Tools & Models for Data Science

Neural Networks (2): Learning

Chris Jermaine & Risa Myers

Rice University

What Do We Need to Learn?

- What is our optimization problem?
- We have
  - Labeled data
  - Network architecture
- How do we “fit” or “learn” our NN to get good predictions/classifications on data **SIMILAR** to our training data?

What Do We Mean by “Fit” and “Learn”?

- This is just learning the weights
  - Such that we minimize the prediction/classification error
  - aka a loss / objective function
- If your new data is not **SIMILAR** (aka from the same population) as your training data, the model will not perform well

Our Optimization Problem

$$\underset{\{w_i, b_i\}}{\text{Min}} L\big(NN(x_n), y_n\big)$$

where

$$NN(x_n) \sim \{w_n, b_n\}$$

and

$$i = \text{layer and } n = \text{Training data}\{x_n, y_n\}$$

- This is an optimization problem (learning the weights)

? How do we solve optimization problems?

Solving Optimization Problems

? How do we solve optimization problems?

- ... with gradient descent

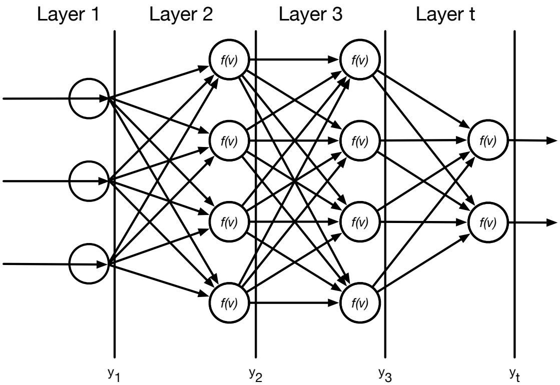
Learning Is Accomplished Via Gradient Descent

- 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: $W = W_1, W_2, ...$
  - "Learning" involves tweaking values of those matrices
  - ... to minimize a loss function
- Prior to back-propagation NN viewed as a computational model
- After: viewed as a way to fit a function to a dataset

Learning Is Accomplished Via Gradient Descent

- NN: viewed as a way to fit a function to a dataset
- Neuron functions are fixed
- Weights are learned

NN Variables



$[y_{1_1} \quad y_{1_2} \quad y_{1_3}]$ $[y_{21} \quad y_{22} \quad y_{23} \quad y_{24}]$ $[y_{31} \quad y_{32} \quad y_{33} \quad y_{34}]$ $[y_{t1} \quad y_{t2}]$

W_2
$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix}$$

W_3
$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{bmatrix}$$

W_4
$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{bmatrix}$$

- $w_{i,j,k}$:
- $i =$ layer
- $j =$ input neuron
- $k =$ output neuron
- W is a non-uniform tensor
- Each layer's dimensions are based on the inputs to and outputs from that layer

GD for NNs

- 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, for some number of training epochs, N and some small value ε

numIter = 0;
Make a non-stupid guess for each W_i;
repeat {
  $W \leftarrow W - \lambda \nabla L(W)$;
  numIter++;
} **while** (change in loss $> \varepsilon$ or numIters < N)

Non-stupid Guess for a Weight

```
numIter = 0;
Make a non-stupid guess for each W_i;
repeat {
   W ← W − λ∇L(W);
   numIter++;
} while (change in loss > ε
                or numIters < N)
```

- Don't make them all the same
- Don't make them too different in magnitude
- Something random is a good idea
- Choose a distribution! (based on ...? active field of research)
- Distribution can be based on network structure
- One choice is standard N(0,1)
- Another is a truncated normal distribution (Glorot Normal)

Choosing ε

```
numIter = 0;
Make a non-stupid guess for each W_i;
repeat {
   W ← W − λ∇L(W);
   numIter++;
} while (change in loss > ε
                    or numIters < N)
```

- As ε gets smaller, you will run more iterations
- Depends on your application's tolerance for error
- "Coarser" problems will have bigger εs
- Scientific computing: ε is as small as possible
- Facial recognition: 0.1 or 0.01

Choosing N

```
numIter = 0;
Make a non-stupid guess for each W_i;
repeat {
  W ← W − λ∇L(W);
  numIter++;
} while (change in loss > ε
                or numIters < N)
```

- How much time do you have?
- Are you making progress - is your Loss function improving?

