

# **CS224d**

## **Deep NLP**

### **Lecture 5:**

## **Project Information**

**+**

## **Neural Networks & Backprop**

**Richard Socher**  
**[richard@metamind.io](mailto:richard@metamind.io)**

# Overview Today:

- Organizational Stuff
- Project Tips
- From one-layer to multi layer neural network!
- Max-Margin loss and **backprop!**  
(This is the hardest lecture of the quarter)

# Announcement:

- 1% extra credit for Piazza participation!
- Hint for PSet1: Understand math and dimensionality, then add print statements, e.g.

```
28
29 def softmaxCostAndGradient(predicted, target, outputVectors, dataset):
30     """ Softmax cost function for word2vec models """
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55     ### YOUR CODE HERE
56
57     print "v_hat", predicted.shape
58     print "expected", target
59     print "U", outputVectors.shape
60
61     assert False
62
```

- Student survey sent out last night, please give us feedback to improve the class :)

# Class Project

- Most important (40%) and lasting result of the class
- PSet 3 a little easier to have more time
- Start early and clearly define your task and dataset
- Project types:
  1. Apply existing neural network model to a new task
  2. Implement a complex neural architecture
  3. Come up with a new neural network model
  4. Theory

# Class Project: Apply Existing NNets to Tasks

## 1. Define Task:

- Example: **Summarization**

## 2. Define Dataset

### 1. Search for academic datasets

- They already have baselines
- E.g.: Document Understanding Conference (DUC)

### 2. Define your own (harder, need more new baselines)

- If you're a graduate student: connect to your research
- Summarization, Wikipedia: Intro paragraph and rest of large article
- Be creative: Twitter, Blogs, News

# Class Project: Apply Existing NNets to Tasks

## 3. Define your metric

- Search online for well established metrics on this task
- Summarization: Rouge (Recall-Oriented Understudy for Gisting Evaluation) which defines n-gram overlap to human summaries

## 4. Split your dataset!

- Train/Dev/Test
- Academic dataset often come pre-split
- Don't look at the test split until ~1 week before deadline!

# Class Project: Apply Existing NNets to Tasks

## 5. Establish a baseline

- Implement the simplest model (often logistic regression on unigrams and bigrams) first
- Compute metrics on train AND dev
- Analyze errors
- If metrics are amazing and no errors: done, problem was too easy, restart :)

## 6. Implement existing neural net model

- Compute metric on train and dev
- Analyze output and errors
- Minimum bar for this class

# Class Project: Apply Existing NNets to Tasks

## 7. Always be close to your data!

- Visualize the dataset
- Collect summary statistics
- Look at errors
- Analyze how different hyperparameters affect performance

## 8. Try out different model variants

- Soon you will have more options
  - Word vector averaging model (neural bag of words)
  - Fixed window neural model
  - Recurrent neural network
  - Recursive neural network
  - Convolutional neural network



# Class Project: A New Model -- Advanced Option

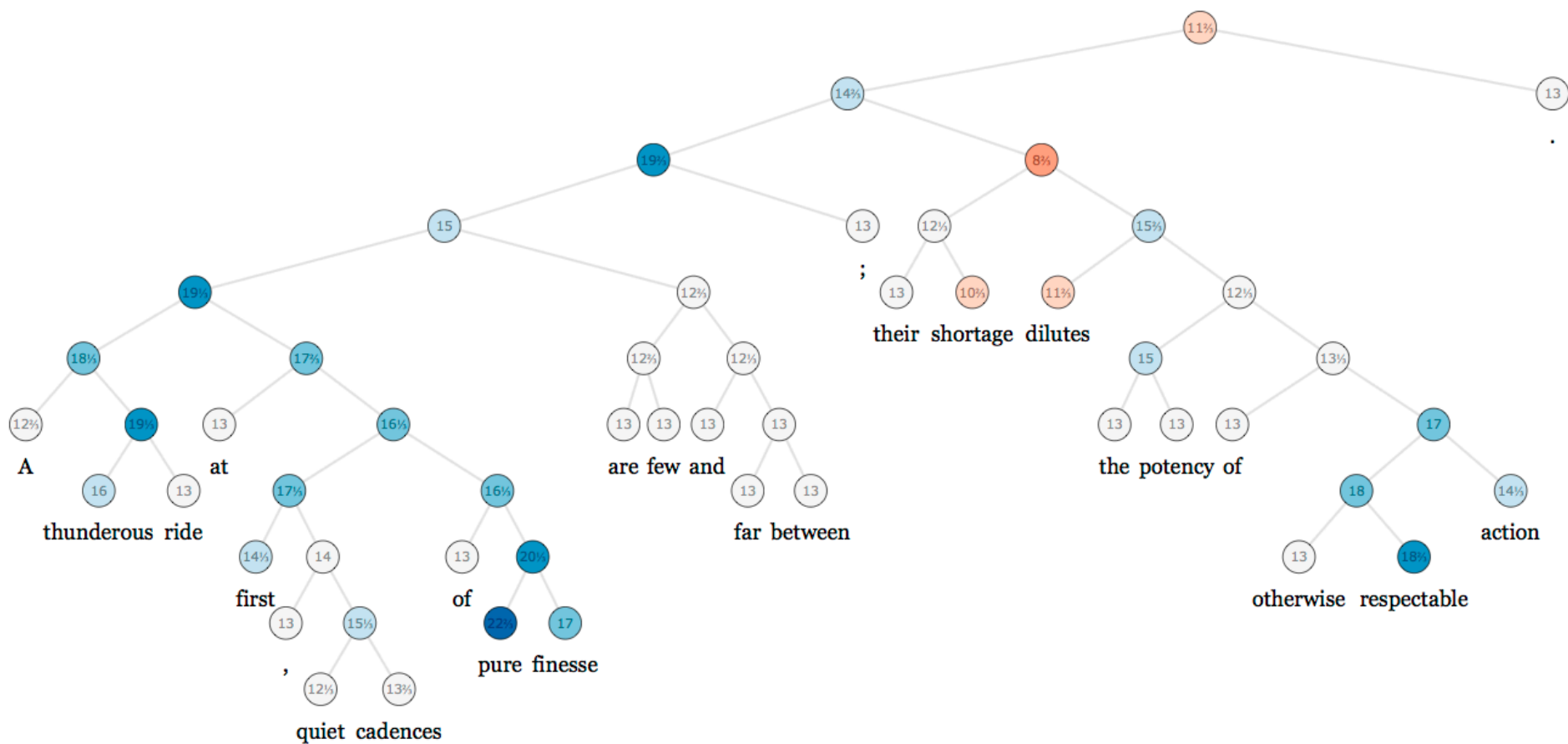
- Do all other steps first (Start early!)
- Gain intuition of why existing models are flawed
- Talk to other researchers, come to my office hours a lot
- Implement new models and iterate quickly over ideas
- Set up efficient experimental framework
- Build simpler new models first
- Example Summarization:
  - Average word vectors per paragraph, then greedy search
  - Implement language model or autoencoder (introduced later)
  - Stretch goal for potential paper: Generate summary!

# Project Ideas

- Summarization
- NER, like PSet 2 but with larger data  
Natural Language Processing (almost) from Scratch, Ronan Collobert, Jason Weston, Leon Bottou, Michael Karlen, Koray Kavukcuoglu, Pavel Kuksa, <http://arxiv.org/abs/1103.0398>
- Simple question answering, [A Neural Network for Factoid Question Answering over Paragraphs](#), Mohit Iyyer, Jordan Boyd-Graber, Leonardo Claudino, Richard Socher and Hal Daumé III (**EMNLP 2014**)
- Image to text mapping or generation,  
[Grounded Compositional Semantics for Finding and Describing Images with Sentences](#), Richard Socher, Andrej Karpathy, Quoc V. Le, Christopher D. Manning, Andrew Y. Ng. (**TACL 2014**)  
or  
Deep Visual-Semantic Alignments for Generating Image Descriptions, Andrej Karpathy, Li Fei-Fei
- Entity level sentiment
- Use DL to solve an NLP challenge on kaggle,  
Develop a scoring algorithm for student-written short-answer responses, <https://www.kaggle.com/c/asap-sas>

## Default project: sentiment classification

- Sentiment on movie reviews: <http://nlp.stanford.edu/sentiment/>
- Lots of deep learning baselines and methods have been tried



