DeepLearningCourse

Lecture 6:
Word Window Classification

and Neural Networks

Valentin Malykh

Based on Stanford CS224d

Overview Today:

- General classification background
- Updating word vectors for classification
- Window classification & cross entropy error derivation tips
- A single layer neural network!
- Max-Margin loss and **backprop**
- RecurrentNeural Nets (introduction)

Classification setup and notation

- Generally we have a training dataset consisting of samples

$$\{x_i, y_i\}_{i=1}^{N}$$

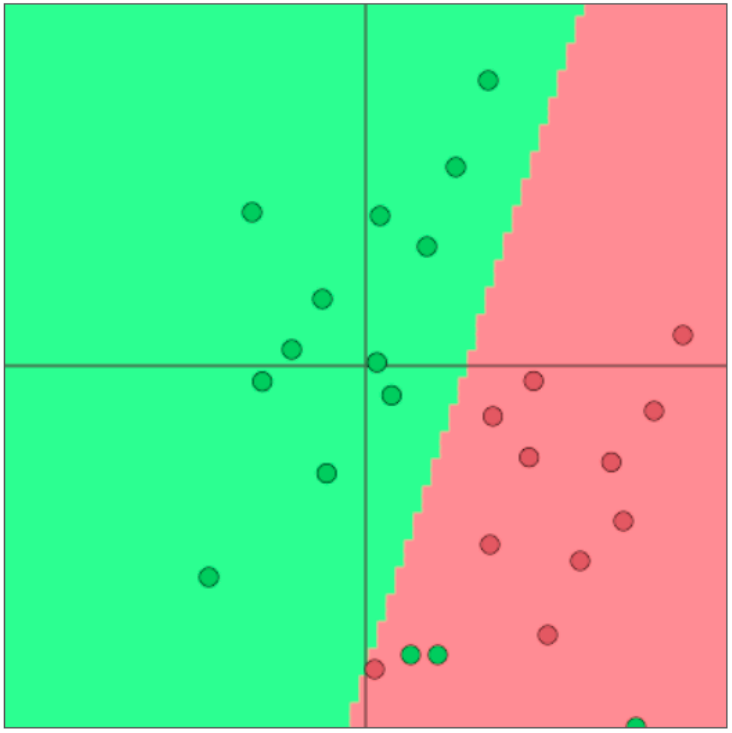
- x_i - inputs, e.g. words (indices or vectors!), context windows, sentences, documents, etc.

- y_i - labels we try to predict,
  - e.g. other words
  - class: sentiment, named entities, buy/sell decision,
  - later: multi-word sequences

Classification intuition

- Training data: $\{x_i, y_i\}^N_{i=1}$

- Simple illustration case:
  - Fixed 2d word vectors to classify
  - Using logistic regression
  - → linear decision boundary →



Visualizations with ConvNetJS by Karpathy!
http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

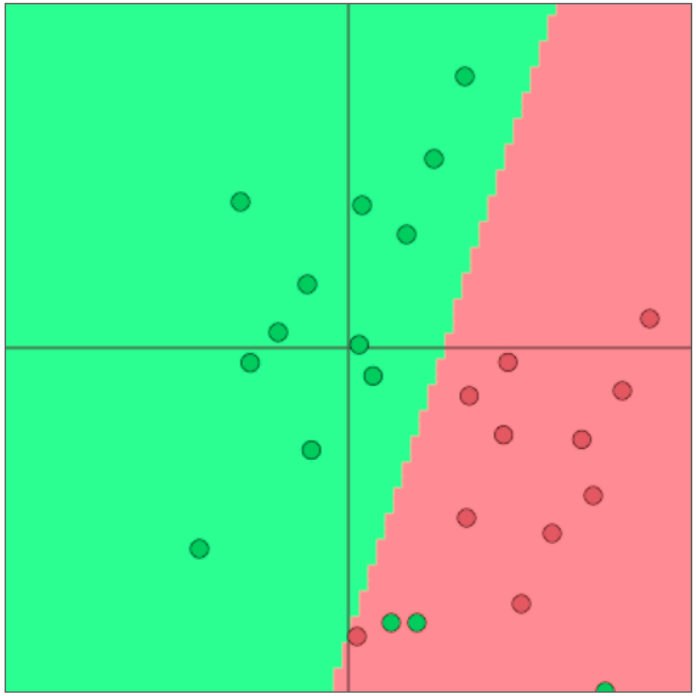
- General ML: assume x is fixed and only train logistic regression weights W and only modify the decision boundary

$$p(y|x) = \frac{\exp(W_y.x)}{\sum_{c=1}^{C} \exp(W_c.x)}$$

Classification notation

- Cross entropy loss function over dataset $\{x_i, y_i\}_{i=1}^N$

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} -\log \left(\frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}} \right)$$



- Where for each data pair (x_i, y_i):

$$f_y = f_y(x) = W_{y\cdot} x = \sum_{j=1}^{d} W_{yj} x_j$$

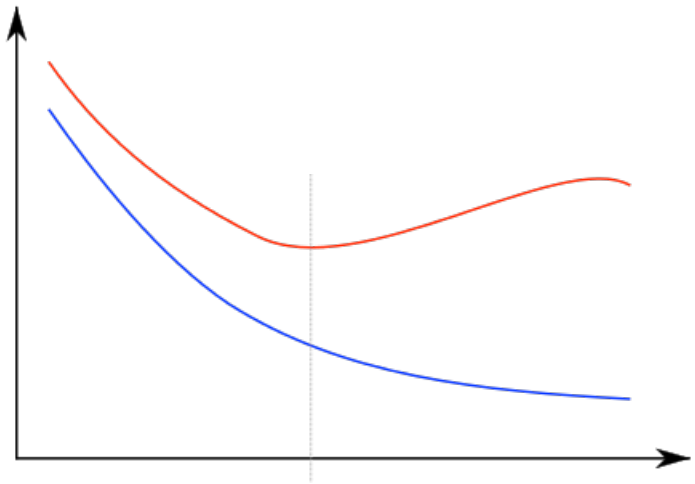
- We can write *f* in matrix notation and index elements of it based on class: $f = Wx$

Classification: Regularization!

- Really full loss function over any dataset includes **regularization** over all parameters μ:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} -\log\left(\frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}}\right) + \lambda \sum_k \theta_k^2$$

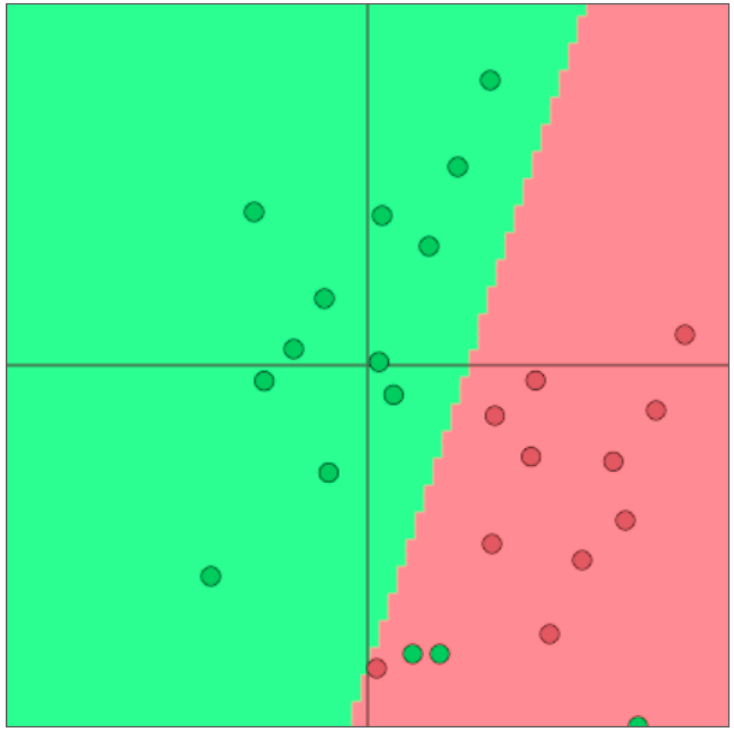
- Regularization will prevent overfitting when we have a lot of features (or later a very powerful/deep model)
  - x-axis: more powerful model or more training iterations
  - Blue: training error, red: test error



Details: General ML optimization

- For general machine learning μ usually only consists of columns of W:

$$\theta = \begin{bmatrix} W_{\cdot 1} \\ \vdots \\ W_{\cdot d} \end{bmatrix} = W(:) \in \mathbb{R}^{Cd}$$



Visualizations with ConvNetJS by Karpathy

- So we only update the decision boundary

$$\nabla_\theta J(\theta) = \begin{bmatrix} \nabla_{W_{\cdot 1}} \\ \vdots \\ \nabla_{W_{\cdot d}} \end{bmatrix} \in \mathbb{R}^{Cd}$$

Classification difference with word vectors

- Common in deep learning:
  - Learn both W and word vectors x

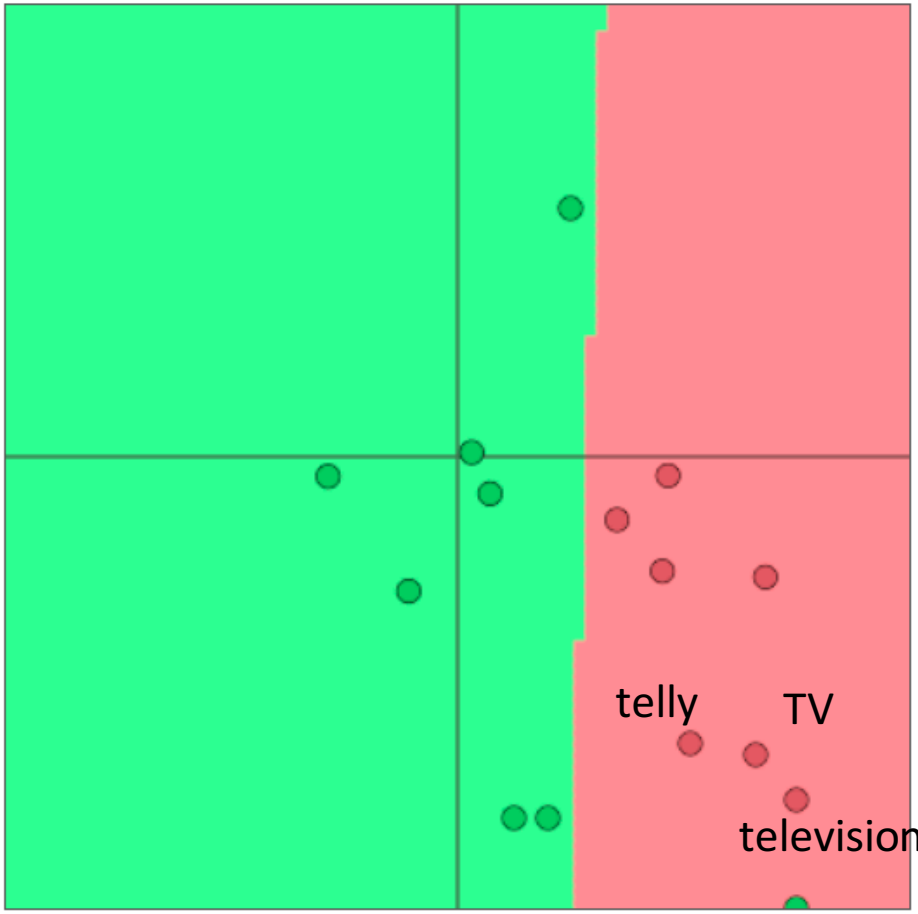
$$\nabla_\theta J(\theta) = \begin{bmatrix} \nabla_{W_{\cdot 1}} \\ \vdots \\ \nabla_{W_{\cdot d}} \\ \nabla_{x_{aardvark}} \\ \vdots \\ \nabla_{x_{zebra}} \end{bmatrix} \in \mathbb{R}^{Cd+Vd}$$

Very large!

Overfitting Danger!

Losing generalization by re-training word vectors

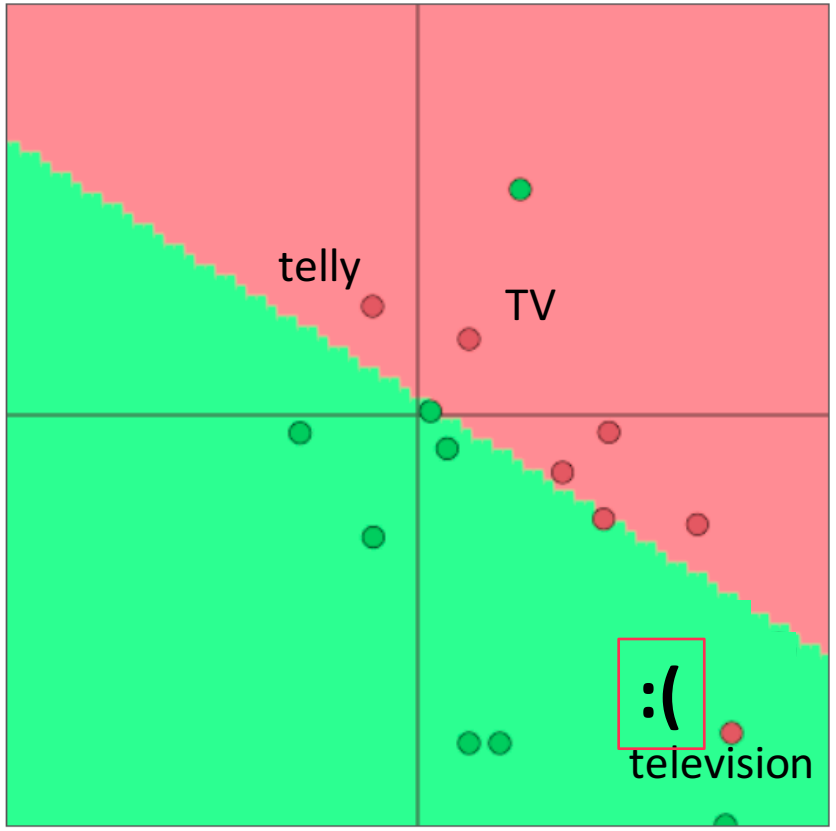
- Setting: Training logistic regression for movie review sentiment and in the training data we have the words
  - "TV" and "telly"
- In the testing data we have
  - "television"
- Originally they were all similar (from pre-training word vectors)
- What happens when we train the word vectors?