Choosing a Loss Function

1. Use the arbitrary accuracy metric given based on the problem
   - E.g. Jaccard or Dice similarity for image segmentation
2. Use a matching loss function if the training model follows a distribution
   - Normal data: L_2 loss
   - Multinomial data: Categorical cross entropy
   - Binomial data: Binomial cross entropy
3. Recall GLM: LLH form matches the data - Mean Squared Error $\left(\frac{L_2}{n}\right)$

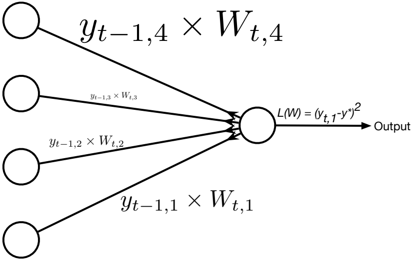
Big Picture

$$W \leftarrow W - \lambda \nabla L(W)$$

1. Make a non-stupid guess for the weights (initialize W)
2. Compute the value of the Loss function ($L(W)$)
3. Redistribute the error from the Loss function among the weights
   $W \leftarrow W - \lambda \nabla L(W)$

- 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



- Neuron 4 has a large contribution to the error
- Neuron 3 has a small contribution to the error
- t is the top layer

Computing the Gradient

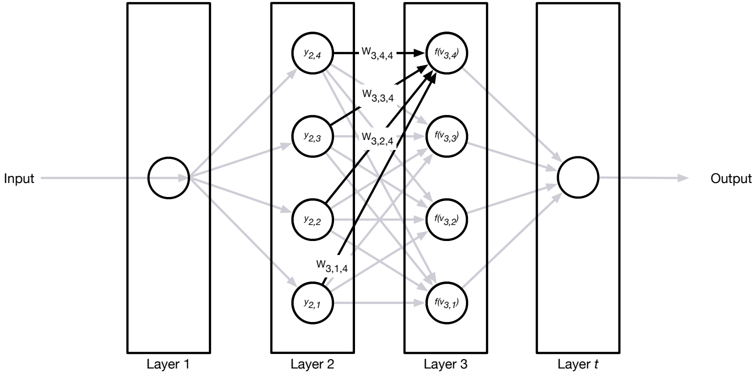
- Need to be able to differentiate L wrt each weight in the entire network!
- Done by applying the chain rule:

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

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

Computing the Neuron Input

$$v_{3,4} = \sum_{j'} w_{3,j',4} y_{2,j'}$$



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

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 $v_{i,k}$
  - where i = layer, j = input neuron index, k = output neuron index
- 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

Step 1: Differentiating Neuron Output

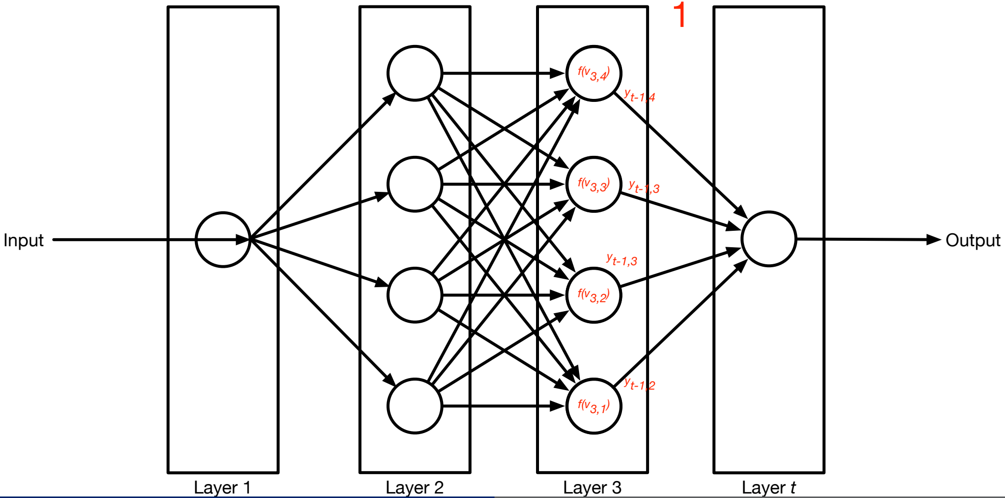
- We start with neuron output
- That is,