# A more powerful window classifier

- Revisiting
- $X_{\text{window}} = [x_{\text{museums}} \quad x_{\text{in}} \quad x_{\text{Paris}} \quad x_{\text{are}} \quad x_{\text{amazing}}]$
- Assume we want to classify whether the center word is a location or not

# A Single Layer Neural Network

- A single layer is a combination of a linear layer and a nonlinearity:

$$z = Wx + b$$

$$a = f(z)$$

- The neural activations  $a$  can then be used to compute some function
- For instance, an unnormalized score or a softmax probability we care about:

$$\text{score}(x) = U^T a \in \mathbb{R}$$

# Summary: Feed-forward Computation

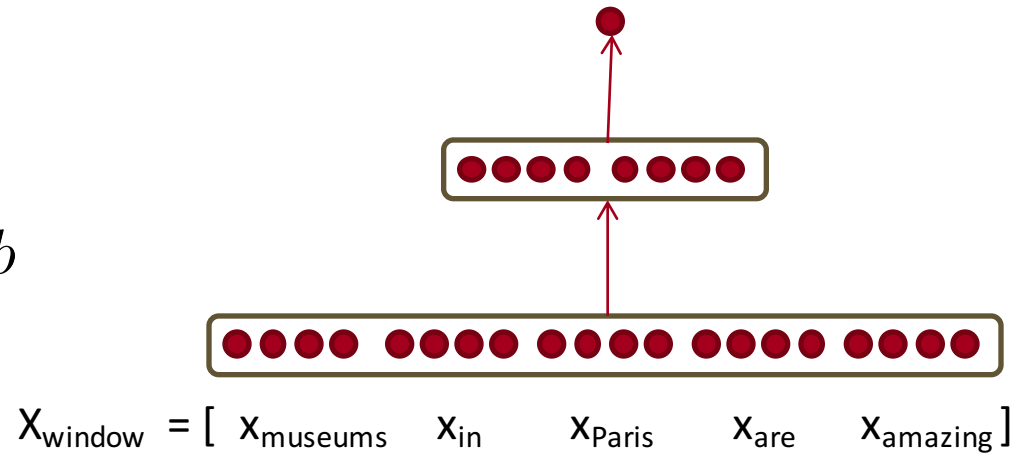
Computing a window's score with a 3-layer neural net:  $s = \text{score}(\text{museums in Paris are amazing})$

$$s = U^T f(Wx + b) \quad x \in \mathbb{R}^{20 \times 1}, W \in \mathbb{R}^{8 \times 20}, U \in \mathbb{R}^{8 \times 1}$$

$$s = U^T a$$

$$a = f(z)$$

$$z = Wx + b$$

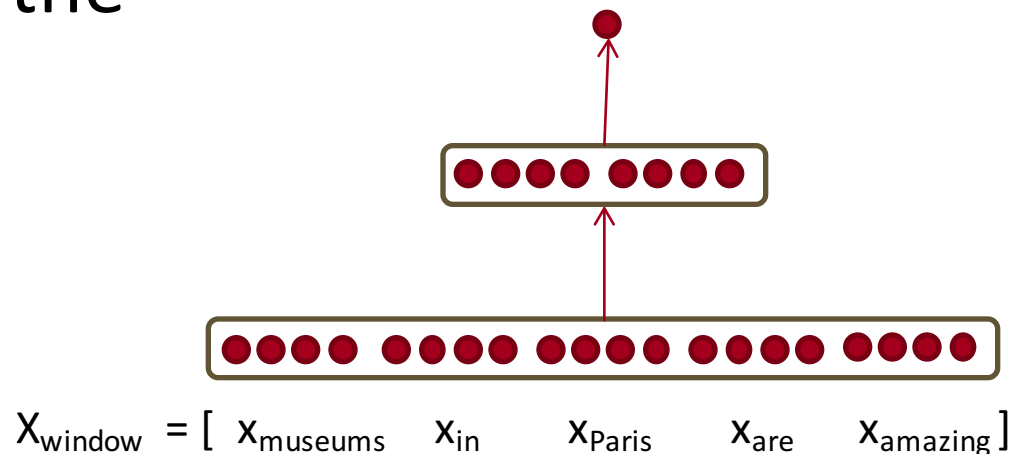


## Main intuition for extra layer

The layer learns non-linear interactions between the input word vectors.

Example:

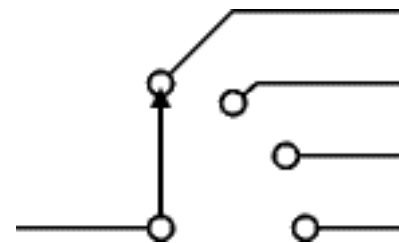
only if “*museums*” is first vector should it matter that “*in*” is in the second position



# Summary: Feed-forward Computation

- $s = \text{score}(\text{museums in Paris are amazing})$
- $s_c = \text{score}(\text{Not all museums in Paris})$
- Idea for training objective: make score of true window larger and corrupt window's score lower (until they're good enough): minimize

$$J = \max(0, 1 - s + s_c)$$



- This is continuous, can perform SGD



# Max-margin Objective function

- Objective for a single window:



$$J = \max(0, 1 - s + s_c)$$

- Each window with a location at its center should have a score +1 higher than any window without a location at its center
- xxx | ←   1   → |   ooo
- For full objective function: Sum over all training windows

# Training with Backpropagation

$$J = \max(0, 1 - s + s_c)$$

$$s = U^T f(Wx + b)$$

$$s_c = U^T f(Wx_c + b)$$

Assuming cost  $J$  is  $> 0$ ,  
compute the derivatives of  $s$  and  $s_c$  wrt all the  
involved variables:  $U, W, b, x$

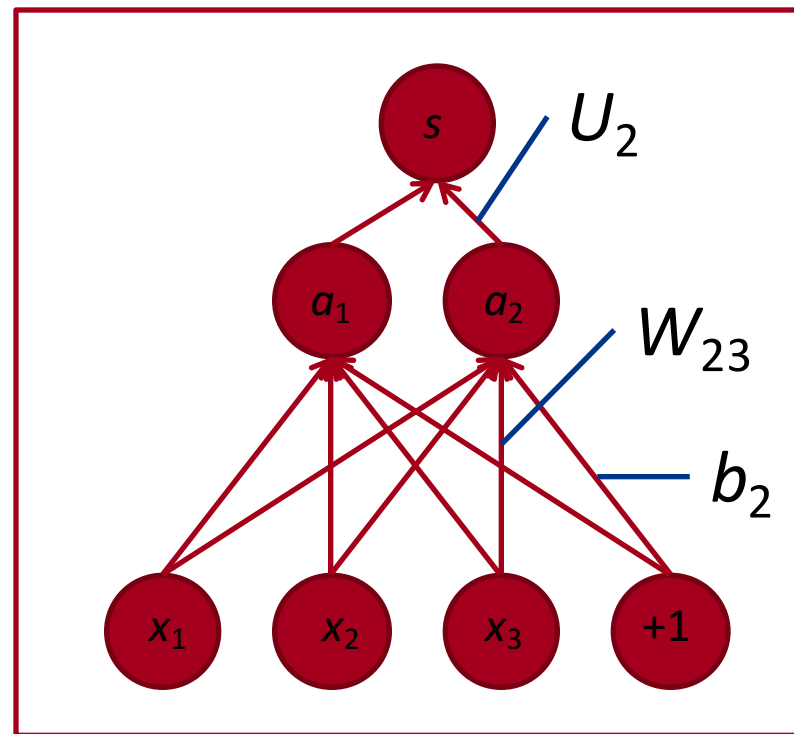
$$\frac{\partial s}{\partial U} = \frac{\partial}{\partial U} U^T a \qquad \frac{\partial s}{\partial U} = a$$

# Training with Backpropagation

- Let's consider the derivative of a single weight  $W_{ij}$

$$\frac{\partial s}{\partial W} = \frac{\partial}{\partial W} U^T a = \frac{\partial}{\partial W} U^T f(z) = \frac{\partial}{\partial W} U^T f(Wx + b)$$

- This only appears inside  $a_i$
- For example:  $W_{23}$  is only used to compute  $a_2$



# Training with Backpropagation

$$\frac{\partial s}{\partial W} = \frac{\partial}{\partial W} U^T a = \frac{\partial}{\partial W} U^T f(z) = \frac{\partial}{\partial W} U^T f(Wx + b)$$