Losing generalization by re-training word vectors

- What happens when we train the word vectors?
  - Those that are in the training data move around
  - Words from pre-training that do NOT appear in training stay

- Example:
- In training data: “TV” and “telly”
- In testing data only: “television”

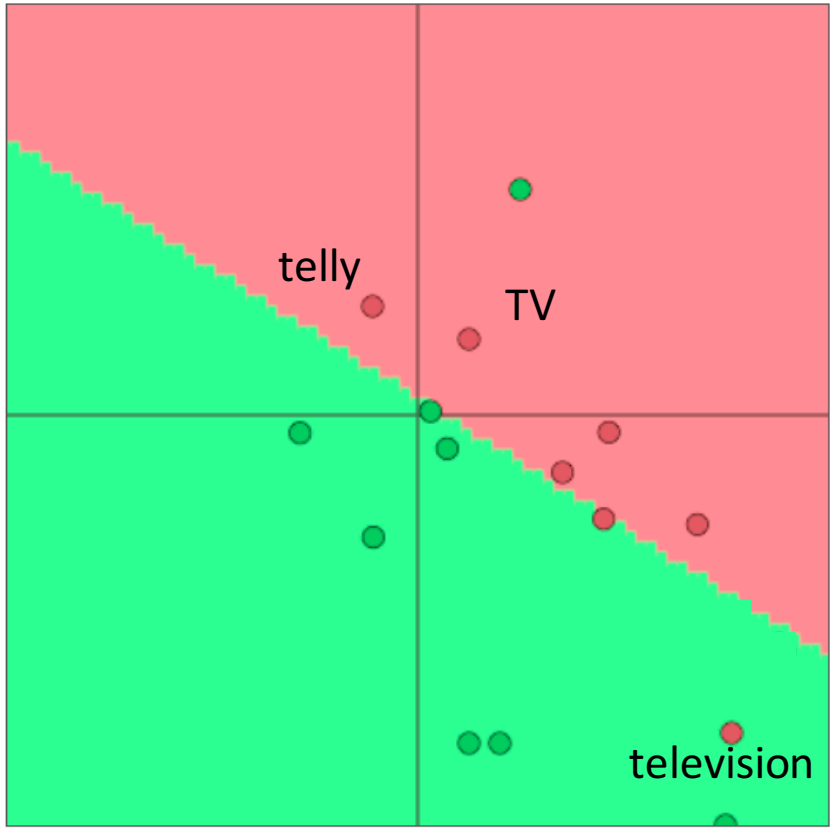


Losing generalization by re-training word vectors

- Take home message:

If you only have a small training data set, don't train the word vectors.

If you have have a very large dataset, it may work better to train word vectors to the task.



Window classification

- Classifying single words is rarely done.
- Interesting problems like ambiguity arise in context!
- Example: auto-antonyms:
  - "To sanction" can mean "to permit" or "to punish."
  - "To seed" can mean "to place seeds" or "to remove seeds."
- Example: ambiguous named entities:
  - Paris → Paris, France vs Paris Hilton
  - Hathaway → Berkshire Hathaway vs Anne Hathaway

Window classification

- Idea: classify a word in its context window of neighboring words.

- For example named entity recognition into 4 classes:
  - Person, location, organization, none

- Many possibilities exist for classifying one word in context, e.g. averaging all the words in a window but that looses position information

Window classification

- Train softmax classifier by assigning a label to a center word and concatenating all word vectors surrounding it

- Example: Classify Paris in the context of this sentence with window length 2:

… museums in Paris are amazing … .

$X_{window} = [\ x_{museums}\quad x_{in}\quad x_{Paris}\quad x_{are}\quad x_{amazing}\ ]^T$

- Resulting vector $x_{window} = x \in R^{5d}$, a column vector!

Simplest window classifier: Softmax

- With $x = x_{window}$ we can use the same softmax classifier as before

$$\hat{y}_y = p(y|x) = \frac{\exp(W_y.x)}{\sum_{c=1}^{C} \exp(W_c.x)}$$

predicted model output probability

same

- With cross entropy error as before:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} -\log \left(\frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}} \right)$$

- But how do you update the word vectors?

Updating concatenated word vectors

- Short answer: Just take derivatives as before

- Long answer: Let's go over the steps together

- Define:
  - $\hat{y}$: softmax probability output vector (see previous slide)
  - t: target probability distribution (all 0's except at ground truth index of class y, where it's 1)
  - $f = f(x) = Wx \in \mathbb{R}^C$ and f_c = c'th element of the f vector

- Hard, the first time, hence some tips now :)

Updating concatenated word vectors

- Tip 1: Carefully define your variables and keep track of their dimensionality!

$$f = f(x) = Wx \in \mathbb{R}^C$$
$$\hat{y} \quad t \qquad W \in \mathbb{R}^{C \times 5d}$$

- Tip 2: **Know thy chain rule** and don't forget which variables depend on what:

$$\frac{\partial}{\partial x} - \log softmax(f_y(x)) = \sum_{c=1}^{C} -\frac{\partial \log softmax(f_y(x))}{\partial f_c} \cdot \frac{\partial f_c(x)}{\partial x}$$

- Tip 3: For the softmax part of the derivative: First take the derivative wrt f_c when c=y (the correct class), then take derivative wrt f_c when c≠ y (all the incorrect classes)

Updating concatenated word vectors

- Tip 4: When you take derivative wrt one element of f, try to see if you can create a gradient in the end that includes all partial derivatives:

$$\hat{y} \quad t$$

$$f = f(x) = Wx \in \mathbb{R}^C$$

$$\frac{\partial}{\partial f} - \log softmax(f_y) = \begin{bmatrix} \hat{y}_1 \\ \vdots \\ \hat{y}_y - 1 \\ \vdots \\ \hat{y}_C \end{bmatrix}$$

- Tip 5: To later not go insane & implementation! → results in terms of vector operations and define single index-able vectors:

$$\frac{\partial}{\partial f} - \log softmax(f_y) = [\hat{y} - t] = \delta$$

Updating concatenated word vectors

- Tip 6: When you start with the chain rule, first use explicit sums and look at partial derivatives of e.g. x_i or W_{ij}

$$\hat{y} \quad t$$
$$f = f(x) = Wx \in \mathbb{R}^C$$

$$\sum_{c=1}^{C} -\frac{\partial \log softmax(f_y(x))}{\partial f_c} \cdot \frac{\partial f_c(x)}{\partial x} = \sum_{c=1}^{C} \delta_c W_{c\cdot}^T$$

- Tip 7: To clean it up for even more complex functions later: Know dimensionality of variables &simplify into matrix notation

$$\frac{\partial}{\partial x} - \log p(y|x) = \sum_{c=1}^{C} \delta_c W_{c\cdot}^T = W^T \delta$$

- Tip 8: Write this out in full sums if it's not clear!

Updating concatenated word vectors

- What is the dimensionality of the window vector gradient?

$$\frac{\partial}{\partial x} - \log p(y|x) = \sum_{c=1}^{C} \delta_c W_{c\cdot} = W^T \delta$$

- x is the entire window, 5 d-dimensional word vectors, so the derivative wrt to x has to have the same dimensionality:

$$\nabla_x J = W^T \delta \in \mathbb{R}^{5d}$$

Updating concatenated word vectors

- The gradient that arrives at and updates the word vectors can simply be split up for each word vector:
- Let $\nabla_x J = W^T \delta = \delta_{x_{window}}$
- With x_{window} = [$x_{museums}$ x_{in} x_{Paris} x_{are} $x_{amazing}$]

- We have

$$\delta_{window} = \begin{bmatrix} \nabla_{x_{museums}} \\ \nabla_{x_{in}} \\ \nabla_{x_{Paris}} \\ \nabla_{x_{are}} \\ \nabla_{x_{amazing}} \end{bmatrix} \in \mathbb{R}^{5d}$$

Updating concatenated word vectors

- This will push word vectors into areas such they will be helpful in determining named entities.

- For example, the model can learn that seeing x_{in} as the word just before the center word is indicative for the center word to be a location

What's missing for training the window model?

- The gradient of J wrt the softmax weights W!

- Similar steps, write down partial wrt W_{ij} first!
- Then we have full

$$\nabla_\theta J(\theta) = \begin{bmatrix} \nabla_{W_{\cdot 1}} \\ \vdots \\ \nabla_{W_{\cdot d}} \\ \nabla_{x_{aardvark}} \\ \vdots \\ \nabla_{x_{zebra}} \end{bmatrix} \in \mathbb{R}^{Cd+Vd}$$

A note on matrix implementations

- There are two expensive operations in the softmax:
- The matrix multiplication $f = Wx$ and the exp
- A for loop is never as efficient when you implement it compared vs when you use a larger matrix multiplication!

- Example code →

A note on matrix implementations

- Looping over word vectors instead of concatenating them all into one large matrix and then multiplying the softmax weights with that matrix

```
from numpy import random
N = 500 # number of windows to classify
d = 300 # dimensionality of each window
C = 5 # number of classes
W = random.rand(C,d)
wordvectors_list = [random.rand(d,1) for i in range(N)]
wordvectors_one_matrix = random.rand(d,N)

%timeit [W.dot(wordvectors_list[i]) for i in range(N)]
%timeit W.dot(wordvectors_one_matrix)
```

- 1000 loops, best of 3: 639 µs per loop
  10000 loops, best of 3: 53.8 µs per loop

A note on matrix implementations

```
from numpy import random
N = 500 # number of windows to classify
d = 300 # dimensionality of each window
C = 5 # number of classes
W = random.rand(C,d)
wordvectors_list = [random.rand(d,1) for i in range(N)]
wordvectors_one_matrix = random.rand(d,N)

%timeit [W.dot(wordvectors_list[i]) for i in range(N)]
%timeit W.dot(wordvectors_one_matrix)
```

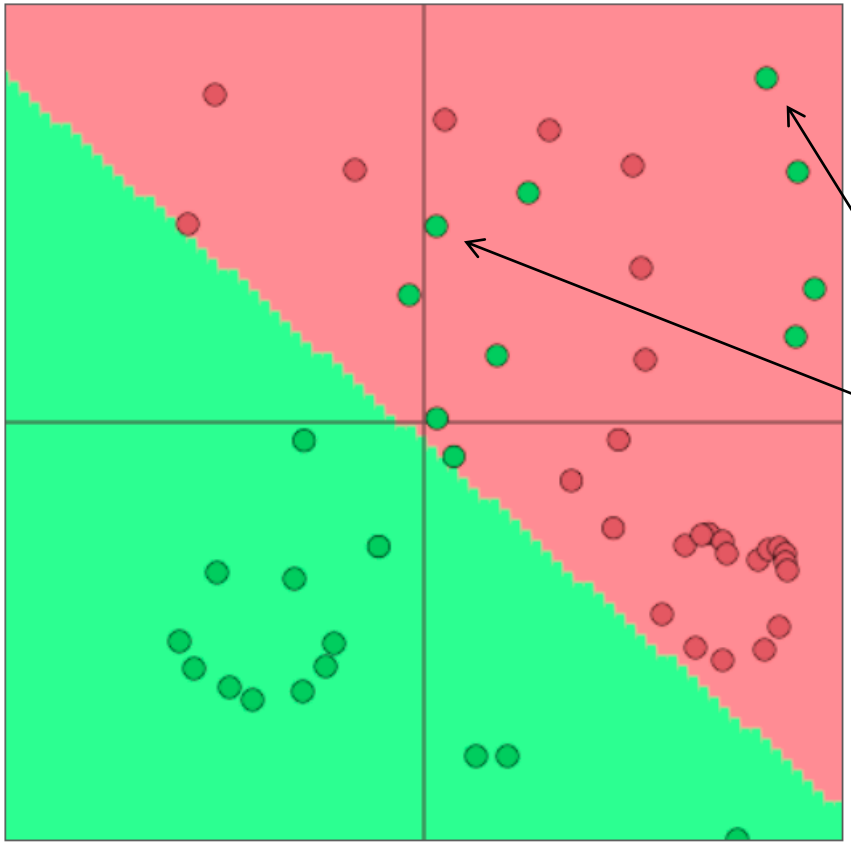
- Result of faster method is a C x N matrix:
  - Each column is an f(x) in our notation (unnormalized class scores)
- Matrices are awesome!
- You should speed test your code a lot too

Softmax (= logistic regression) is not very powerful

- Softmax only gives linear decision boundaries in the original space.
- With little data that can be a good regularizer
- With more data it is very limiting!