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

Step 1: Computing the Gradient - Neuron Output

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

1



Step 1: Differentiating Neuron Output

- Assume activation function is the logistic function

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

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

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

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

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

- Which is surprisingly nice for Logistic Regression

Step 2: 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:

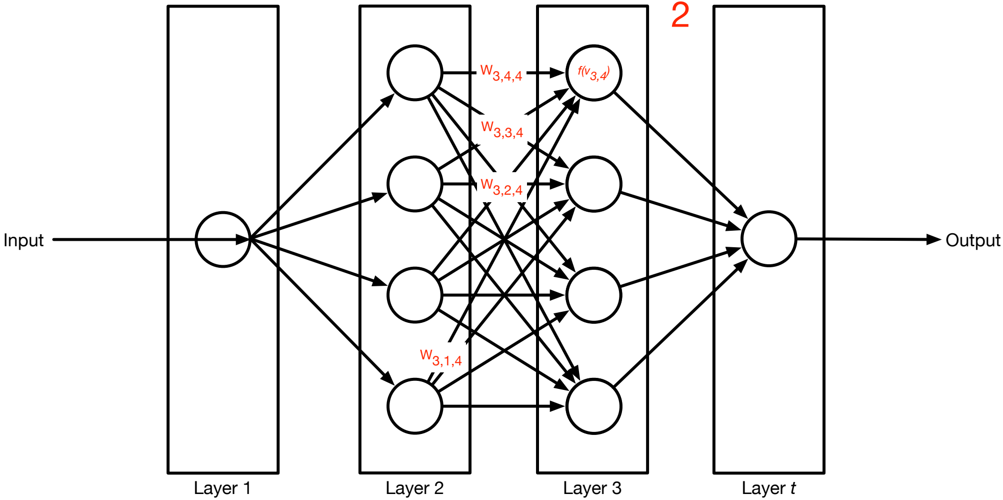
$$\frac{\partial v_{i,k}}{\partial w_{i,j,k}} = \frac{\partial}{\partial w_{i,j,k}} \sum_{j'} w_{i,j',k} y_{i-1,j'}$$

- 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

Step 2: Computing the Gradient - Neuron Input

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

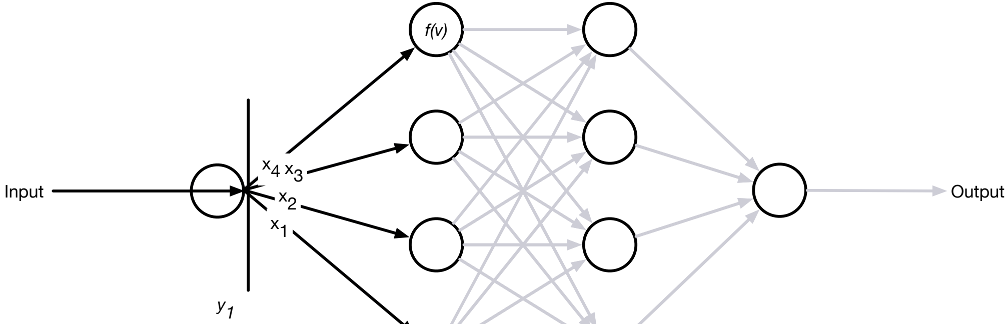
2



Step 2: 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 jth entry in the input vector