Derivative of weight  $W_{ij}$ :

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \frac{\partial u}{\partial x}$$

$$\frac{\partial}{\partial W_{ij}} U^T a \rightarrow \frac{\partial}{\partial W_{ij}} U_i a_i$$

$$U_i \frac{\partial}{\partial W_{ij}} a_i = U_i \frac{\partial a_i}{\partial z_i} \frac{\partial z_i}{\partial W_{ij}}$$

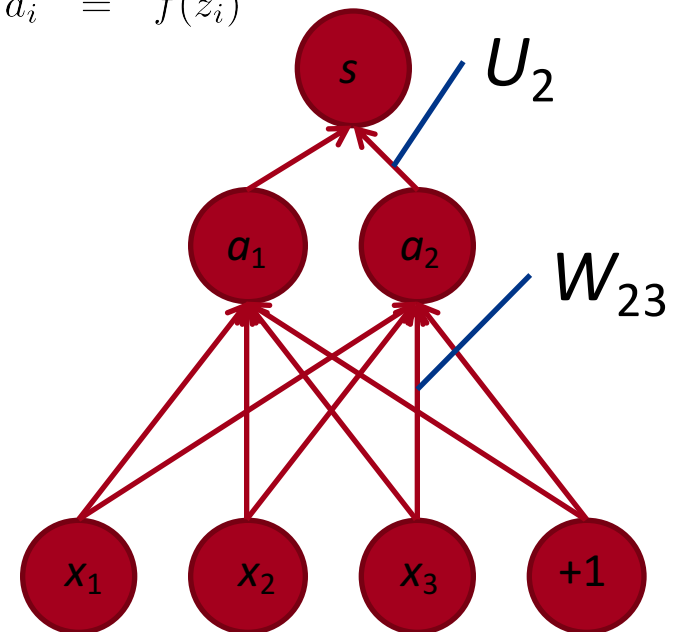
$$= U_i \frac{\partial f(z_i)}{\partial z_i} \frac{\partial z_i}{\partial W_{ij}}$$

$$= U_i f'(z_i) \frac{\partial z_i}{\partial W_{ij}}$$

$$= U_i f'(z_i) \frac{\partial W_i \cdot x + b_i}{\partial W_{ij}}$$

$$z_i = W_i \cdot x + b_i = \sum_{j=1}^3 W_{ij} x_j + b_i$$

$$a_i = f(z_i)$$



# Training with Backpropagation

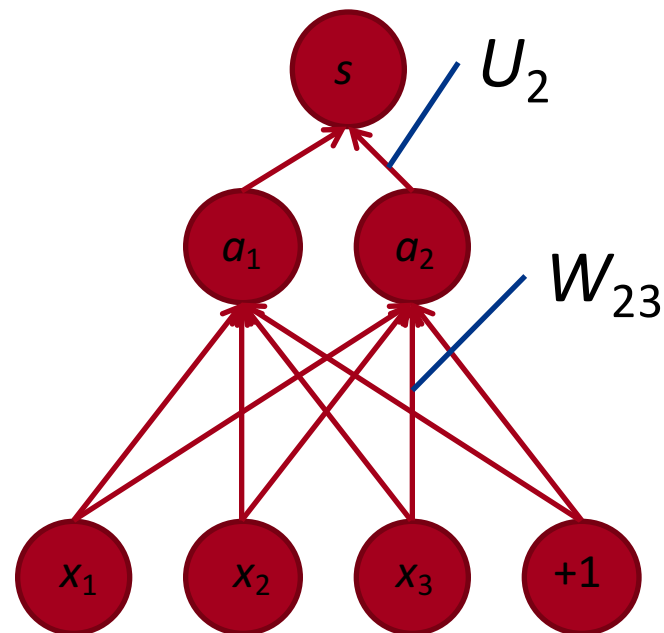
Derivative of single weight  $W_{ij}$  :

$$\begin{aligned} U_i \frac{\partial}{\partial W_{ij}} a_i &= U_i f'(z_i) \frac{\partial W_i \cdot x + b_i}{\partial W_{ij}} \\ &= U_i f'(z_i) \frac{\partial}{\partial W_{ij}} \sum_k W_{ik} x_k \\ &= \underbrace{U_i f'(z_i)}_{\delta_i} x_j \\ &= \underbrace{\delta_i}_{\text{Local error signal}} \underbrace{x_j}_{\text{Local input signal}} \end{aligned}$$

where  $f'(z) = f(z)(1 - f(z))$  for logistic  $f$

$$z_i = W_i \cdot x + b_i = \sum_{j=1}^3 W_{ij} x_j + b_i$$

$$a_i = f(z_i)$$



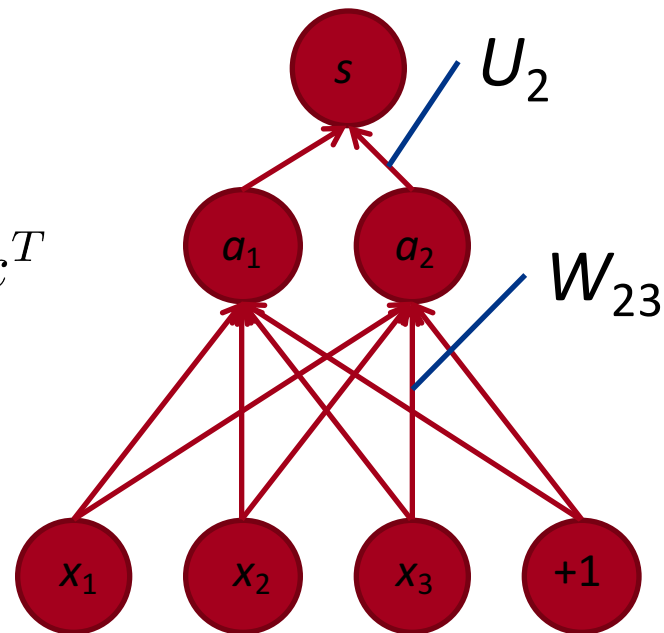
# Training with Backpropagation

- From single weight  $W_{ij}$  to full  $W$ :

$$\begin{aligned}\frac{\partial s}{\partial W_{ij}} &= \underbrace{U_i f'(z_i)}_{\delta_i} x_j \\ &= \delta_i x_j\end{aligned}$$

$$\begin{aligned}z_i &= W_i \cdot x + b_i = \sum_{j=1}^3 W_{ij} x_j + b_i \\ a_i &= f(z_i)\end{aligned}$$

- We want all combinations of  $i = 1, 2$  and  $j = 1, 2, 3 \rightarrow ?$
- Solution: Outer product:  $\frac{\partial J}{\partial W} = \delta x^T$   
where  $\delta \in \mathbb{R}^{2 \times 1}$  is the “responsibility” or error message coming from each activation  $a$

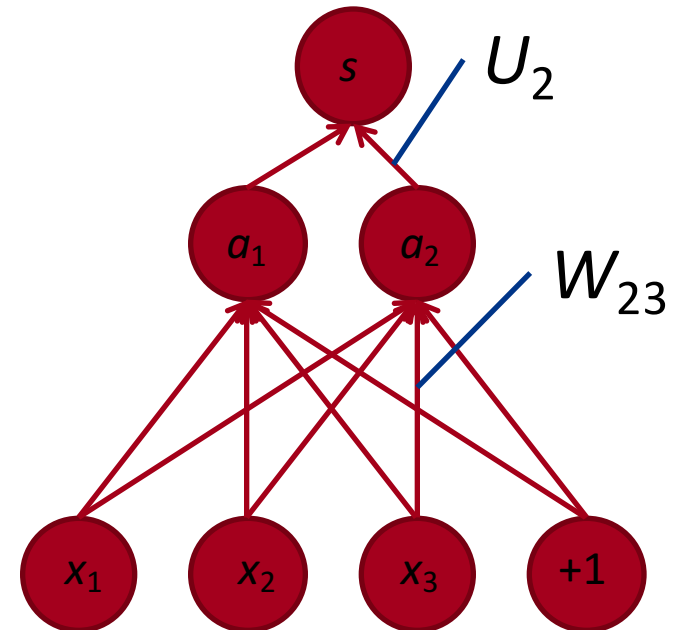


# Training with Backpropagation

- For biases  $b$ , we get:

$$\begin{aligned} & U_i \frac{\partial}{\partial b_i} a_i \\ = & U_i f'(z_i) \frac{\partial W_i \cdot x + b_i}{\partial b_i} \\ = & \delta_i \end{aligned}$$

$$\begin{aligned} z_i &= W_i \cdot x + b_i = \sum_{j=1}^3 W_{ij} x_j + b_i \\ a_i &= f(z_i) \end{aligned}$$



# Training with Backpropagation

That's almost backpropagation

It's simply taking derivatives and using the chain rule!

