer, it is easy:

$$\frac{\partial L}{\partial y_{t,1}} = 2(y_{t,1} - y^*) \approx (y_{t,1} - y^*)$$

- Recall: y^* is the hoped for output
- Can drop the 2 since will be swallowed into the learning rate

Computing the Gradient - Loss for Output Layer



Differentiating the Loss

- Next, need to be able to deal with differentiating the loss

$$\frac{\partial L}{\partial y_{i,j}}$$

- If this is the output layer, it is easy:

$$\frac{\partial L}{\partial y_{t,1}} = 2(y_{t,1} - y^*) \approx (y_{t,1} - y^*)$$

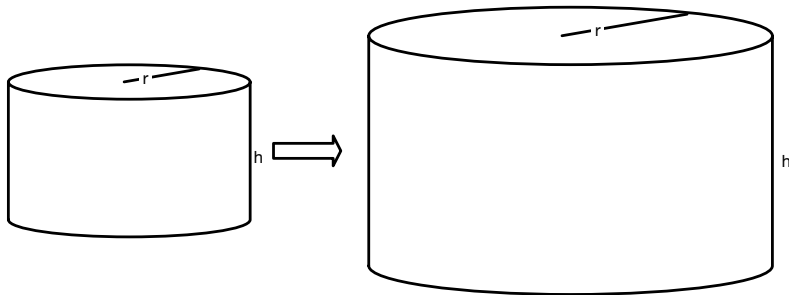
- Recall: y^* is the hoped for output
- Can drop the 2 since will be swallowed into the learning rate
- ? But what about the other layers?

- Quick detour: idea of a “total derivative” (consequence of chain rule)
 - Say we have a function f of several variables x_1, x_2, \dots
 - Each of which is a function of variable t ; we want $\frac{df}{dt}$
 - Is computed via the “total derivative”:

$$\frac{df}{dt} = \frac{\partial f}{\partial x_1} \frac{\partial x_1}{\partial t} + \frac{\partial f}{\partial x_2} \frac{\partial x_2}{\partial t} + \dots$$

Total Derivative Example

- A cylinder has radius and height of 2 units
 - Recall $V = \pi r^2 h$
 - It's getting bigger...
 - Radius increases at 2 units/sec, height at 1 unit/sec



Total Derivative Example

- A cylinder has radius, height of 2 units
 - Recall $V = \pi r^2 h$
 - It's getting bigger...
 - Radius increases at 2 units/sec, height at 1 unit/sec
 - What is the instantaneous rate of increase in volume?

$$\begin{aligned}\frac{dV}{dt} &= \frac{\partial V}{\partial r} \frac{\partial r}{\partial t} + \frac{\partial V}{\partial h} \frac{\partial h}{\partial t} \\ &= \pi 2rh \times \left(\frac{\partial r}{\partial t}\right) + \pi r^2 \times \left(\frac{\partial h}{\partial t}\right) \\ &= \pi \times 2 \times 2 \times 2 \times (2) + \pi \times 2^2 \times (1) = 20\pi \text{ units}^3/\text{sec}\end{aligned}$$

- Note: could substitute $r = 2 + 2t, h = 2 + t$ in $V = \pi r^2 h$
- Would get same answer using “classic” derivative over this new expression

Total Derivative vs. Partial Derivative

- Partial derivative: Other variables are treated as constants
- Total derivative: Other variables are NOT constant

Differentiating Loss for Hidden Layers

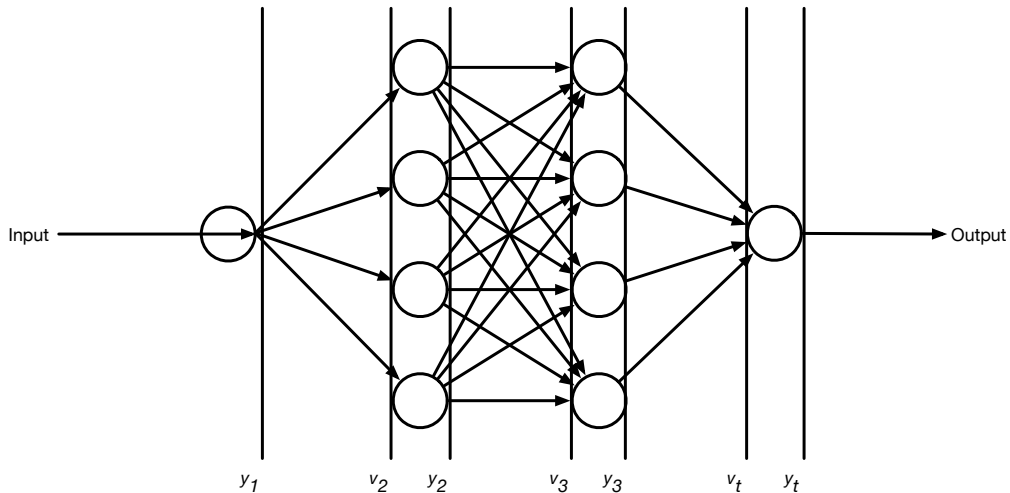
- Why is total derivative relevant?