Step 2: Computing the Gradient - Neuron Input



Step 3: Differentiating the Loss

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

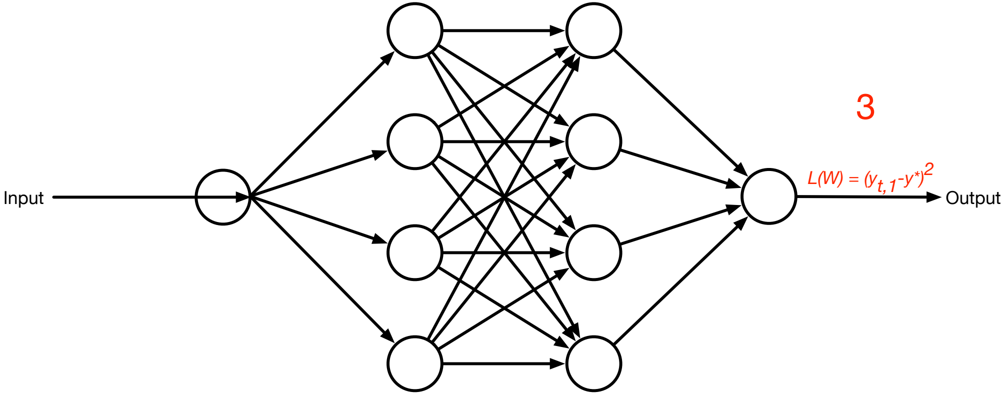
$$\frac{\partial L}{\partial y_{i,k}}$$

- 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

Step 3: Computing the Gradient - Loss for Output Layer



Step 3: Differentiating the Loss

Recall

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

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

- 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?

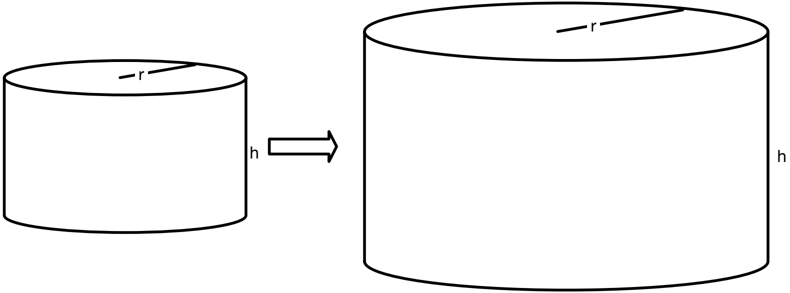
Sidebar: the Total Derivative

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

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

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

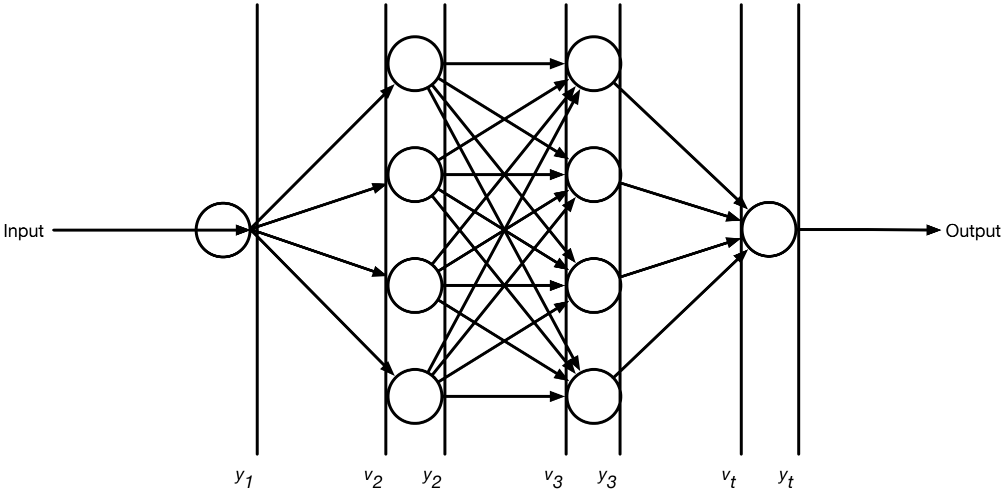
Step 3: Differentiating Loss for Hidden Layers

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

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

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

Step 3: Computing the Gradient - Loss for Hidden Layers



Step 3: 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:

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

  - Further, note that, by chain rule:

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

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

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

Recursion!!

- 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:

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

Breaking down the Transition

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

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

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

$$= \delta_{i,k} \left(y_{i-1,j} \right)$$

Recursion!!

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

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

- 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 $i-1$

Dynamic Programming Algorithm

- 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 $\delta_{i,k} = \left(\sum_{k'} \delta_{i+1,k'} w_{i+1,k,k'}\right) \left(f(v_{i,k})(1 - f(v_{i,k}))\right)$; 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

What If Your Network Is not a Fully Connected FF Network?

- Some changes are easy:
  - Different loss: only changes $\frac{\partial L}{\partial y_{t,1}}$
  - 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

Auto Differentiation

- 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 care 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?