Remaining trick: we can **re-use** derivatives computed for higher layers in computing derivatives for lower layers!

Example: last derivatives of model, the word vectors in  $x$



# Training with Backpropagation

- Take derivative of score with respect to single element of word vector
- Now, we cannot just take into consideration one  $a_i$  because each  $x_j$  is connected to all the neurons above and hence  $x_j$  influences the overall score through all of these, hence:

$$\begin{aligned}
 \frac{\partial s}{\partial x_j} &= \sum_{i=1}^2 \frac{\partial s}{\partial a_i} \frac{\partial a_i}{\partial x_j} \\
 &= \sum_{i=1}^2 \frac{\partial U^T a}{\partial a_i} \frac{\partial a_i}{\partial x_j} \\
 &= \sum_{i=1}^2 U_i \frac{\partial f(W_{i \cdot} x + b)}{\partial x_j} \\
 &= \sum_{i=1}^2 \underbrace{U_i f'(W_{i \cdot} x + b)}_{\delta_i} \frac{\partial W_{i \cdot} x}{\partial x_j} \\
 &= \sum_{i=1}^2 \delta_i W_{ij} \\
 &= W_{\cdot j}^T \delta
 \end{aligned}$$

Re-used part of previous derivative 

# Training with Backpropagation

- With  $\frac{\partial s}{\partial x_j} = W_{\cdot j}^T \delta$ , what is the full gradient?  $\rightarrow$

$$\frac{\partial s}{\partial x} = W^T \delta$$

- Observations: The error message  $\pm$  that arrives at a hidden layer has the same dimensionality as that hidden layer

## Putting all gradients together:

- Remember: Full objective function for each window was:

$$J = \max(0, 1 - s + s_c) \quad \begin{aligned} s &= U^T f(Wx + b) \\ s_c &= U^T f(Wx_c + b) \end{aligned}$$

- For example: gradient for U:

$$\frac{\partial s}{\partial U} = 1\{1 - s + s_c > 0\} (-f(Wx + b) + f(Wx_c + b))$$

$$\frac{\partial s}{\partial U} = 1\{1 - s + s_c > 0\} (-a + a_c)$$

# Two layer neural nets and full backprop

- Let's look at a 2 layer neural network
- Same window definition for  $x$
- Same scoring function
- 2 hidden layers (carefully not superscripts now!)

$$x = z^{(1)} = a^{(1)}$$

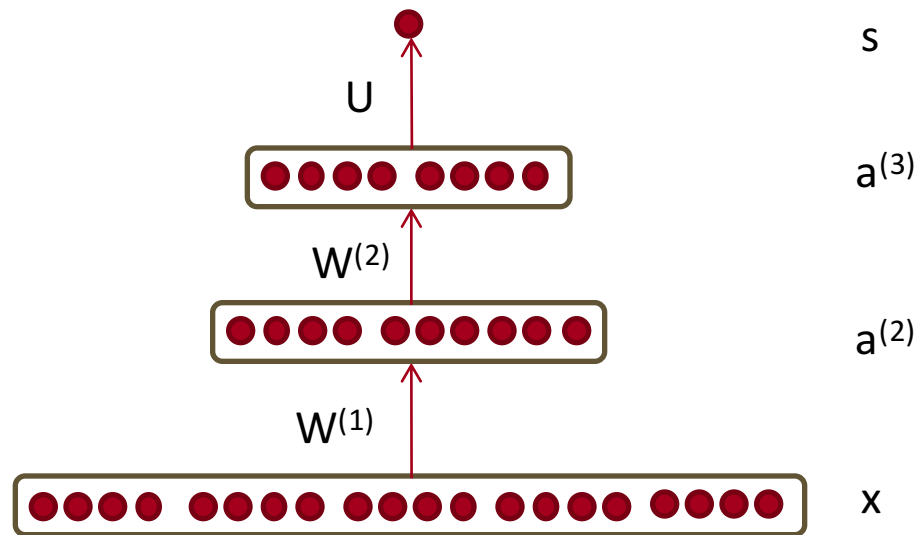
$$z^{(2)} = W^{(1)}x + b^{(1)}$$

$$a^{(2)} = f\left(z^{(2)}\right)$$

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$

$$a^{(3)} = f\left(z^{(3)}\right)$$

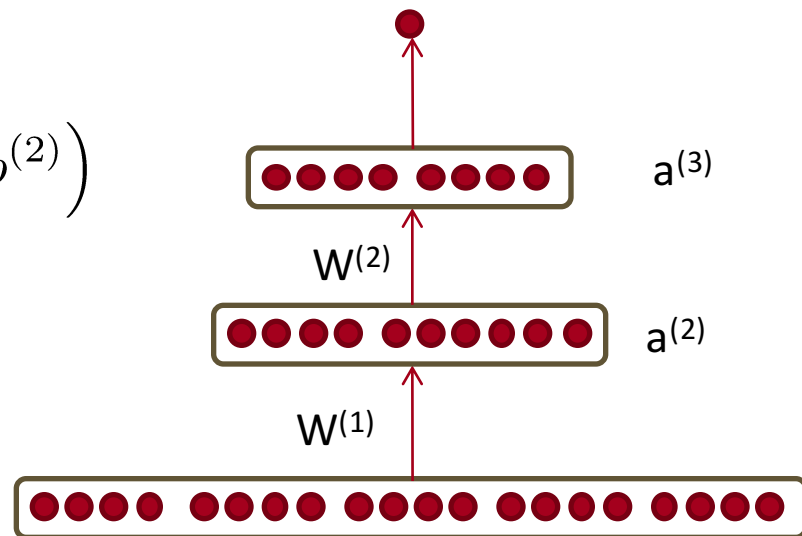
$$s = U^T a^{(3)}$$



# Two layer neural nets and full backprop

- Fully written out as one function:

$$\begin{aligned} s &= U^T f \left( W^{(2)} f \left( W^{(1)} x + b^{(1)} \right) + b^{(2)} \right) \\ &= U^T f \left( W^{(2)} a^{(2)} + b^{(2)} \right) \\ &= U^T a^{(3)} \end{aligned}$$



- Same derivation as before for  $W^{(2)}$  (now sitting on  $a^{(1)}$ )

$$\begin{aligned} \frac{\partial s}{\partial W_{ij}^{(1)}} &= \underbrace{U_i f'(z_i)}_{\delta_i} x_j \\ \frac{\partial s}{\partial W_{ij}^{(2)}} &= \underbrace{U_i f'(z_i^{(3)})}_{\delta_i^{(3)}} a_j^{(2)} \end{aligned}$$

# Two layer neural nets and full backprop

- Same derivation as before for top  $W^{(2)}$  :
 
$$\frac{\partial s}{\partial W_{ij}^{(2)}} = \underbrace{U_i f' \left( z_i^{(3)} \right)}_{\delta_i^{(3)}} a_j^{(1)}$$

$$= \delta_i^{(3)} a_j^{(2)}$$

$$\begin{aligned} x &= z^{(1)} = a^{(1)} \\ z^{(2)} &= W^{(1)}x + b^{(1)} \\ a^{(2)} &= f \left( z^{(2)} \right) \\ z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\ a^{(3)} &= f \left( z^{(3)} \right) \\ s &= U^T a^{(3)} \end{aligned}$$

- In matrix notation:  $\frac{\partial s}{\partial W^{(2)}} = \delta^{(3)} a^{(2)T}$

where  $\delta^{(3)} = U \circ f' \left( z^{(3)} \right)$  and  $\pm$  is the element-wise product also called Hadamard product

- Last missing piece for understanding general backprop:  $\frac{\partial s}{\partial W^{(1)}}$