- Note that Loss is a function of all the neuron inputs
- All v 's at layer $i + 1$: $v_{i+1,1}, v_{i+1,2}, v_{i+1,3}, \dots$
- Where $v_{i+1,k}$ is itself a function of $y_{i,j}$
- (Plus a lot of other things we'll treat as constants)
- This is exactly the case where $\frac{\partial L}{\partial y_{i,j}}$ must be computed using total derivative
- So take the total derivative wrt $y_{i,j}$:

$$\frac{\partial L}{\partial y_{i,j}} = \sum_k \frac{\partial L}{\partial v_{i+1,k}} \frac{\partial v_{i+1,k}}{\partial y_{i,j}}$$

- Partial of L wrt the j th neuron in layer i
- Recall that $y_{i,j}$ depends on all of the input weights

Computing the Gradient - Loss for Hidden Layers



Differentiating the Loss for Hidden Layers

- And since $\frac{\partial v_{i+1,k}}{\partial y_{i,j}} = w_{i+1,j,k}$

- We have:

$$\frac{\partial L}{\partial y_{i,j}} = \sum_k \frac{\partial L}{\partial v_{i+1,k}} w_{i+1,j,k}$$

- Further, note that, by chain rule:

$$\frac{\partial L}{\partial v_{i+1,k}} = \frac{\partial L}{\partial y_{i+1,k}} \frac{\partial y_{i+1,k}}{\partial v_{i+1,k}}$$

- Refer to $\frac{\partial L}{\partial v_{i+1,k}}$ as $\delta_{i+1,k}$.

- So

$$\frac{\partial L}{\partial y_{i,j}} = \sum_k \delta_{i+1,k} w_{i+1,j,k}$$

- Recall,

$$\frac{\partial L}{\partial w_{i,j,k}} = \frac{\partial L}{\partial y_{i,k}} \frac{\partial y_{i,k}}{\partial v_{i,k}} \frac{\partial v_{i,k}}{\partial w_{i,j,k}}$$

- And we can write this as:

$$\begin{aligned} \frac{\partial L}{\partial w_{i,j,k}} &= \left(\sum_{k'} \delta_{i+1,k'} w_{i+1,k,k'} \right) (f(v_{i,k})(1 - f(v_{i,k}))) (y_{i-1,j}) \\ &= \delta_{i,k} (y_{i-1,j}) \end{aligned}$$

Breaking down the Transition

$$\begin{aligned}\frac{\partial L}{\partial w_{i,j,k}} &= \frac{\partial L}{\partial y_{i,k}} \frac{\partial y_{i,k}}{\partial v_{i,k}} \frac{\partial v_{i,k}}{\partial w_{i,j,k}} \\ \frac{\partial L}{\partial w_{i,j,k}} &= \left(\sum_{k'} \delta_{i+1,k'} w_{i+1,k,k'} \right) (f(v_{i,k})(1 - f(v_{i,k}))) (y_{i-1,j}) \\ \frac{\partial L}{\partial v_{i+1,k}} &= \delta_{i+1,k} \\ &= \delta_{i,k} (y_{i-1,j})\end{aligned}$$

The diagram illustrates the chain rule for the partial derivative of the loss function L with respect to the weight $w_{i,j,k}$. It shows how the derivative is broken down into three components: the derivative of L with respect to the output $y_{i,k}$, the derivative of $y_{i,k}$ with respect to the input $v_{i,k}$, and the derivative of $v_{i,k}$ with respect to the weight $w_{i,j,k}$. The second equation shows the derivative of L with respect to $w_{i,j,k}$ as a product of three terms. The first term is a sum over k' of $\delta_{i+1,k'} w_{i+1,k,k'}$, which is then simplified in the third equation to $\delta_{i+1,k}$. The second term is $f(v_{i,k})(1 - f(v_{i,k}))$, which is simplified in the fourth equation to $\delta_{i,k}$. The third term is $y_{i-1,j}$, which remains unchanged. The final result is $\delta_{i,k} (y_{i-1,j})$.

Recursion!!