Softmax (= logistic regression) is not very powerful

- Softmax only linear decision boundaries

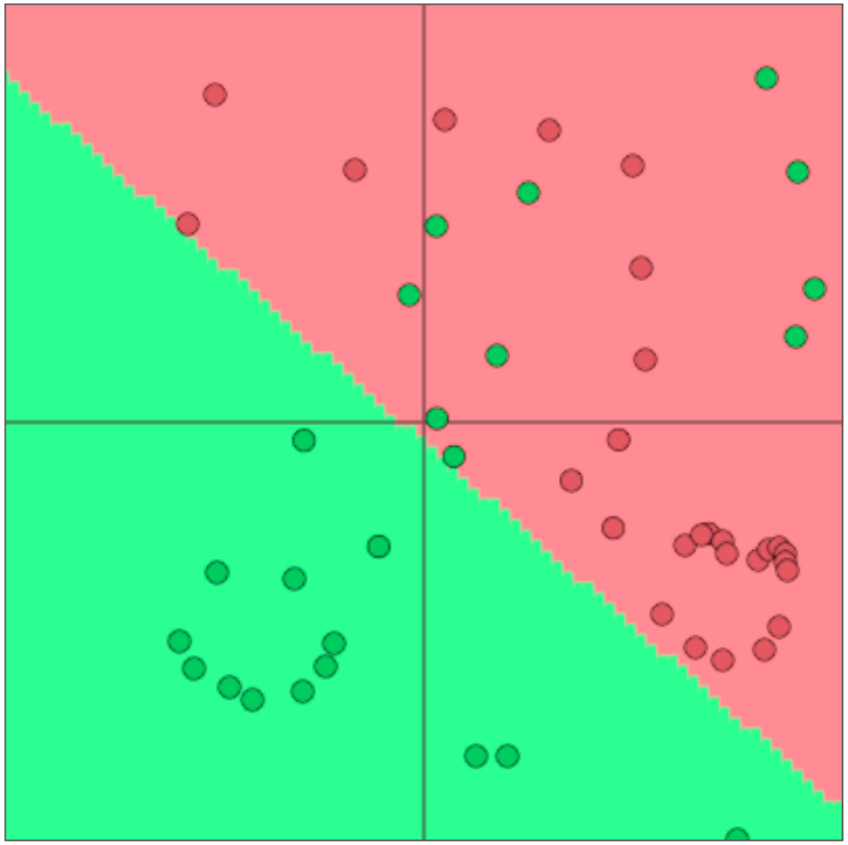


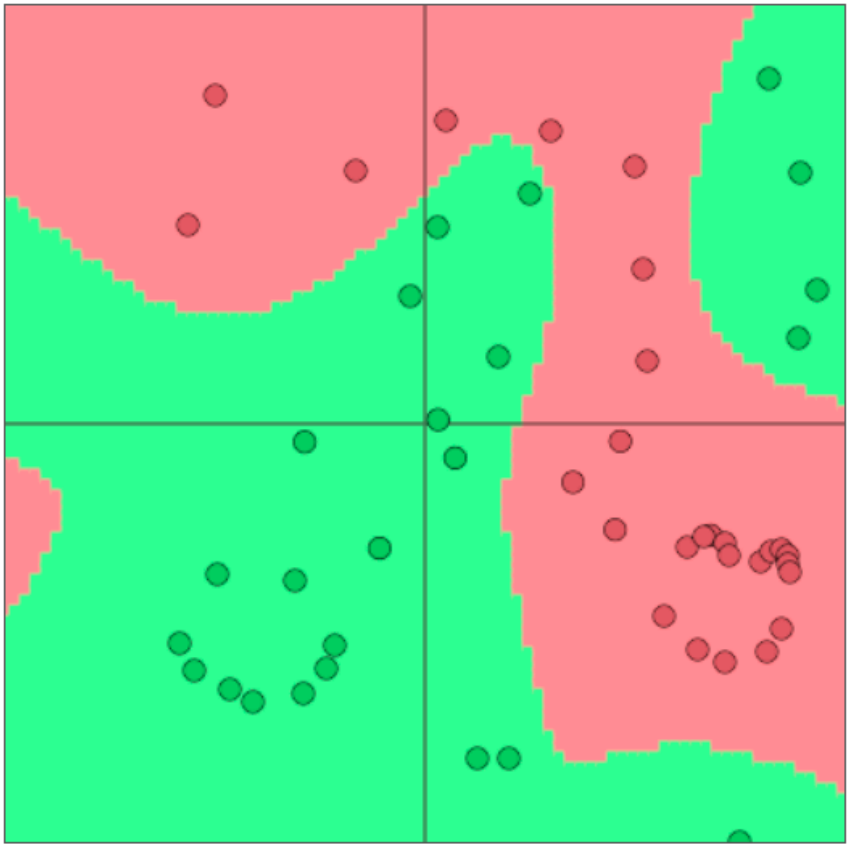
→ Lame when problem is complex

Wouldn't it be cool to get these correct?

Neural Nets for the Win!

- Neural networks can learn much more complex functions and nonlinear decision boundaries!





From logistic regression to neural nets

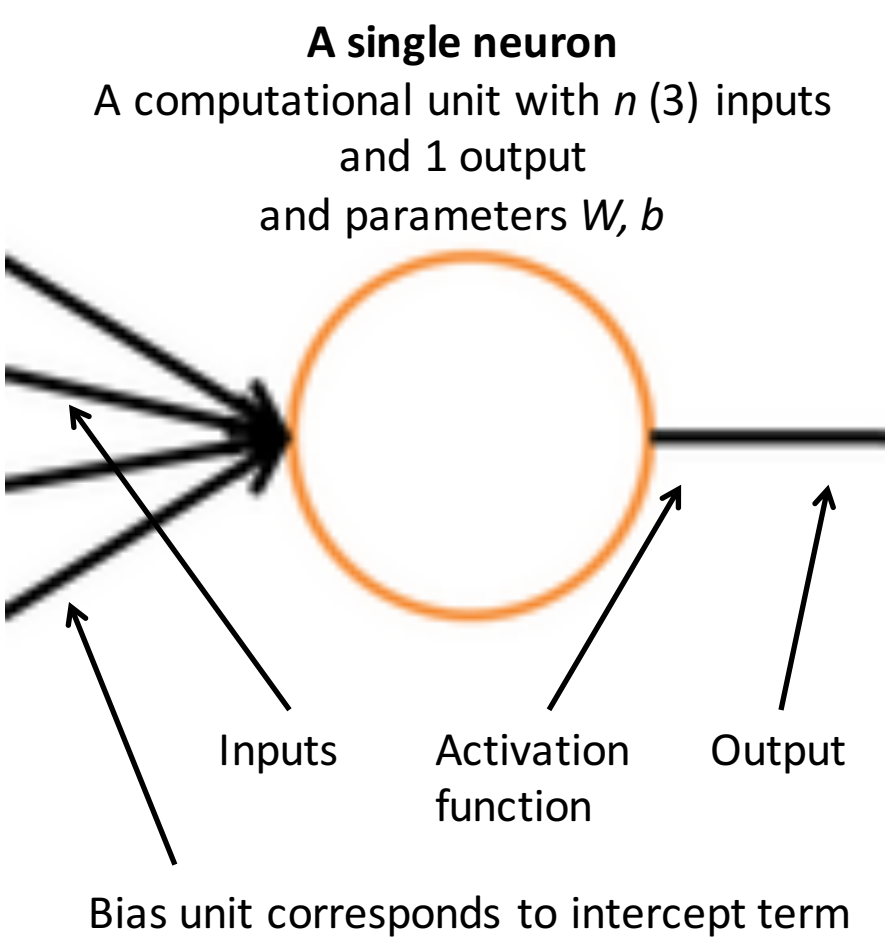
Demystifying neural networks

Neural networks come with their own terminological baggage

… just like SVMs

But if you understand how softmax models work

Then **you already understand** the operation of a basic neural network neuron!

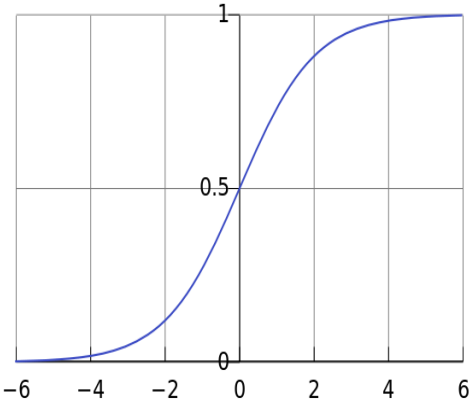


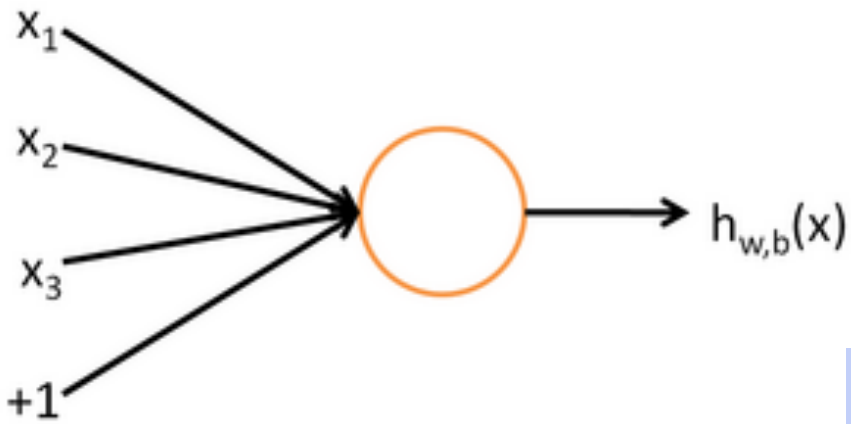
A neuron is essentially a binary logistic regression unit

$$h_{w,b}(x) = f(w^{\mathsf{T}}x + b)$$

$$f(z) = \frac{1}{1+e^{-z}}$$

b: We can have an "always on" feature, which gives a class prior, or separate it out, as a bias term

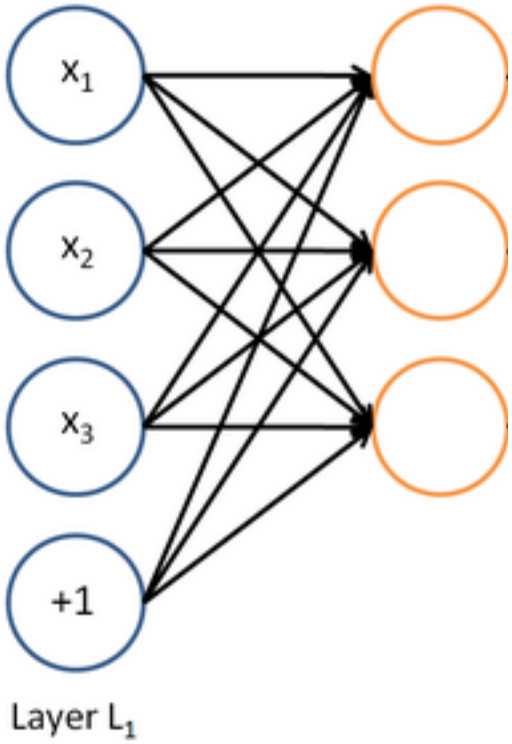




w, *b* are the parameters of this neuron i.e., this logistic regression model

A neural network
= running several logistic regressions at the same time

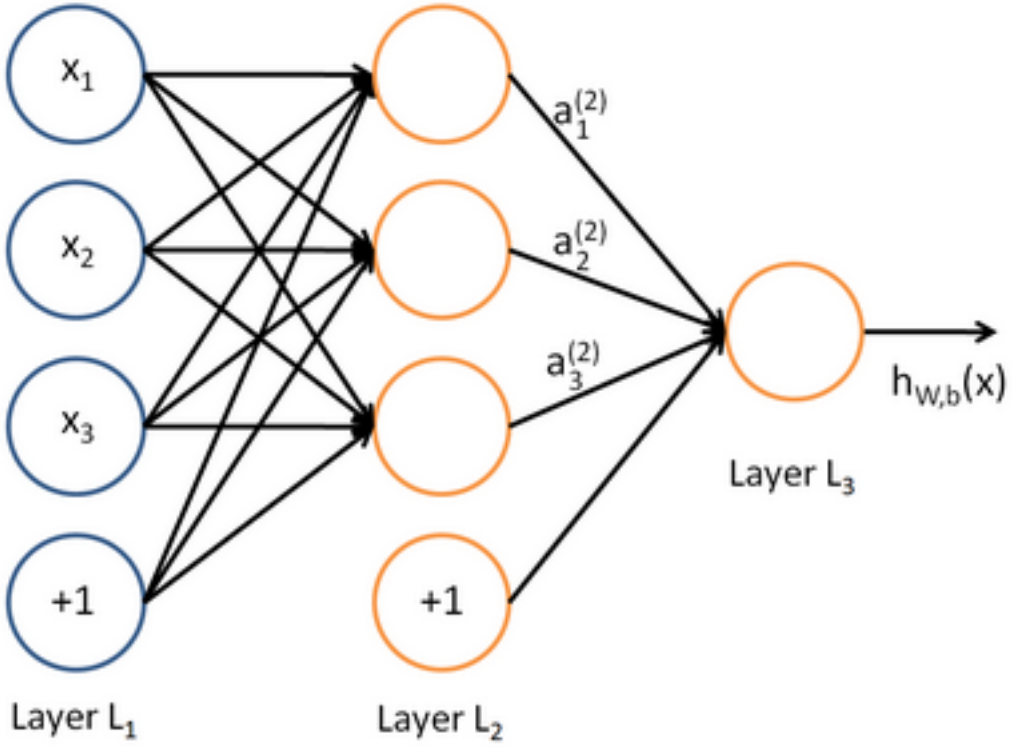
If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs …



But we don't have to decide ahead of time what variables these logistic regressions are trying to predict!

A neural network
= running several logistic regressions at the same time

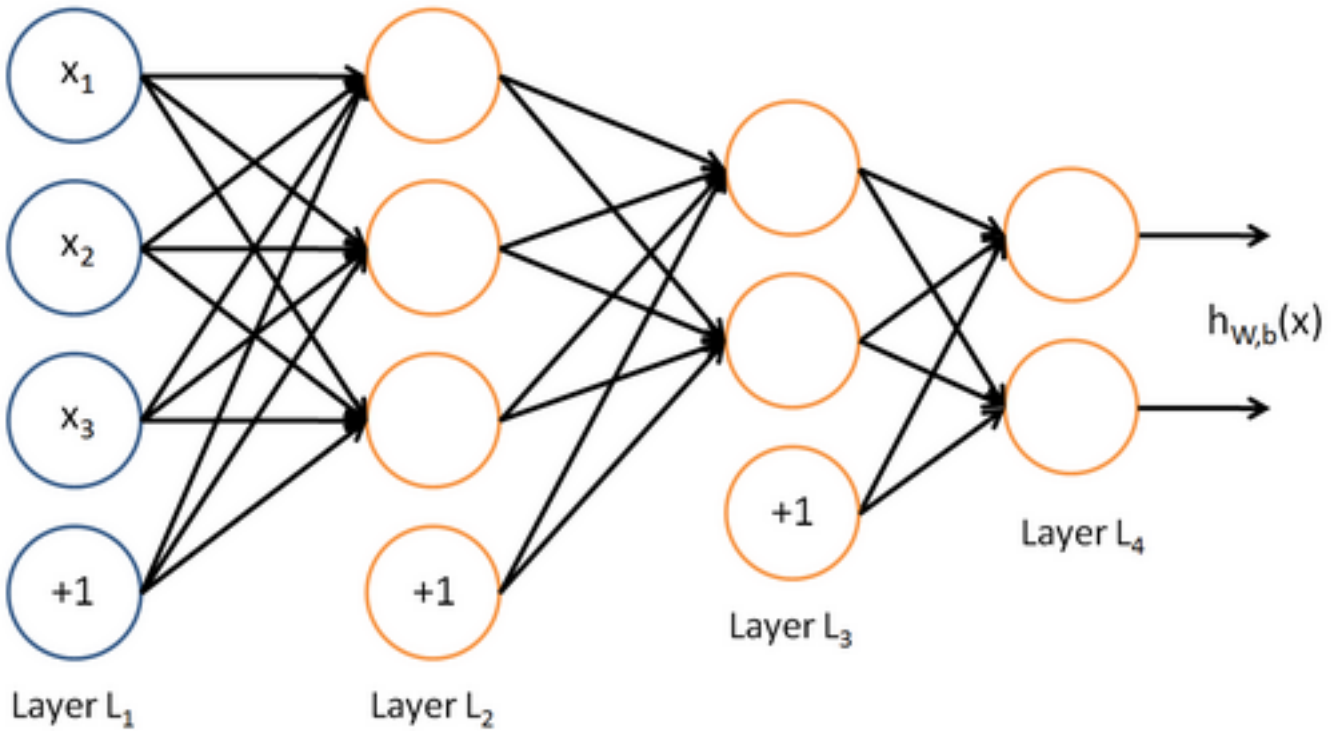
… which we can feed into another logistic regression function



It is the loss function that will direct what the intermediate hidden variables should be, so as to do a good job at predicting the targets for the next layer, etc.