# Two layer neural nets and full backprop

- Last missing piece:  $\frac{\partial s}{\partial W^{(1)}}$
- What's the bottom layer's error message  $\delta^{(2)}$ ?
- Similar derivation to single layer model
- Main difference, we already have  $W^{(2)T} \delta^{(3)}$  and need to apply the chain rule again on  $f'(z^{(2)})$

$$\begin{aligned}x &= z^{(1)} = a^{(1)} \\z^{(2)} &= W^{(1)}x + b^{(1)} \\a^{(2)} &= f\left(z^{(2)}\right) \\z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\a^{(3)} &= f\left(z^{(3)}\right) \\s &= U^T a^{(3)}\end{aligned}$$

# Two layer neural nets and full backprop

- Chain rule for:  $s = U^T f \left( W^{(2)} f \left( W^{(1)} x + b^{(1)} \right) + b^{(2)} \right)$
- Get intuition by deriving  $\frac{\partial s}{\partial W^{(1)}}$  as if it was a scalar
- Intuitively, we have to sum over all the nodes coming into layer
- Putting it all together:  $\delta^{(2)} = \left( W^{(2)}{}^T \delta^{(3)} \right) \circ f' \left( z^{(2)} \right)$



# Two layer neural nets and full backprop

- Last missing piece:  $\frac{\partial s}{\partial W^{(1)}} = \delta^{(2)} x^T$
- In general for any matrix  $W^{(l)}$  at internal layer  $l$  and any error with regularization  $E_R$  all backprop in standard multilayer neural networks boils down to 2 equations:

$$\begin{aligned}
 x &= z^{(1)} = a^{(1)} \\
 z^{(2)} &= W^{(1)}x + b^{(1)} \\
 a^{(2)} &= f\left(z^{(2)}\right) \\
 z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\
 a^{(3)} &= f\left(z^{(3)}\right) \\
 s &= U^T a^{(3)}
 \end{aligned}$$

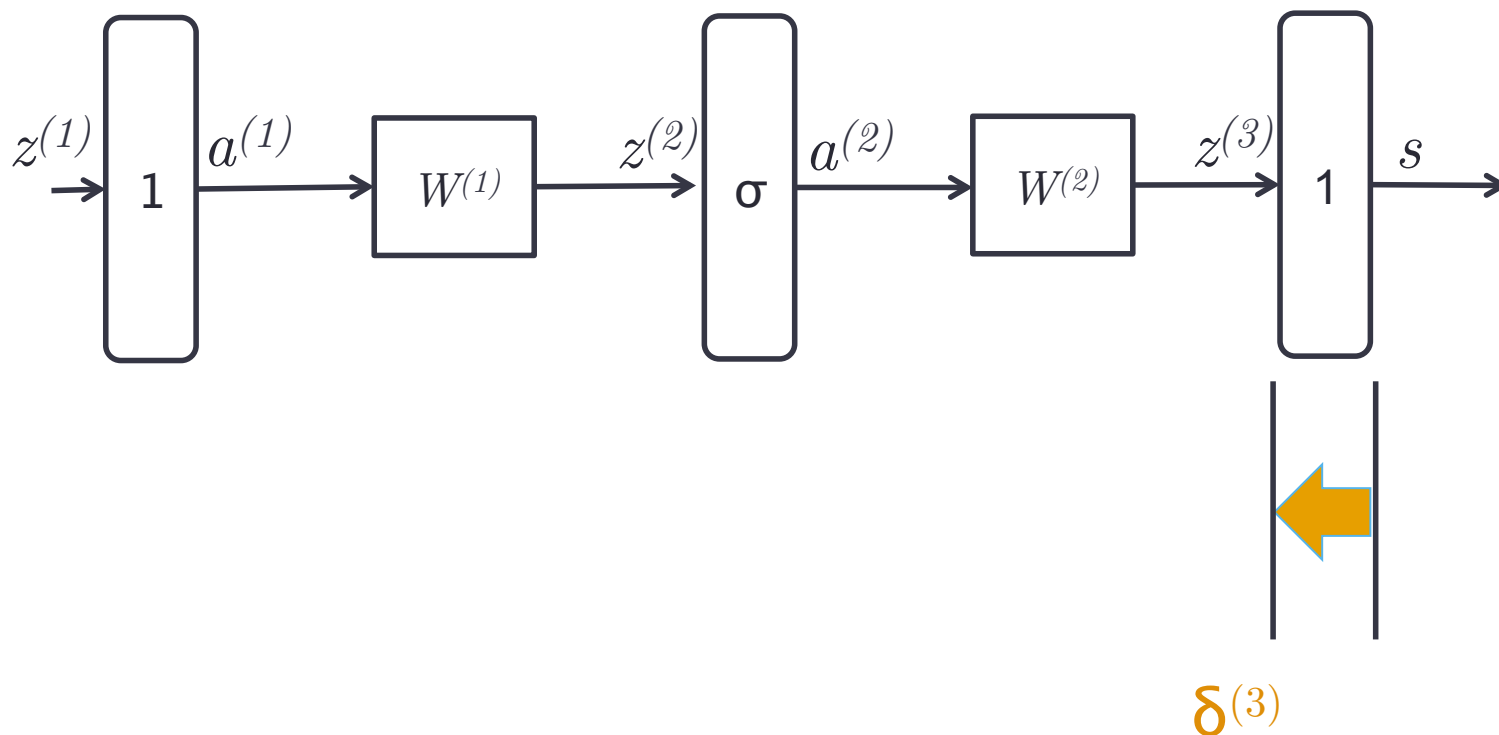
$$\delta^{(l)} = \left( (W^{(l)})^T \delta^{(l+1)} \right) \circ f'(z^{(l)}),$$

$$\frac{\partial}{\partial W^{(l)}} E_R = \delta^{(l+1)} (a^{(l)})^T + \lambda W^{(l)}$$

- Top and bottom layers have simpler  $\pm$

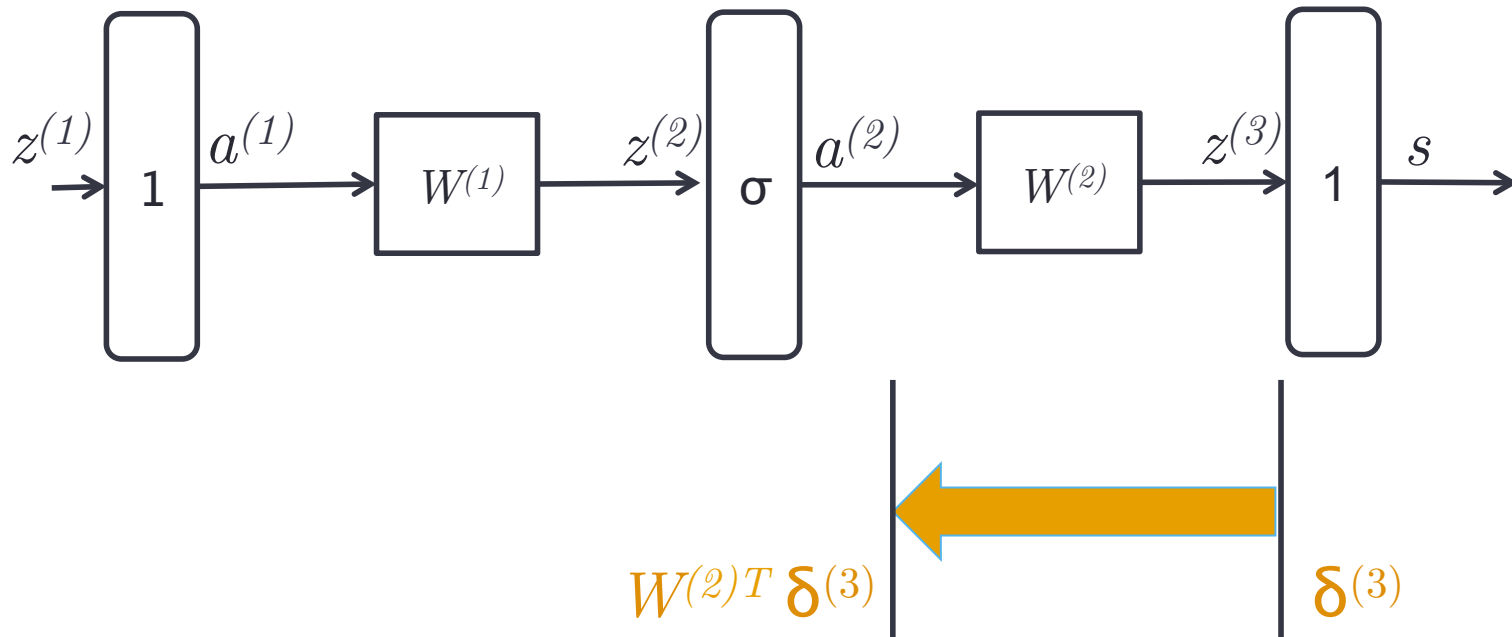
# Visualization of intuition

- Let's say we want  $\frac{\partial s}{\partial W^{(1)}} = \delta^{(2)} a^{(1)T}$  with previous layer and  $f = \frac{3}{4}$