- Remember our goal
- To be able to differentiate L wrt each weight

$$\begin{aligned}\frac{\partial L}{\partial w_{i,j,k}} &= \left(\sum_{k'} \delta_{i+1,k'} w_{i+1,k,k'} \right) (f(v_{i,k})(1 - f(v_{i,k}))) (y_{i-1,j}) \\ &= \delta_{i,k} (y_{i-1,j})\end{aligned}$$

- This suggests a dynamic programming algorithm!
 - Start at top, recurse back
 - At layer i , to compute each $\frac{\partial L}{\partial w_{i,j,k}}$ you use each $\delta_{i+1,k'}$ from previous layer
 - As a side-effect of computing each $\frac{\partial L}{\partial w_{i,j,k}}$, you compute $\delta_{i,k}$
 - Can record this and use when computing layer $i - 1$

- First, make a forward pass thru the network
 - Compute each $y_{i,k}$, $v_{i,k}$, $w_{i,j,k}$
 - For each possible i,j,k
- Then, make a backward pass thru the network, starting at the top layer
 - Top layer is base case; compute

$$\delta_{t,1} = \frac{\partial L}{\partial y_{t,1}} \frac{\partial y_{t,1}}{\partial v_{t,1}} = f(v_{t,1})(1 - f(v_{t,1}))(y_{t,1} - y^*)$$

- And then

$$\frac{\partial L}{\partial w_{t,j,1}} = \delta_{t,1} y_{t-1,j}$$

DP for Back-propagation: Other Layers

- Then at every other layer, want $\frac{\partial L}{\partial w_{i,j,k}}$, for each j, k pair
- To do this:
 - At layer i , for a given j, k pair
 - First compute $\delta_{i,k} = (\sum_{k'} \delta_{i+1,k'} w_{i+1,k,k'}) (f(v_{i,k})(1 - f(v_{i,k})))$; save this value
 - Next compute $\frac{\partial L}{\partial w_{i,j,k}} = \delta_{i,k} y_{i-1,j}$
- Keep recursing until you have each $\frac{\partial L}{\partial w_{2,j,k}}$
- All of the $\frac{\partial L}{\partial w_{i,j,k}}$ values together then constitute $\nabla L(W)$
- Use in an iteration of GD

How To Implement Learning?

- This math has been known for many years
- Once you understand it, is quite simple
- So would be easy to implement learning directly in Python
- But applies to particular narrow case:
 - Fully-connected network
 - Logistic activation
 - Feed-forward
 - L2 loss
 - One output neuron
 - One data item

What If Your Network Is Different?

- Some changes are easy:
 - Different loss: only changes $\frac{\partial L}{\partial y_{t,1}}$
 - Multiple data points in a batch: compute $\frac{\partial L}{\partial w_{i,j,k}}$ wrt each, and then add them up (can be made efficient with vector/matrix ops)
 - Different activation: only changes $\frac{\partial y_{i,j}}{\partial v_{i,j}}$
- But some are hard...
- What about adding convolutional layers?
- Pooling layers?
- Suddenly a lot of error-prone math and code needs to be written

So In Practice...

- ...automatic differentiation tools are used
- This is a large part of TensorFlow
- Idea:
 - You describe your network in a high level language
 - Maybe just fit together layers
 - And the system generates the learning algorithm for you
 - Automatically figuring out necessary partial derivatives

- Long history in CS (compilers in particular)
- Actually quite simple
 - Idea: don't generate code to actually execute math ops
 - Rather, they generate code to differentiate wrt that math operation
 - Example: user writes code $f(x, y, z) = exp_1 \times exp_2$
 - Wants partial derivative of f wrt x
 - Compiler won't generate code for $exp_1 \times exp_2$
 - Rather, compiler will generate code $exp_1 \frac{\partial exp_2}{\partial x} + exp_2 \frac{\partial exp_1}{\partial x}$
 - Using the chain rule

Auto Differentiation Example

- I write code to multiply two values
- The program overloads the multiplication operation
- And generates code that computes the partial derivative of the operation

Good/Bad of this Approach

■ Good

- Good: much less error prone
- Good: very high programmer/data scientist productivity
- Good: automatically generate algorithms that use hardware well
- There can be an optimization step
- Which provides an abstract representation of the partial derivative

■ Bad

- Bad: challenging to debug, since algorithm generated by compiler may look nothing like original code
- Bad: algorithm will not be as efficient as one written by hard-core expert

Huh?

- You could spend some time and implement back propagation
- Then you could debug it, since it's your code
- Instead, you have to debug generated code
- You build (e.g. in TensorFlow) a computation graph
- And ask the system to differentiate it for you
- You get a new compute graph
- If it doesn't work
- it's much harder to debug the automatically generated code

Questions?