A neural network
= running several logistic regressions at the same time

Before we know it, we have a multilayer neural network….



Matrix notation for a layer

We have

$$a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1)$$

$$a_2 = f(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2)$$

etc.

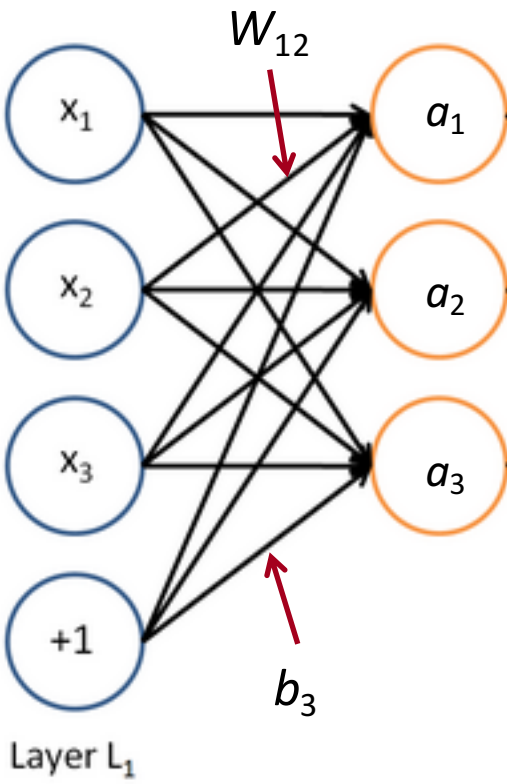
In matrix notation

$$z = Wx + b$$

$$a = f(z)$$

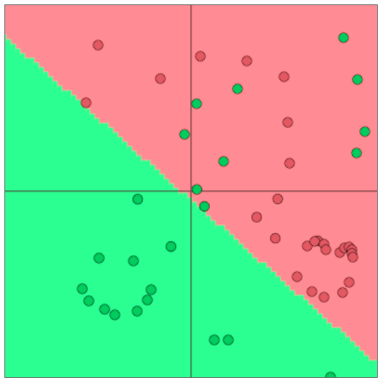
where f is applied element-wise:

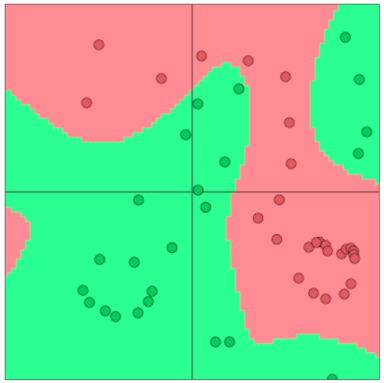
$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$

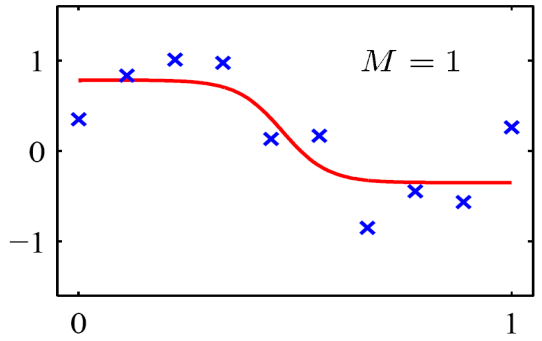


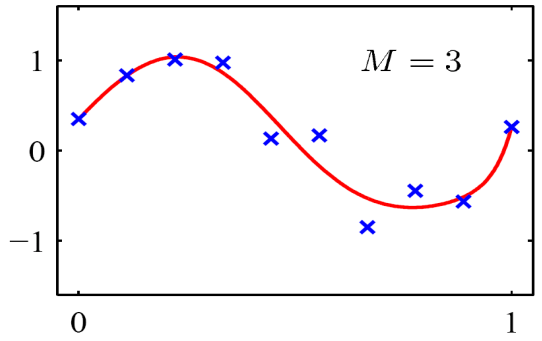
Non-linearities (f): Why they're needed

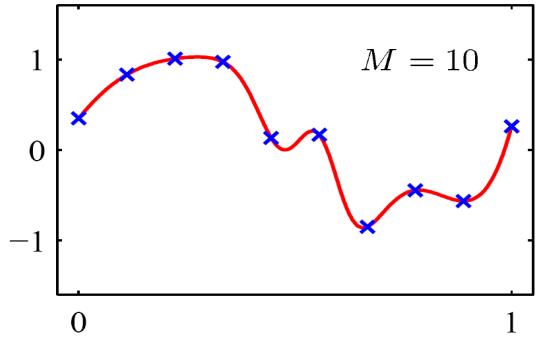
- Example: function approximation, e.g., regression or classification
  - Without non-linearities, deep neural networks can't do anything more than a linear transform
  - Extra layers could just be compiled down into a single linear transform: $W_1 W_2 x = Wx$
  - With more layers, they can approximate more complex functions!











Backpropagation (Another explanation)

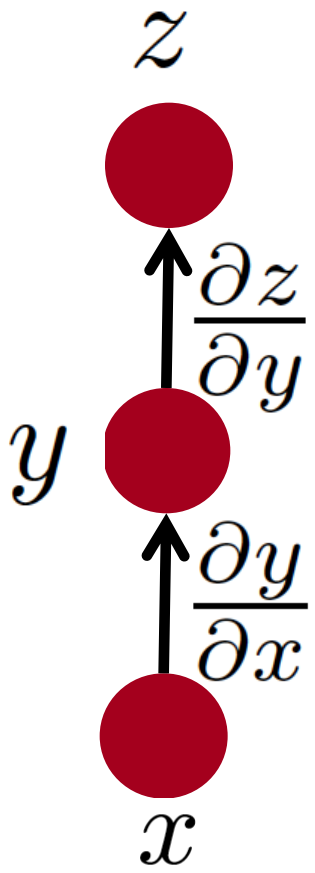
- Compute gradient of example-wise loss wrt parameters

- Simply applying the derivative chain rule wisely

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

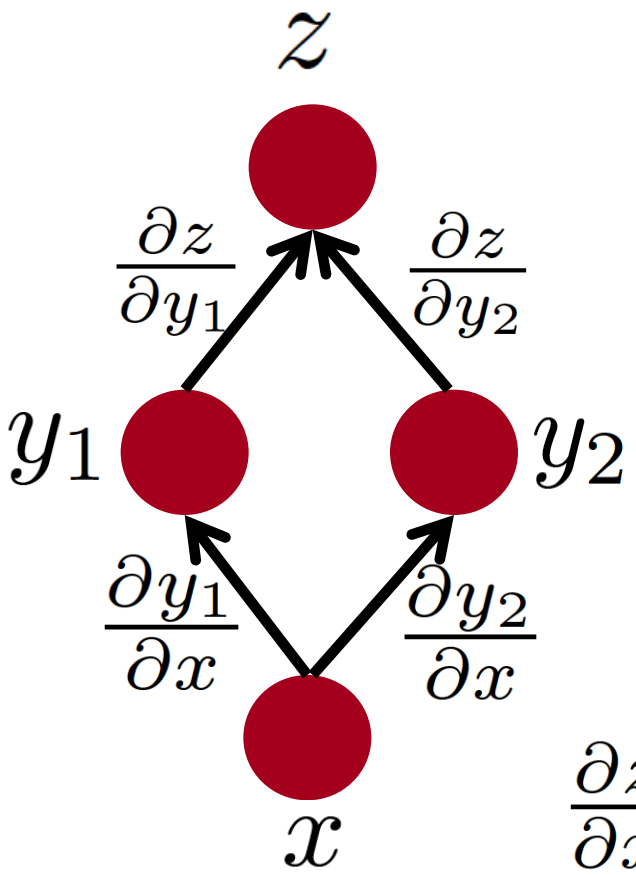
- If computing the loss(example, parameters) is $O(n)$ computation, then so is computing the gradient

Simple Chain Rule



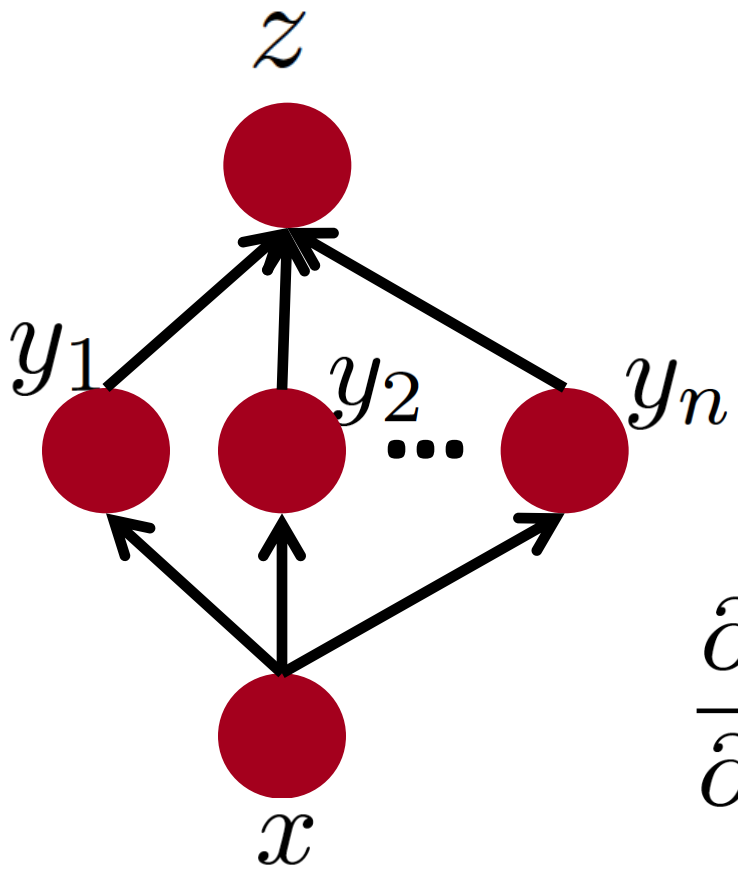
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

Multiple Paths Chain Rule



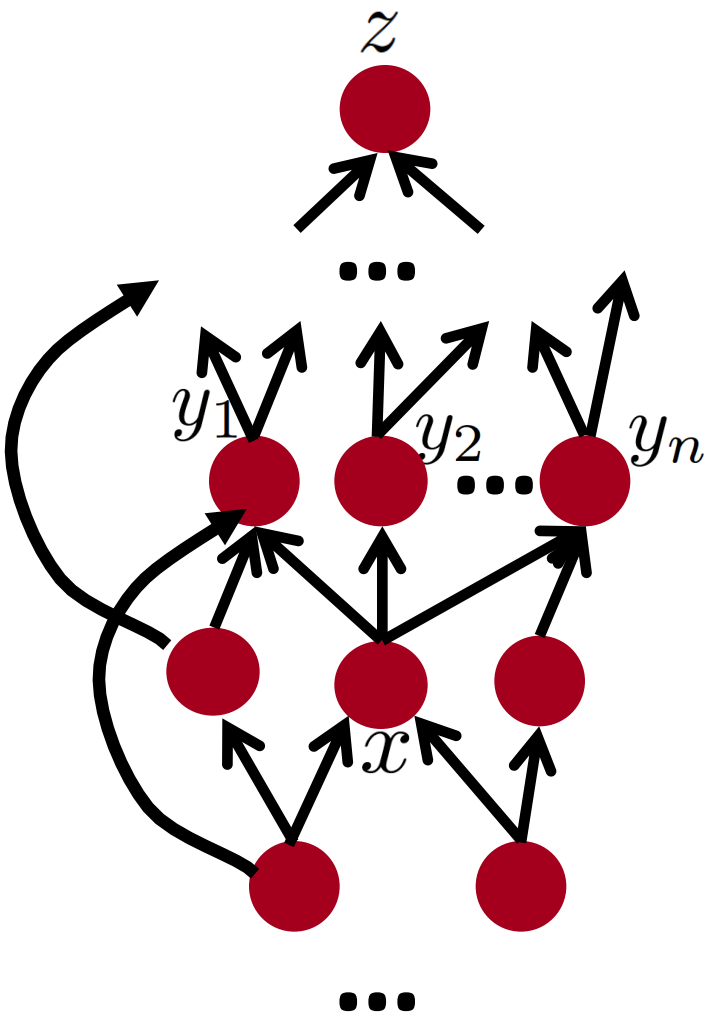
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$$