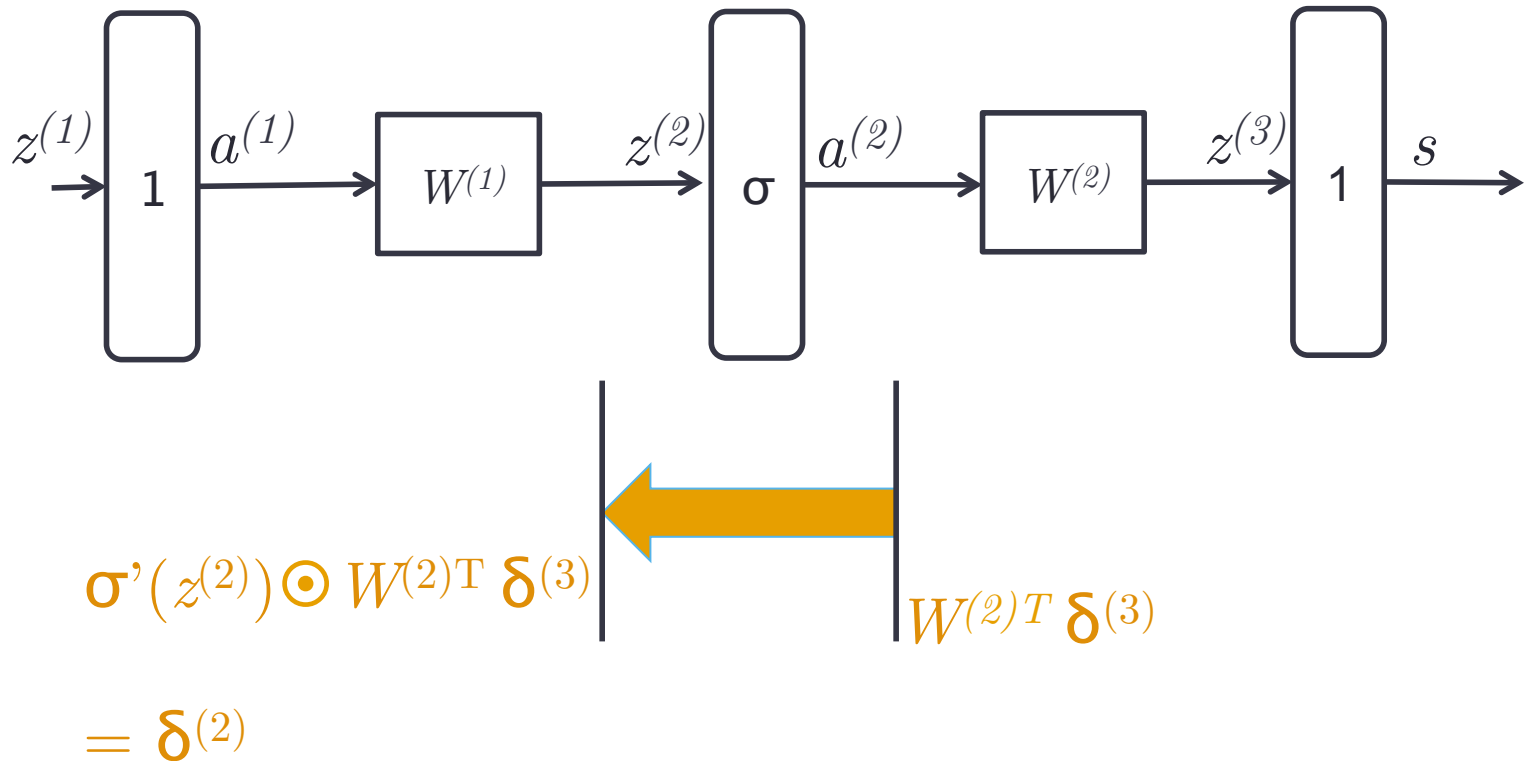
Gradient w.r.t  $W^{(2)} = \delta^{(3)} a^{(2)T}$

# Visualization of intuition



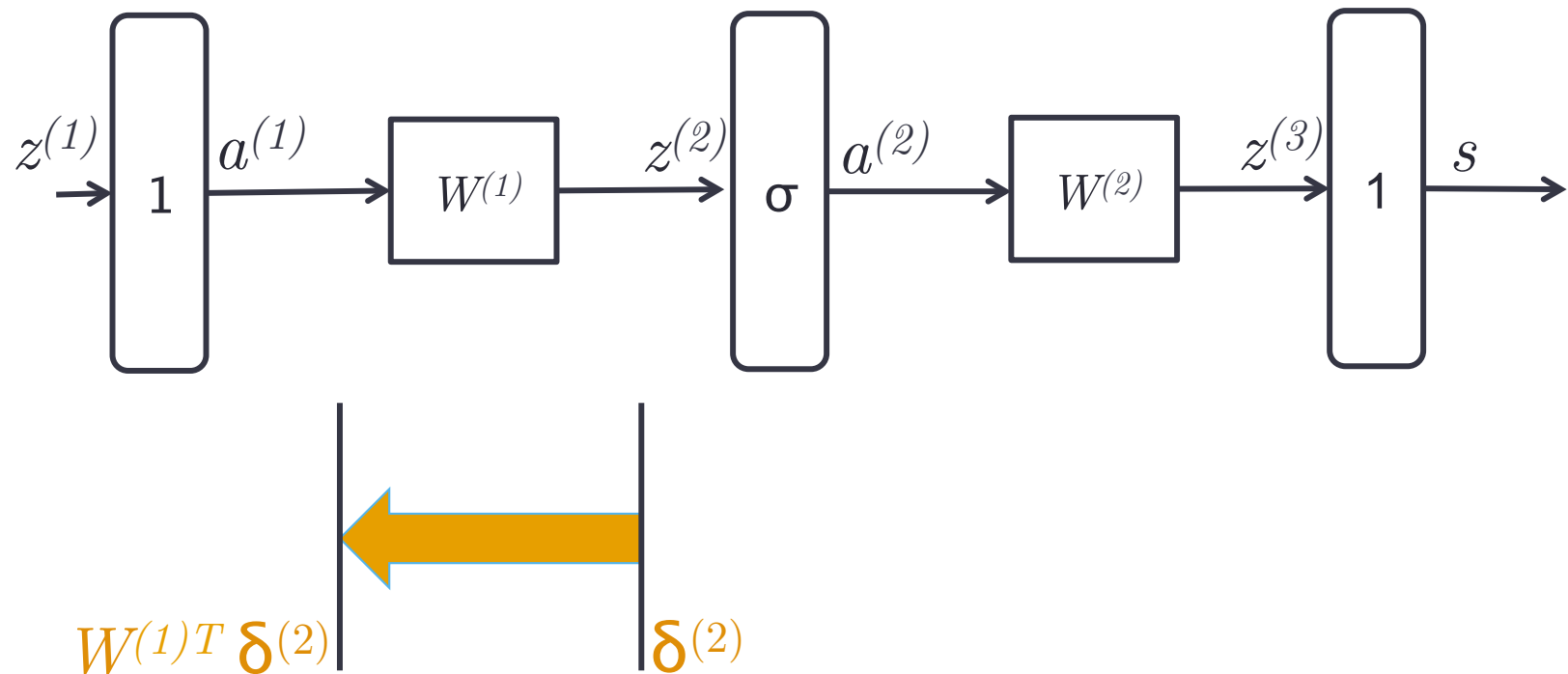
- Reusing the  $\delta^{(3)}$  for downstream updates.
- Moving error vector across affine transformation simply requires multiplication with the transpose of forward matrix
- Notice that the dimensions will line up perfectly too!

# Visualization of intuition



--Moving error vector across point-wise non-linearity requires point-wise multiplication with local gradient of the non-linearity

# Visualization of intuition



Gradient w.r.t  $W^{(1)} = \delta^{(2)} a^{(1)T}$

## Backpropagation (Another explanation)

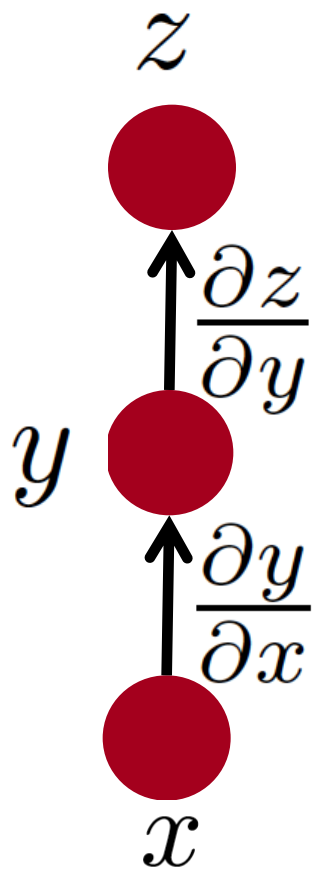
- Compute gradient of example-wise loss wrt parameters

- Simply applying the derivative chain rule wisely

$$z = f(y) \quad y = g(x) \quad \frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

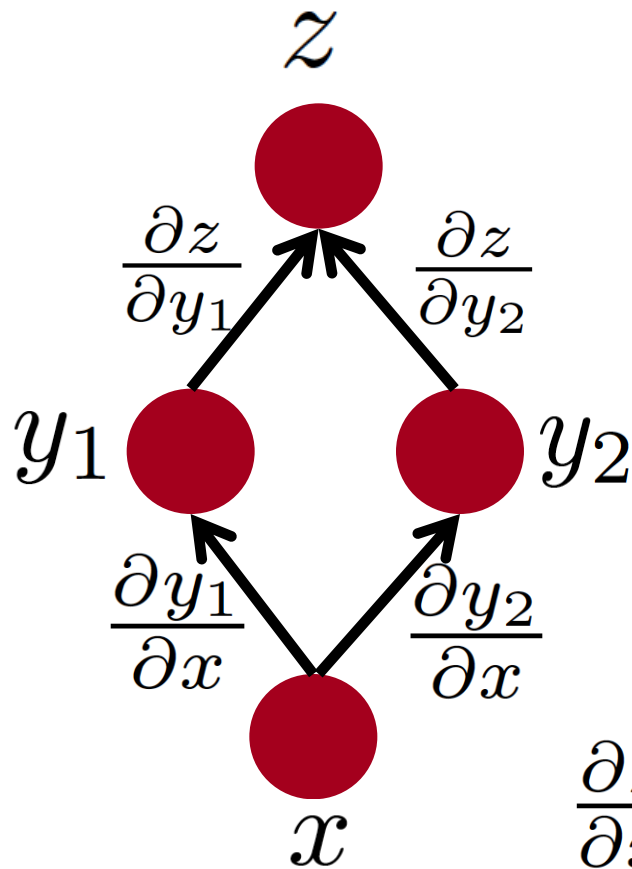
- If computing the loss(example, parameters) is  $O(n)$  computation, then so is computing the gradient

# Simple Chain Rule



$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

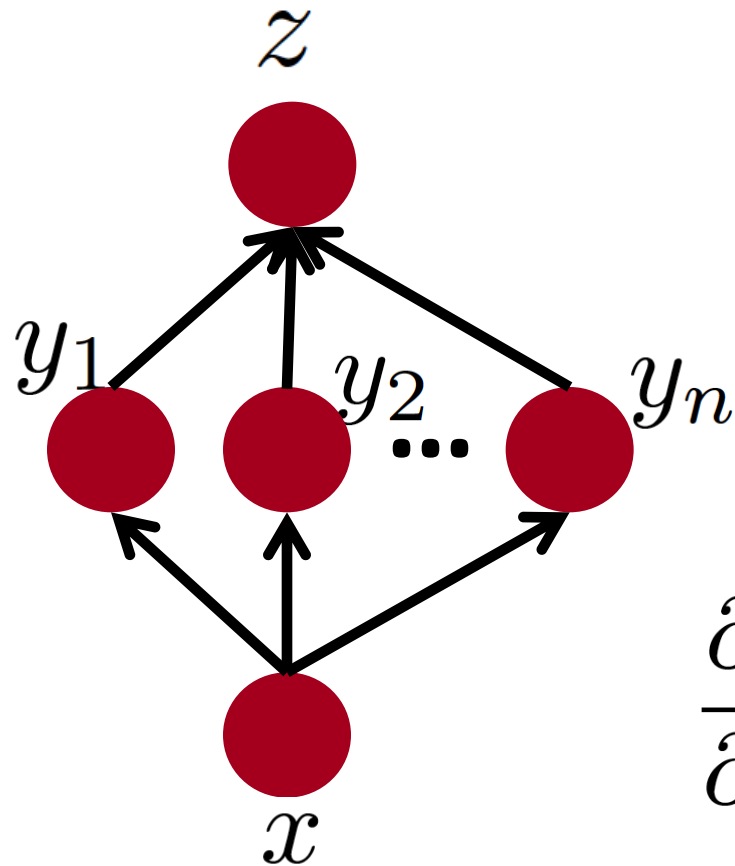
# Multiple Paths Chain Rule



$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1} \frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2} \frac{\partial y_2}{\partial x}$$

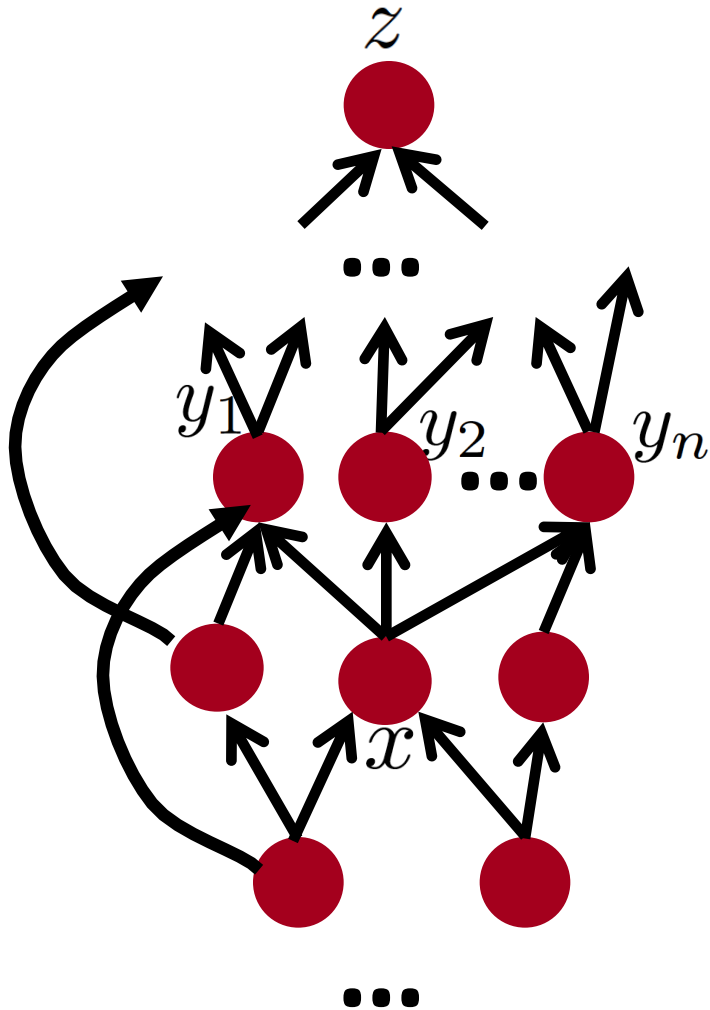


# Multiple Paths Chain Rule - General



$$\frac{\partial z}{\partial x} = \sum_{i=1}^n \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