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

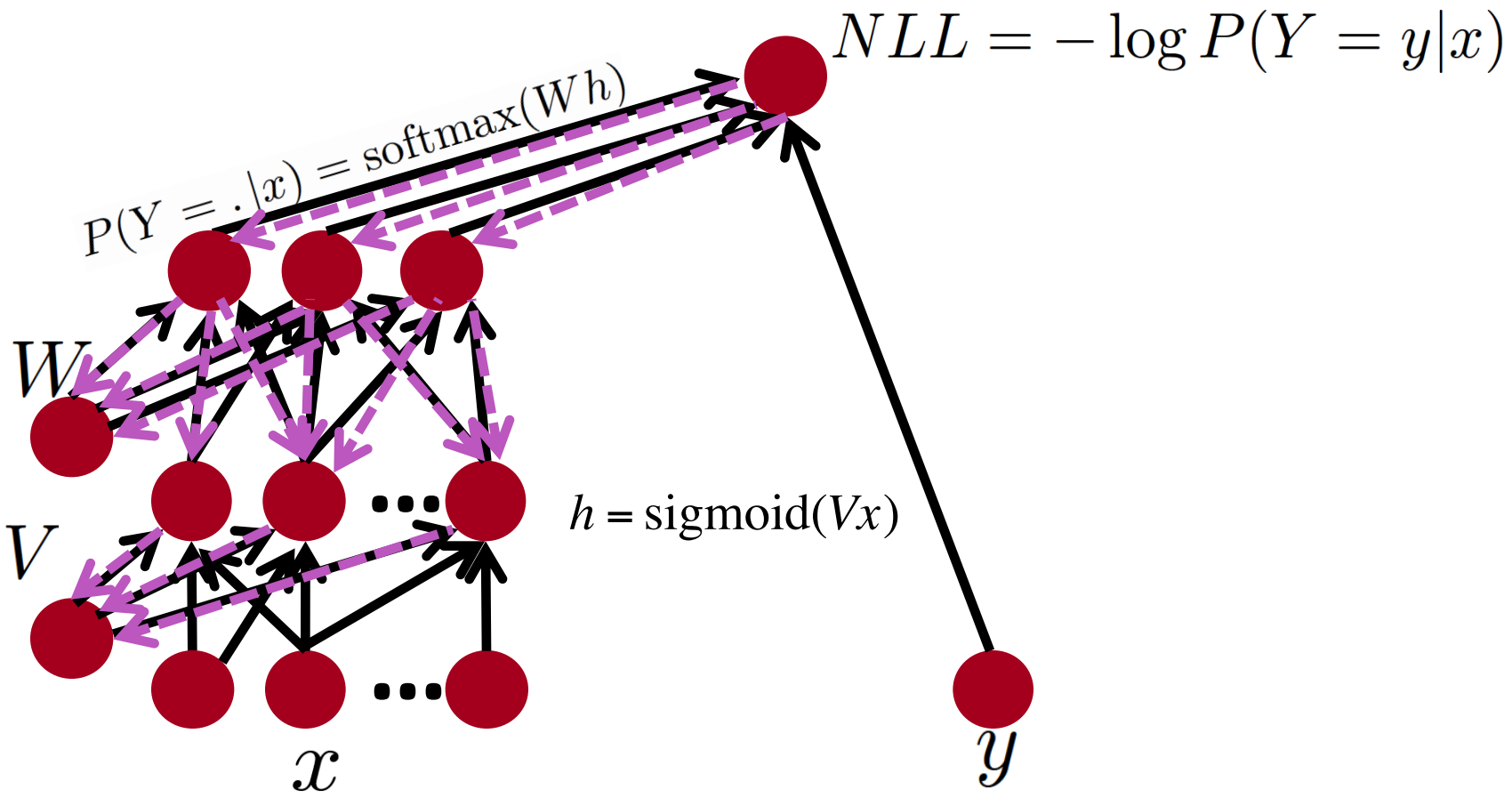


Flow graph: any directed acyclic graph
node = computation result
arc = computation dependency

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

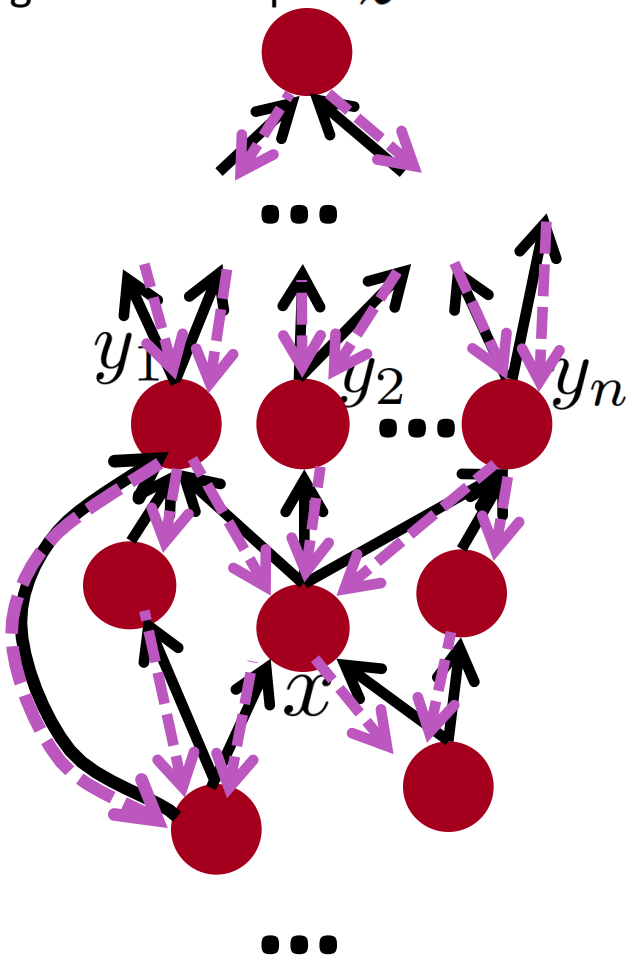
$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Back-Prop in Multi-Layer Net



Back-Prop in General Flow Graph

Single scalar output z

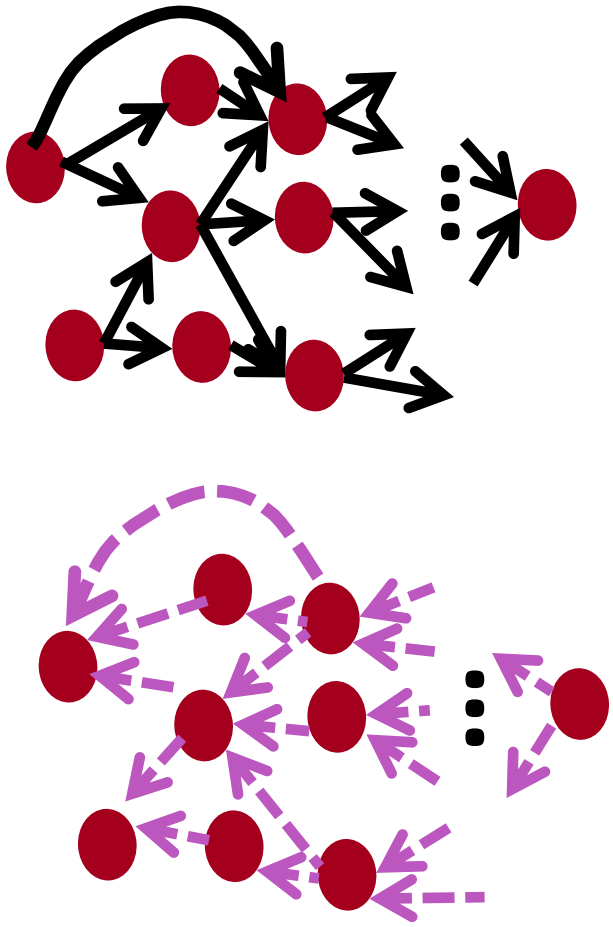


1. Fprop: visit nodes in topo-sort order
   - Compute value of node given predecessors
2. Bprop:
   - initialize output gradient = 1
   - visit nodes in reverse order:
     Compute gradient wrt each node using gradient wrt successors

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

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

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

A more powerful window classifier

- Revisiting

- $X_{window} = [\ x_{museums} \quad x_{in} \quad x_{Paris} \quad x_{are} \quad x_{amazing}\ ]$

- 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:

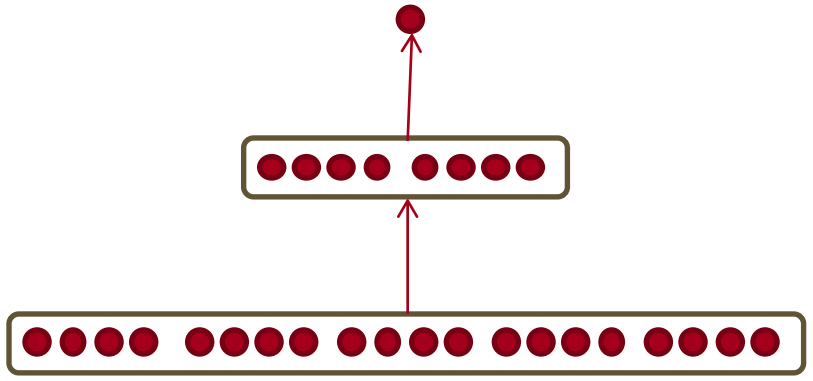
$$score(x) = U^T a \in \mathbb{R}$$

Summary: Feed-forward Computation

Computing a window's score with a 3-layer neural net: *s = score*(museums in Paris are amazing)

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

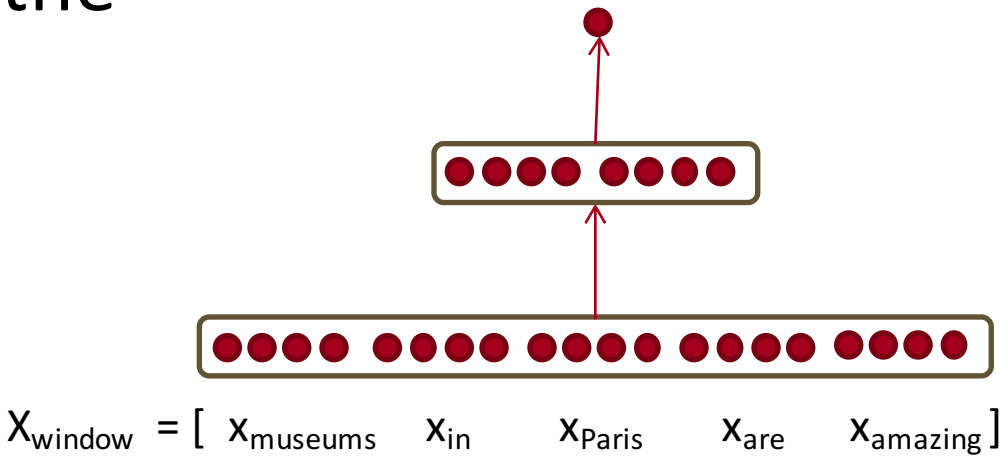
$$\begin{aligned} s &= U^T a \\ a &= f(z) \\ z &= Wx + b \end{aligned}$$



Main intuition for extra layer

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

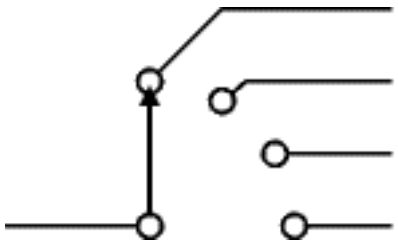
Example:
only if "*museums*" is first vector should it matter that "*in*" is in the second position



Summary: Feed-forward Computation

- s = score(museums in Paris are amazing)
- s_c = score(Not all museums in Paris)
- Idea for training objective: make score of true window larger and **c**orrupt 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*

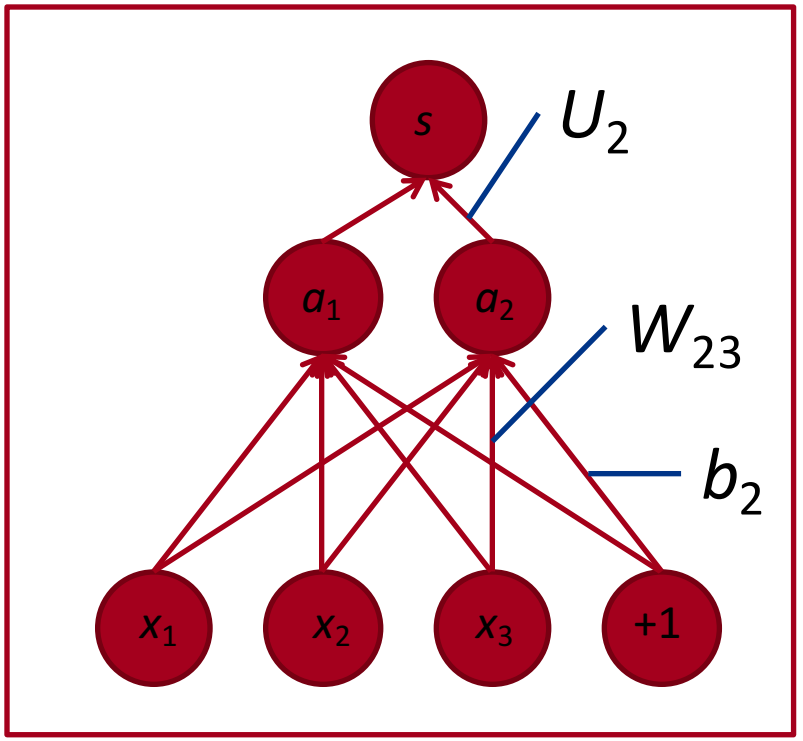
$$\frac{\partial s}{\partial U} = \frac{\partial}{\partial U} U^T a \qquad \frac{\partial s}{\partial U} = a$$

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

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

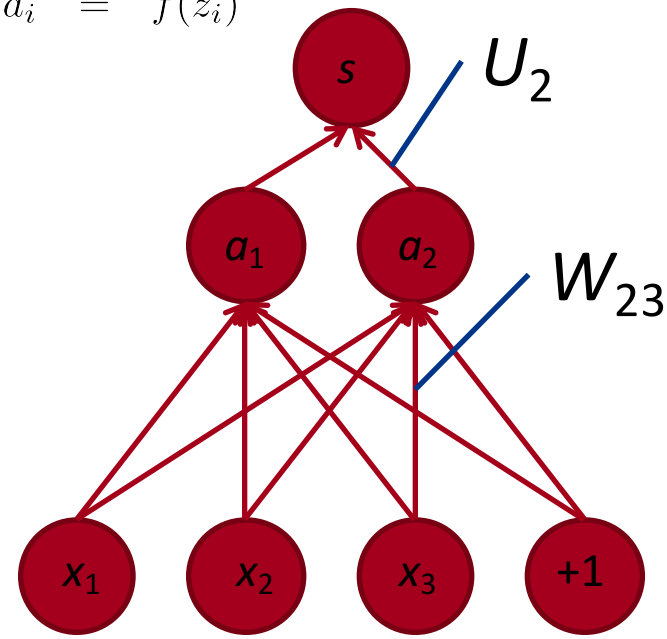
Derivative of weight W_{ij}:

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

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

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

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



Training with Backpropagation

Derivative of single weight W_{ij} :

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

$$a_i = f(z_i)$$

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

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

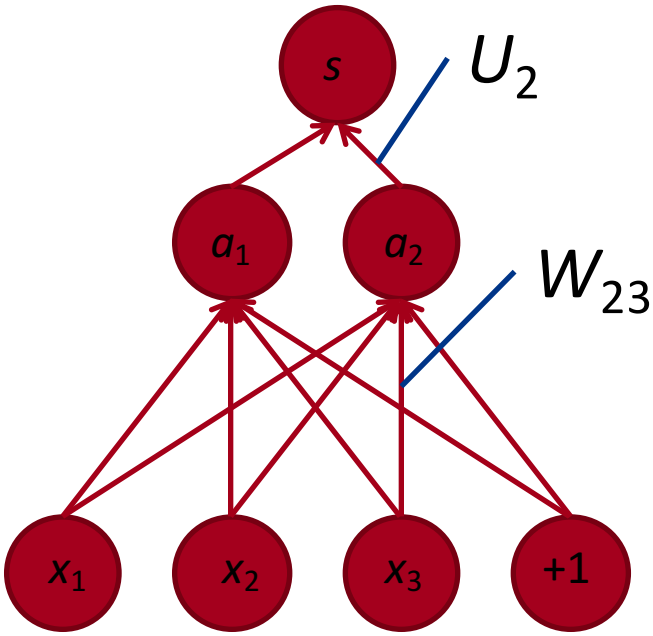
$$= \underbrace{U_i f'(z_i)}\, x_j$$

$$= \delta_i \quad x_j$$

Local error signal

Local input signal

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



Training with Backpropagation

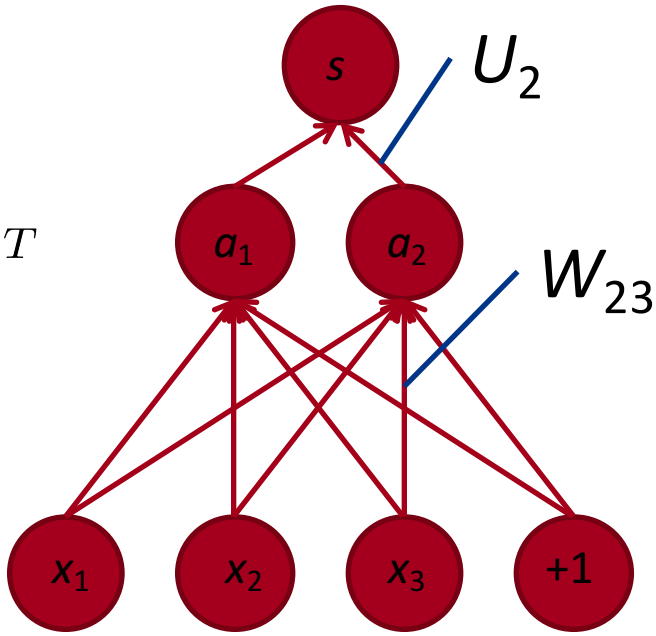
- From single weight W_{ij} to full W:

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

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

$$a_i = f(z_i)$$

- We want all combinations of $i = 1, 2$ and $j = 1, 2, 3$ → ?
- 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



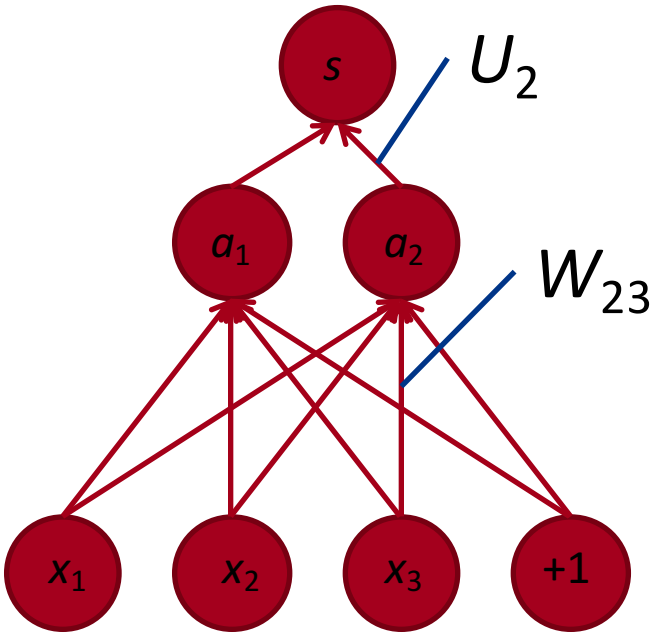
Training with Backpropagation

- For biases *b*, we get:

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

$$a_i = f(z_i)$$

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



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:

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

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? →

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

- 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:

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

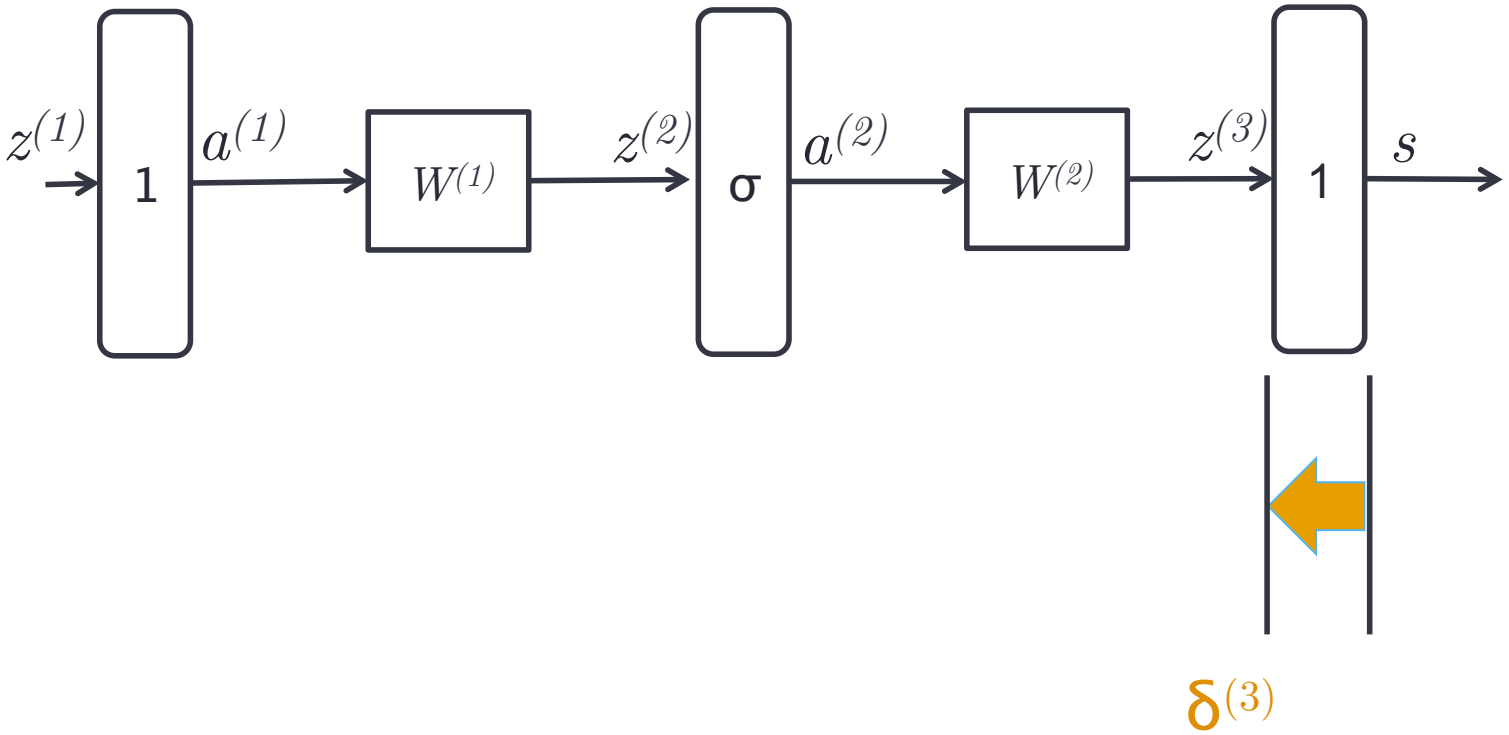
$$s = U^T f(Wx + b)$$
$$s_c = U^T f(Wx_c + b)$$

- For example: gradient for U:

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

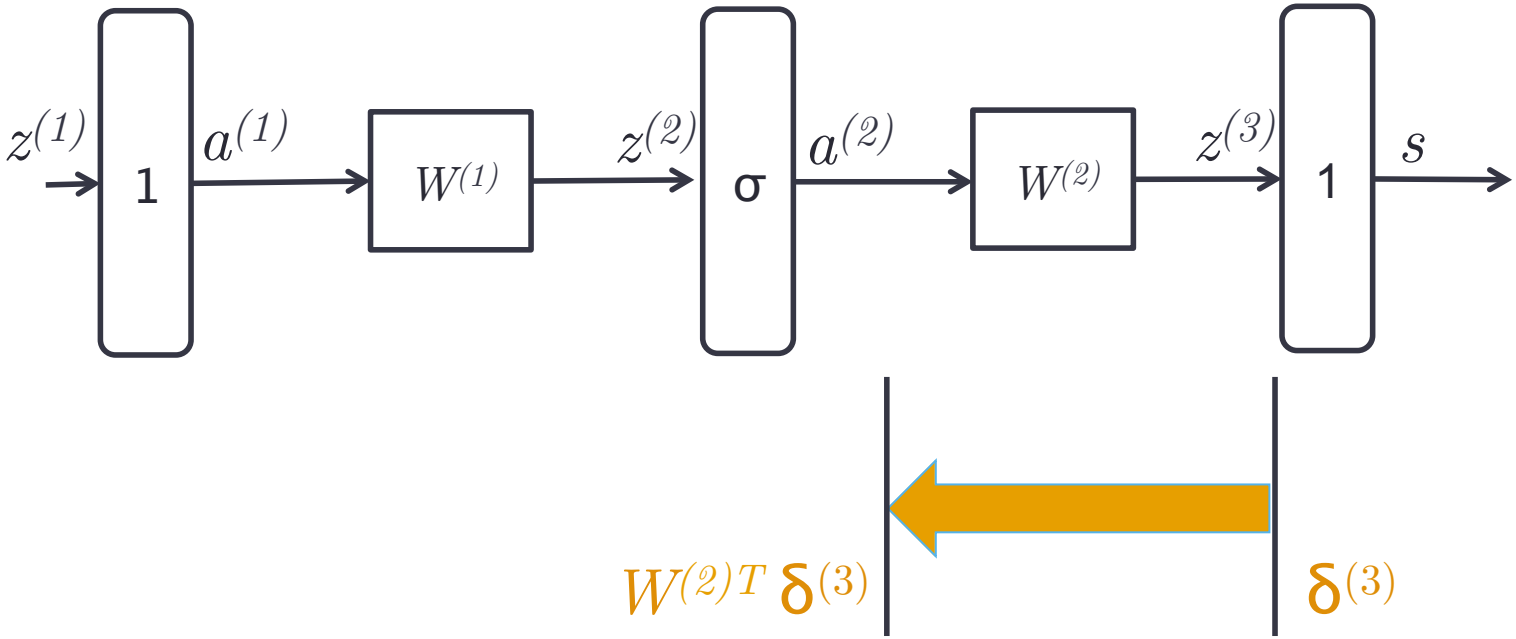
Visualization of intuition

- Let's say we want $\frac{\partial s}{\partial W^{(1)}} = \delta^{(2)} a^{(1)^T}$ with previous layer and f = ¾



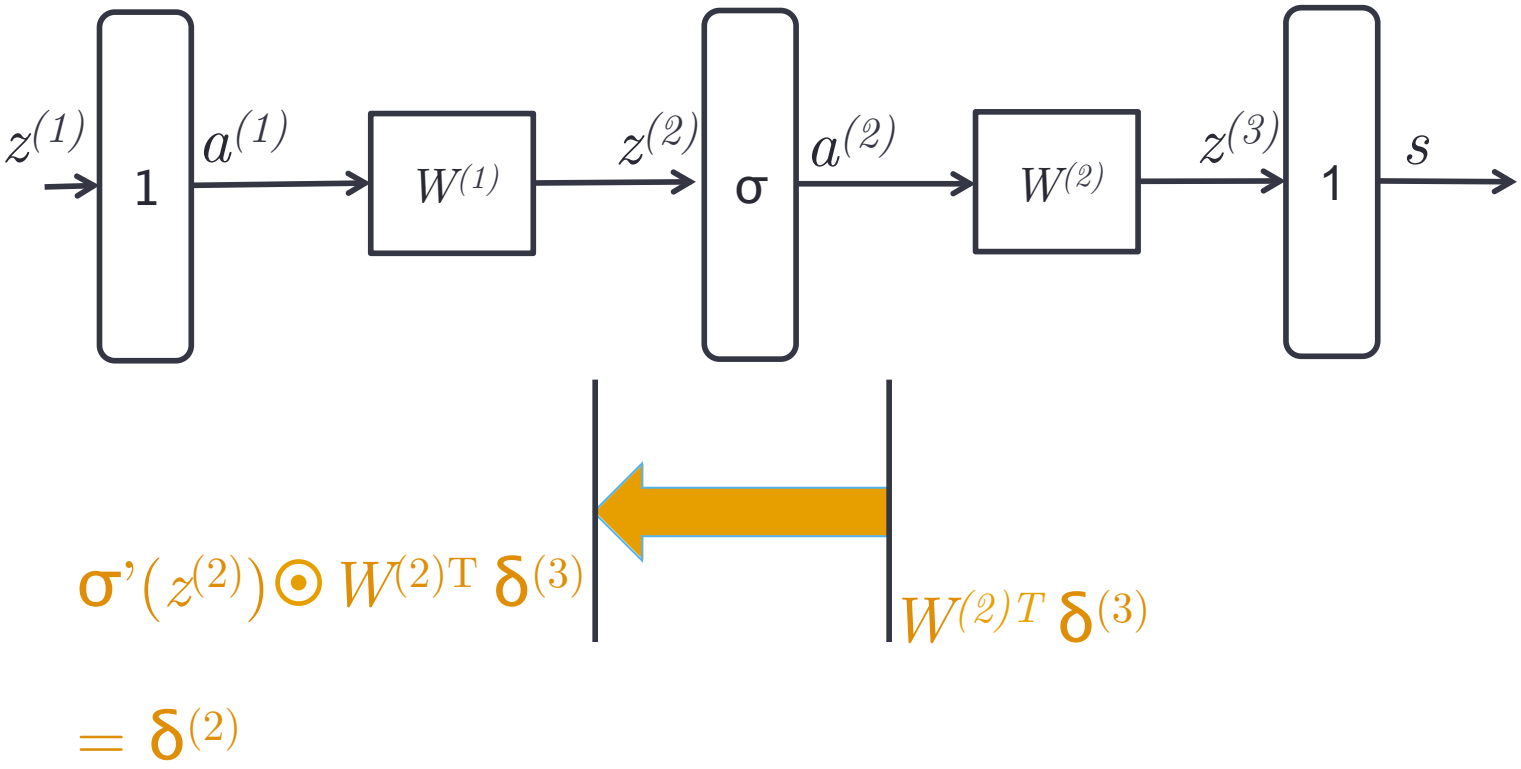
Gradient w.r.t $W^{(2)} = \delta^{(3)} a^{(2)\mathrm{T}}$