# Chain Rule in Flow Graph

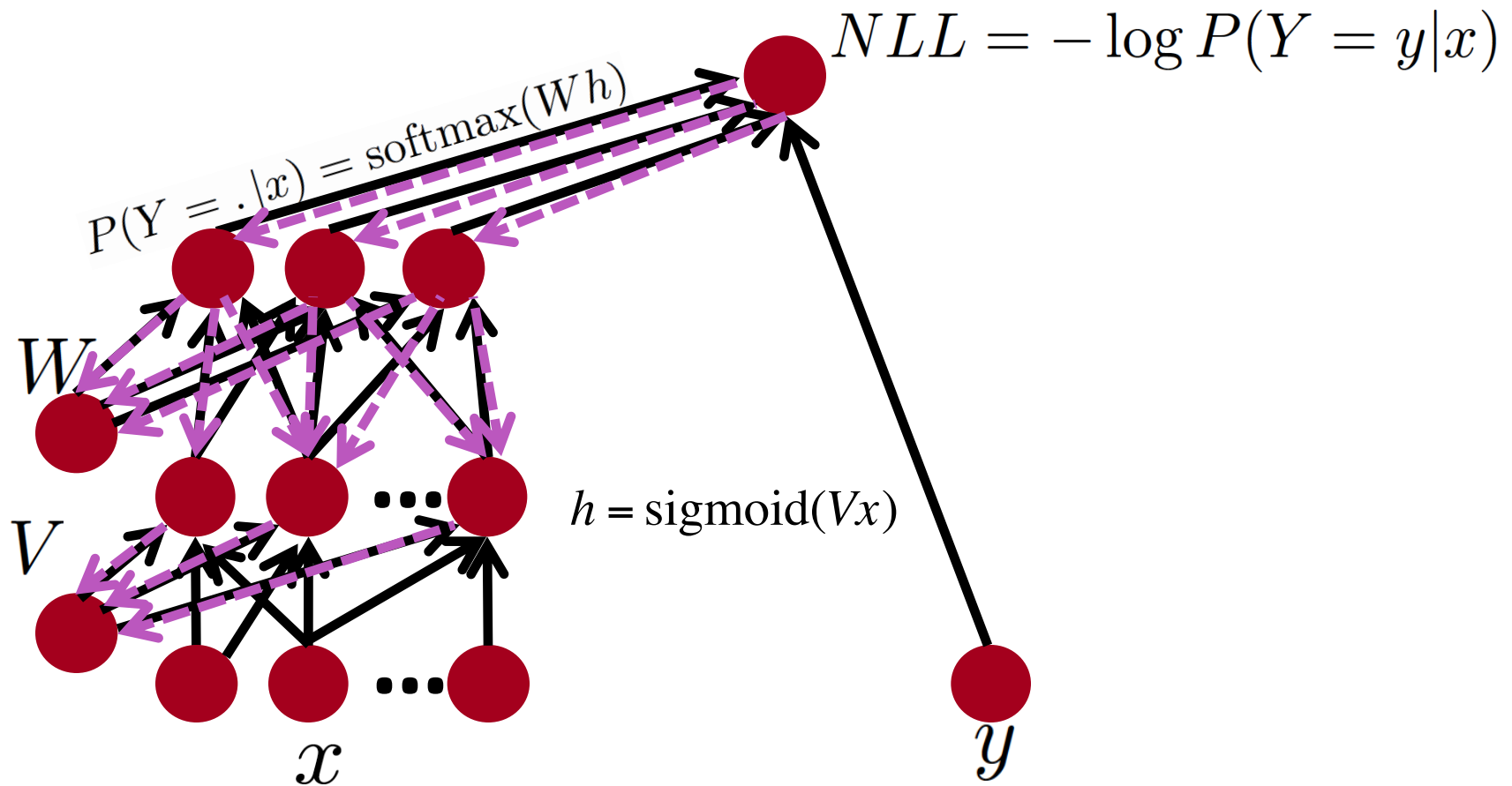


Flow graph: any directed acyclic graph  
node = computation result  
arc = computation dependency

$\{y_1, y_2, \dots, y_n\}$  = successors of  $x$

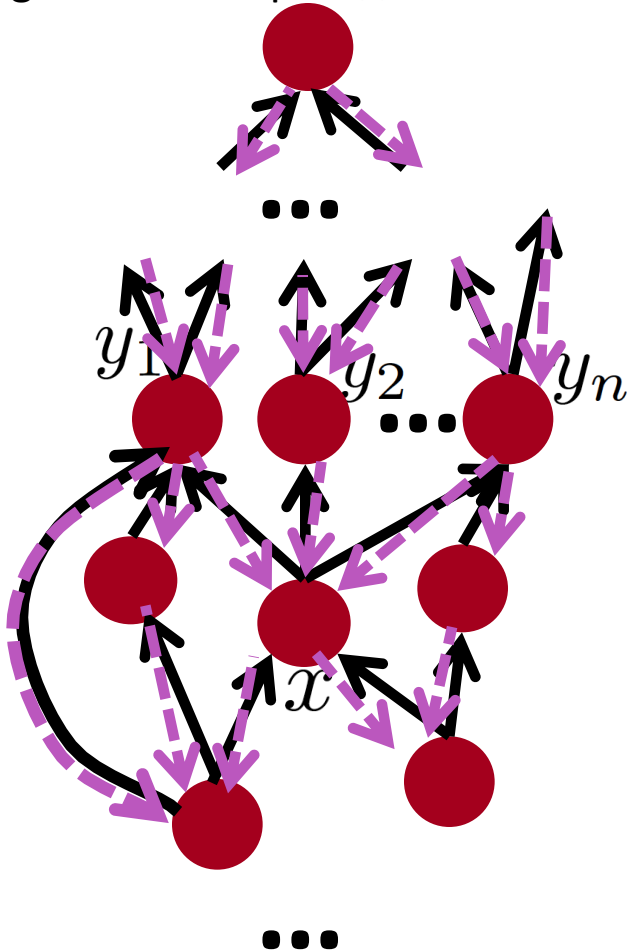
$$\frac{\partial z}{\partial x} = \sum_{i=1}^n \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

# Back-Prop in Multi-Layer Net



# Back-Prop in General Flow Graph

Single scalar output  $z$



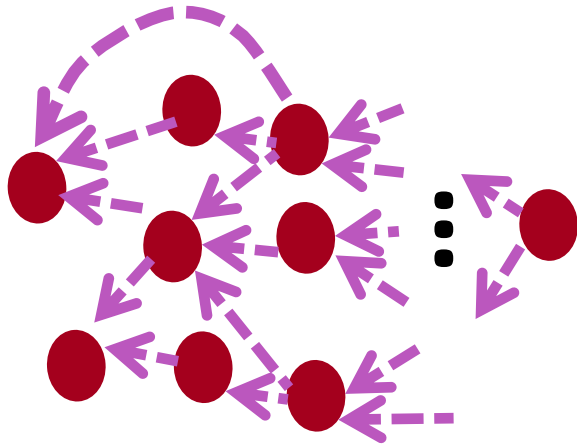
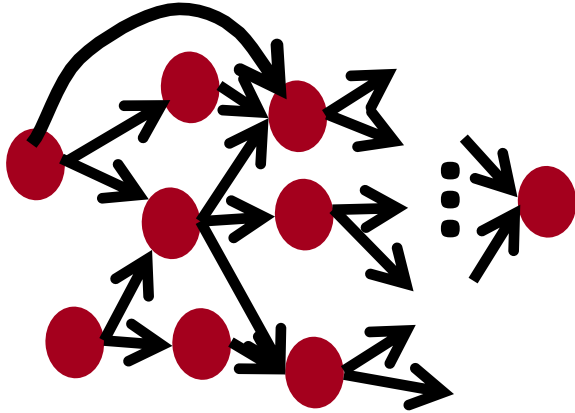
1. Fprop: visit nodes in topo-sort order
  - Compute value of node given predecessors
2. Bprop:
  - initialize output gradient = 1
  - visit nodes in reverse order:

Compute gradient wrt each node using gradient wrt successors

$\{y_1, y_2, \dots, y_n\} = \text{successors of } x$

$$\frac{\partial z}{\partial x} = \sum_{i=1}^n \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

# Automatic Differentiation



- The gradient computation can be **automatically inferred** from the symbolic expression of the fprop.
- Each node type needs to know how to compute its output and how to compute the gradient wrt its inputs given the gradient wrt its output.
- Easy and fast prototyping

# Summary

- Congrats!
- You survived the hardest part of this class.
- Everything else from now on is just more matrix multiplications and backprop :)
- Next up:
  - Recurrent Neural Networks