Visualization of intuition



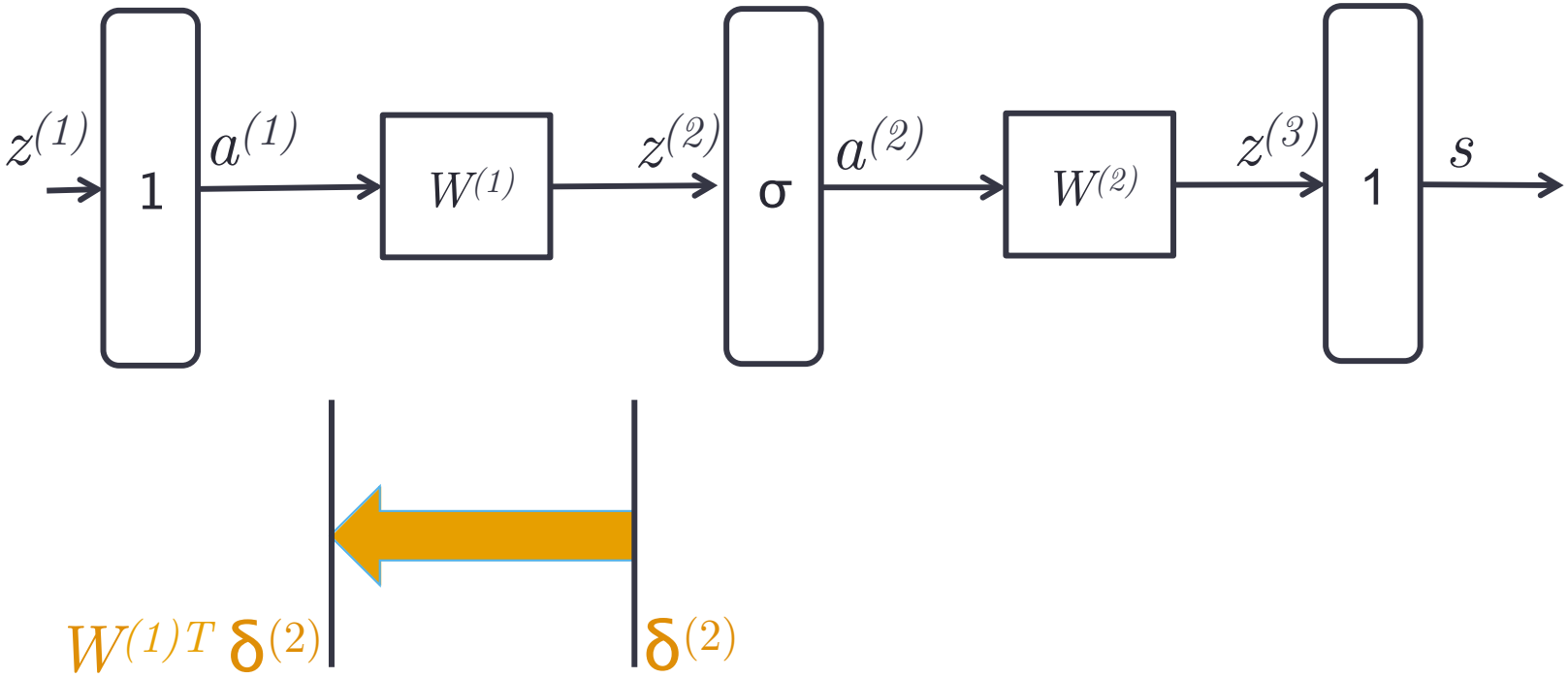
--Reusing the $\delta^{(3)}$ for downstream updates.
--Moving error vector across affine transformation simply requires multiplication with the transpose of forward matrix
--Notice that the dimensions will line up perfectly too!

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 $W^{(1)} = \delta^{(2)} a^{(1)\mathrm{T}}$

Neural Tips and Tricks

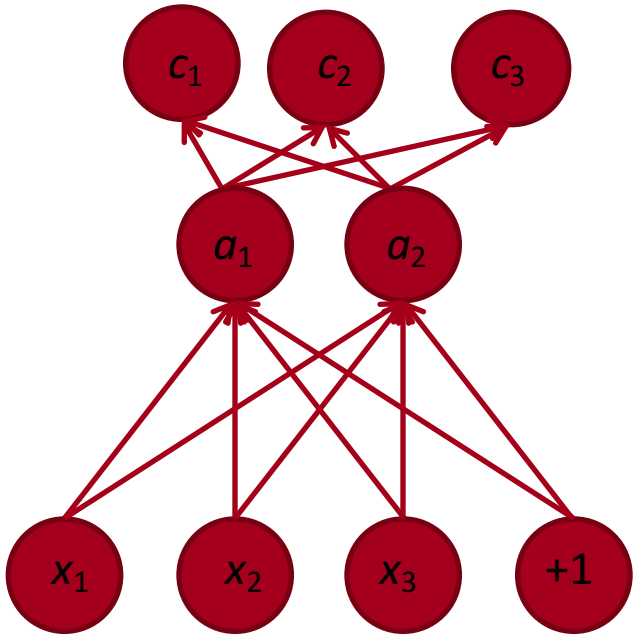
Multi-task learning / Weight sharing

- Base model: Neural network from last class but replaces the single scalar score with a *Softmax* classifier

- Training is again done via backpropagation

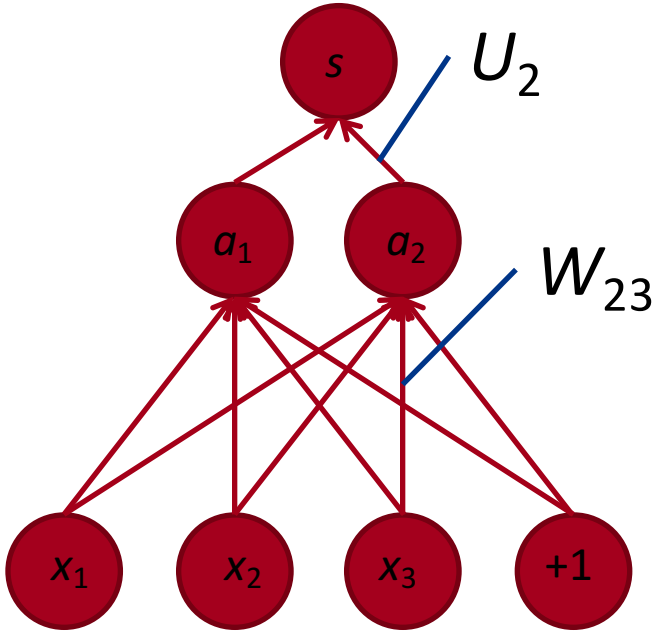
- NLP (almost) from scratch, Collobert et al. 2011

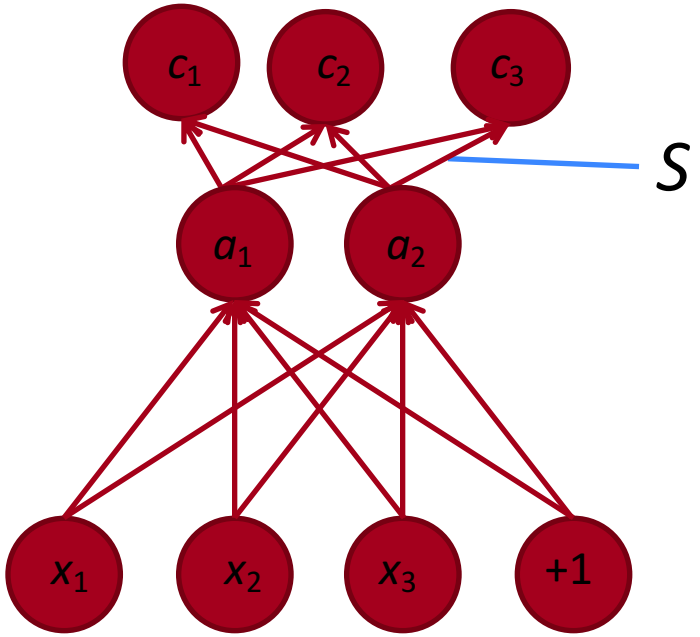
$$\hat{y} = softmax\left(W^{(S)} f(Wx + b)\right)$$



The model

- We already know the softmax classifier and how to optimize it
- The interesting twist in deep learning is that the input features x and their transformations in a hidden layer are also learned.
- Two final layers are possible:

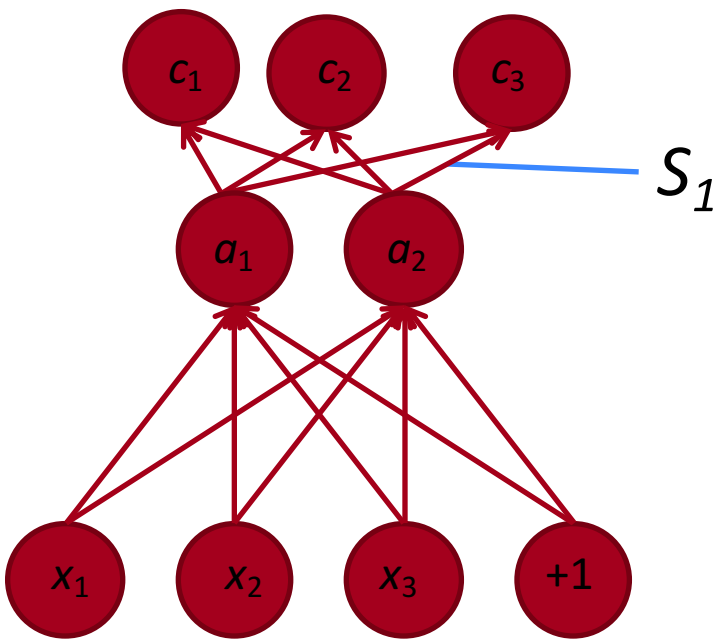


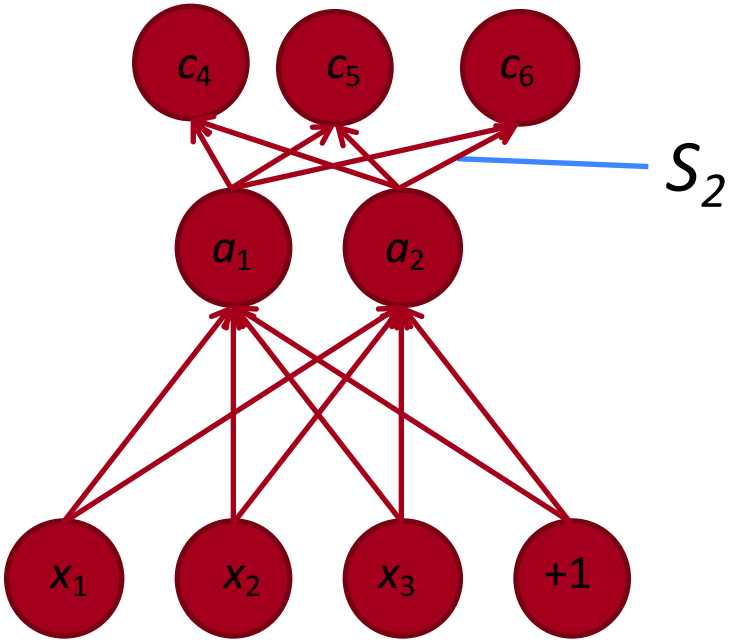


Multitask learning

- Main idea: We can share both the word vectors AND the hidden layer weights. Only the softmax weights are different.
- Cost function is just the sum of two cross entropy errors

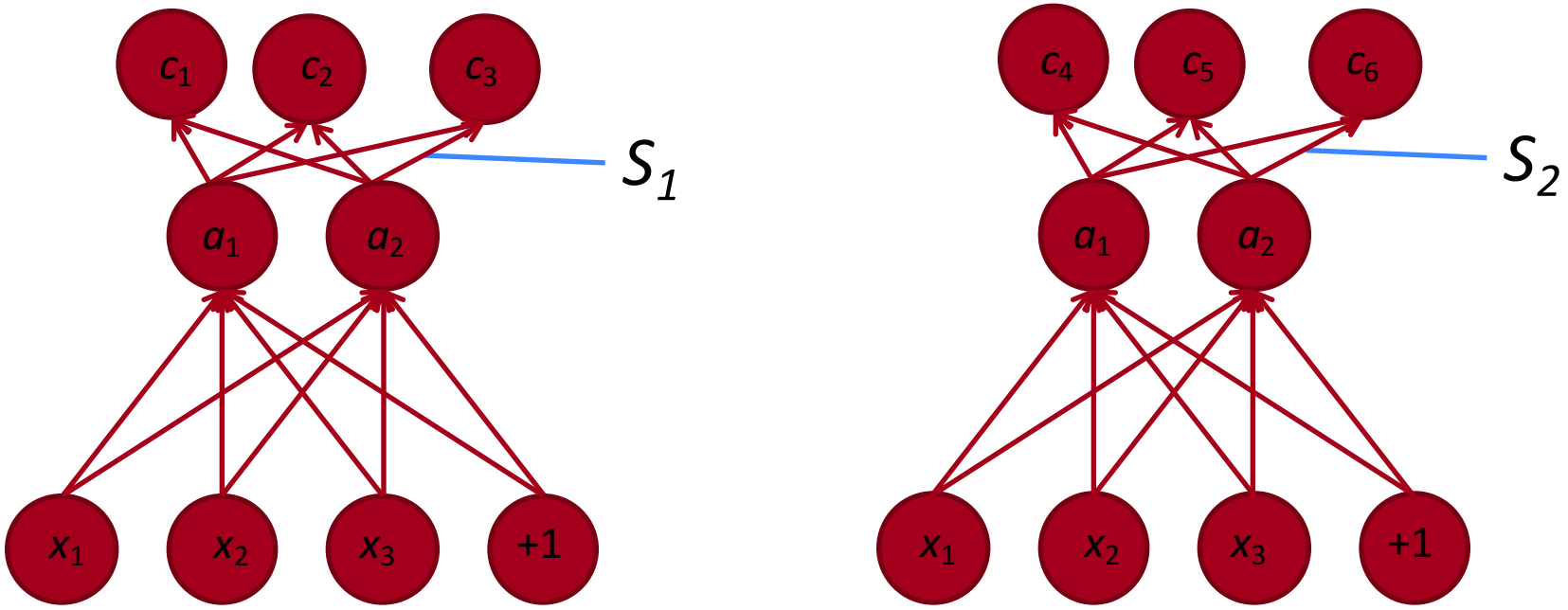
$$\hat{y}^{(1)} = softmax\left(W^{(S_1)} f(Wx + b)\right) \qquad \hat{y}^{(2)} = softmax\left(W^{(S_2)} f(Wx + b)\right)$$





The multitask model - Training

- Example: predict each window's center NER tag and POS tag: (example POS tags: DT, NN, NNP, JJ, JJS (superlative adj), VB,…)
- Efficient implementation: Same forward prop, compute errors on hidden vectors and add $\delta^{total} = \delta^{NER} + \delta^{POS}$



The secret sauce (sometimes) is the unsupervised word vector pre-training on a large text collection

	POS WSJ (acc.)	NER CoNLL (F1)
State-of-the-art*	97.24	89.31
Supervised NN	96.37	81.47
Word vector pre-training followed by supervised NN**	97.20	88.87
+ hand-crafted features***	97.29	89.59

* Representative systems: POS: (Toutanova et al. 2003), NER: (Ando & Zhang 2005)

** 130,000-word embedding trained on Wikipedia and Reuters with 11 word window, 100 unit hidden layer – then supervised task training

***Features are character suffixes for POS and a gazetteer for NER

Supervised refinement of the unsupervised word representation helps

	POS WSJ (acc.)	NER CoNLL (F1)
Supervised NN	96.37	81.47
NN with Brown clusters	96.92	87.15
Fixed embeddings*	**97.10**	**88.87**
C&W 2011**	**97.29**	**89.59**

* Same architecture as C&W 2011, but word embeddings are kept constant during the supervised training phase

** C&W is unsupervised pre-train + supervised NN + features model of last slide

General Strategy

1. Select appropriate Network Structure
    1. Structure: Single words, fixed windows vs Recursive Sentence Based vs Bag of words
    2. Nonlinearity
2. Check for implementation bugs with gradient check
3. Parameter initialization
4. Optimization tricks
5. Check if the model is powerful enough to overfit
    1. If not, change model structure or make model “larger”
    2. If you can overfit: Regularize

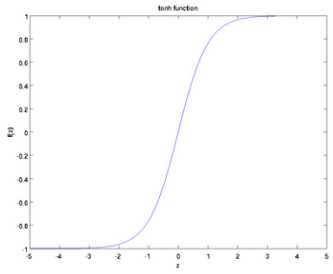
Parameter Initialization

- Initialize hidden layer biases to 0 and output (or reconstruction) biases to optimal value if weights were 0 (e.g., mean target or inverse sigmoid of mean target).

- Initialize weights ~ Uniform(−*r*, *r*), *r* inversely proportional to fan-in (previous layer size) and fan-out (next layer size):

$$\sqrt{6/(\text{fan-in} + \text{fan-out})}$$

for tanh units, and 4x bigger for sigmoid units [Glorot AISTATS 2010]



Stochastic Gradient Descent (SGD)

- Gradient descent uses total gradient over all examples per update, SGD updates after only 1 or few examples:

$$\theta^{new} = \theta^{old} - \alpha \nabla_\theta J_t(\theta)$$

- J_t = loss function at current example, μ = parameter vector, ® = learning rate.
- Ordinary gradient descent as a batch method is very slow, should never be used. Use 2nd order batch method such as L-BFGS.
- On large datasets, SGD usually wins over all batch methods. On smaller datasets L-BFGS or Conjugate Gradients win. Large-batch L-BFGS extends the reach of L-BFGS [Le et al. ICML 2011].

Mini-batch Stochastic Gradient Descent (SGD)

- Gradient descent uses total gradient over all examples per update, SGD updates after only 1 example
- Most commonly used now: Mini batches

- Size of each mini batch B: 20 to 1000:

$$\theta^{new} = \theta^{old} - \alpha \nabla_{\theta} J_{t:t+B}(\theta)$$

- Helps parallelizing any model by computing gradients for multiple elements of the batch in parallel

Improvement over SGD: Momentum

- Idea: Add a fraction v of previous update to current one
- When the gradient keeps pointing in the same direction, this will increase the size of the steps taken towards the minimum
- Reduce global learning rate ® when using a lot of momentum

- Update rule:

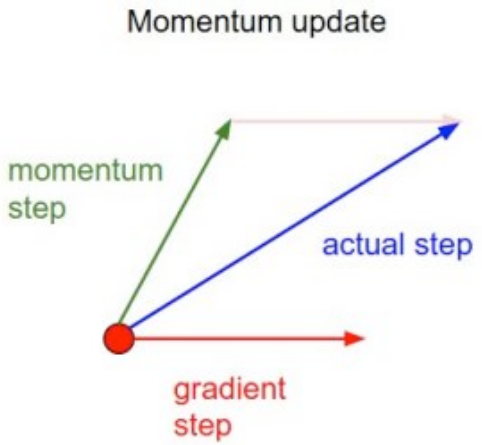
$$v = \mu v - \alpha \nabla_\theta J_t(\theta)$$

$$\theta^{new} = \theta^{old} + v$$

- v is initialized at 0
- Common: ^{1} = 0.9
- Momentum often increased after some epochs (0.5 → 0.99)

Intuition Momentum

- Adds friction (*momentum ~ misnomer)*



- Parameters build up velocity in direction of consistent gradient

- Simple convex function optimization dynamics
  without momentum with momentum:

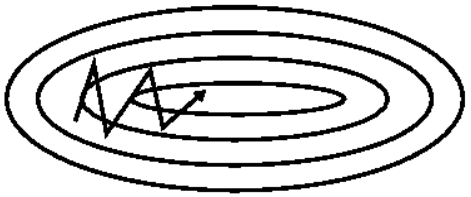


Figure from https://www.willamette.edu/~gorr/classes/cs449/momrate.html

Learning Rates

- Simplest recipe: keep it fixed and use the same for all parameters. Standard: $\theta^{new} = \theta^{old} - \alpha \nabla_\theta J_t(\theta)$

- Better results by allowing learning rates to decrease Options:
  - Reduce by 0.5 when validation error stops improving
  - Reduction by $O(1/t)$ because of theoretical convergence guarantees, e.g.: $\alpha = \frac{\alpha_0 \tau}{\max(t, \tau)}$
    with hyper-parameters ε_0 and τ and t is iteration numbers
  - Better yet: No hand-set learning of rates by using AdaGrad →

Adagrad

- Adaptive learning rates for each parameter!
- Related paper: Adaptive Subgradient Methods for Online Learning and Stochastic Optimization, Duchi et al. 2010

- Learning rate is adapting differently for each parameter and rare parameters get larger updates than frequently occurring parameters. Word vectors!

- Let $g_{t,i} = \frac{\partial}{\partial \theta_i^t} J_t(\theta)$, then: $\theta_{t,i} = \theta_{t-1,i} - \frac{\alpha}{\sqrt{\sum_{\tau=1}^{t} g_{\tau,i}^2}} g_{t,i}$

General Strategy

1. Select appropriate Network Structure
    1. Structure: Single words, fixed windows vs Recursive Sentence Based vs Bag of words
    2. Nonlinearity
2. Check for implementation bugs with gradient check
3. Parameter initialization
4. Optimization tricks
5. Check if the model is powerful enough to overfit
    1. If not, change model structure or make model “larger”
    2. If you can overfit: Regularize

Assuming you found the right network structure, implemented it correctly, optimize it properly and you can make your model overfit on your training data.

Now, it’s time to regularize

Prevent Feature Co-adaptation

Dropout (Hinton et al. 2012)

- Training time: at each instance of evaluation (in online SGD-training), randomly set 50% of the inputs to each neuron to 0
- Test time: halve the model weights (now twice as many)
- This prevents feature co-adaptation: A feature cannot only be useful in the presence of particular other features
- In a single layer: A kind of middle-ground between Naïve Bayes (where all feature weights are set independently) and logistic regression models (where weights are set in the context of all others)
- Can be thought of as a form of model bagging
- It also acts as a strong regularizer

Deep Learning Tricks of the Trade

- Y. Bengio (2012), "Practical Recommendations for Gradient-Based Training of Deep Architectures"

  - Unsupervised pre-training
  - Stochastic gradient descent and setting learning rates
  - Main hyper-parameters
    - Learning rate schedule & early stopping, Minibatches, Parameter initialization, Number of hidden units, regularization (= weight decay)
  - How to efficiently search for hyper-parameter configurations
    - Short answer: **Random hyperparameter search** (!)

- Some more advanced and recent tricks in later lectures

Language Models

Language Models

A language model computes a probability for a sequence of words: $P(w_1, \ldots, w_T)$

Probability is usually conditioned on window of n previous words :

$$P(w_1, \ldots, w_m) = \prod_{i=1}^{m} P(w_i \mid w_1, \ldots, w_{i-1}) \approx \prod_{i=1}^{m} P(w_i \mid w_{i-(n-1)}, \ldots, w_{i-1})$$

Very useful for a lot of tasks:

Can be used to determine whether a sequence is a good / grammatical translation or speech utterance

Original neural language model

A Neural Probabilistic Language Model, Bengio et al. 2003

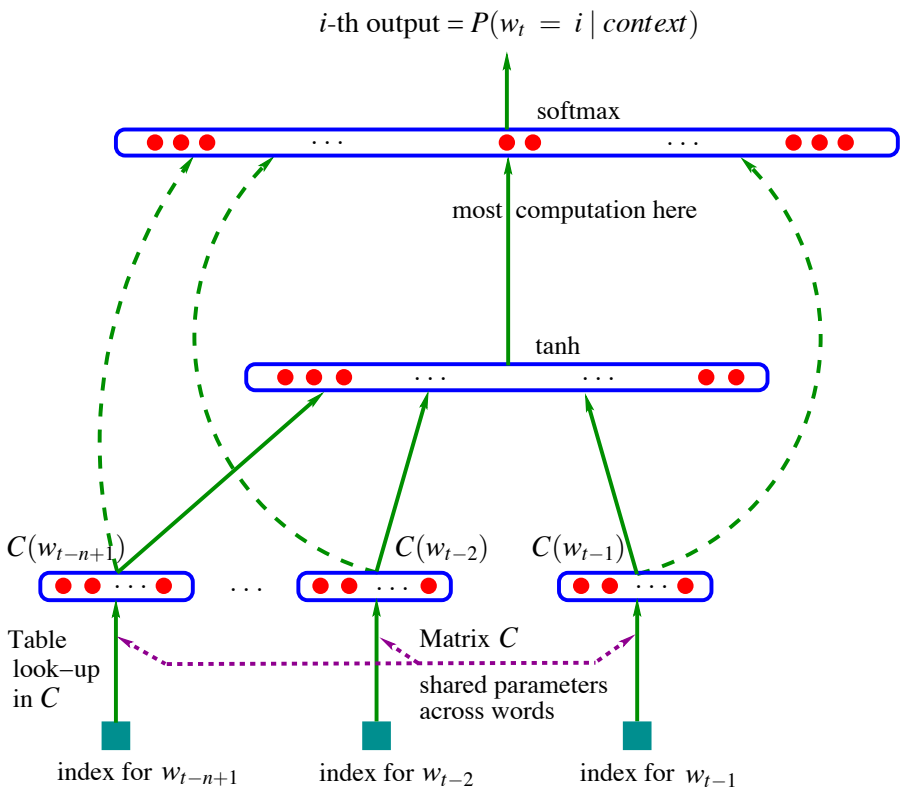
$$\hat{y} = softmax\left(W^{(2)} f\left(W^{(1)} x + b^{(1)}\right) + W^{(3)} x + b^{(3)}\right)$$

Original equations:

$$y = b + Wx + U\tanh(d + Hx)$$

$$\hat{P}(w_t | w_{t-1}, \cdots w_{t-n+1}) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}}.$$

Problem: Fixed window of context for conditioning :(



Recurrent Neural Networks

Recurrent Neural Network language model

Main idea: we use the same set of W weights at all time steps!

Everything else is the same:

$$
\begin{aligned}
h_t &= \sigma\left(W^{(hh)} h_{t-1} + W^{(hx)} x_{[t]}\right) \\
\hat{y}_t &= \text{softmax}\left(W^{(S)} h_t\right) \\
\hat{P}(x_{t+1} = v_j \mid x_t, \ldots, x_1) &= \hat{y}_{t,j}
\end{aligned}
$$

$h_0 \in \mathbb{R}^{D_h}$ is some initialization vector for the hidden layer at time step 0

$x_{[t]}$ is the column vector of L at index [t] at time step t

$W^{(hh)} \in \mathbb{R}^{D_h \times D_h}$ $W^{(hx)} \in \mathbb{R}^{D_h \times d}$ $W^{(S)} \in \mathbb{R}^{|V| \times D_h}$

Recurrent Neural Network language model

$\hat{y} \in \mathbb{R}^{|V|}$ is a probability distribution over the vocabulary

Same cross entropy loss function but predicting words instead of classes

$$J^{(t)}(\theta) = -\sum_{j=1}^{|V|} y_{t,j} \log \hat{y}_{t,j}$$

Recurrent Neural Network language model

Evaluation could just be negative of average log probability over dataset of size (number of words) T:

$$J = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{|V|} y_{t,j} \log \hat{y}_{t,j}$$

But more commonly: Perplexity: 2^J

Lower is better!

Training RNNs is hard

- The gradient is a product of Jacobian matrices, each associated with a step in the forward computation.

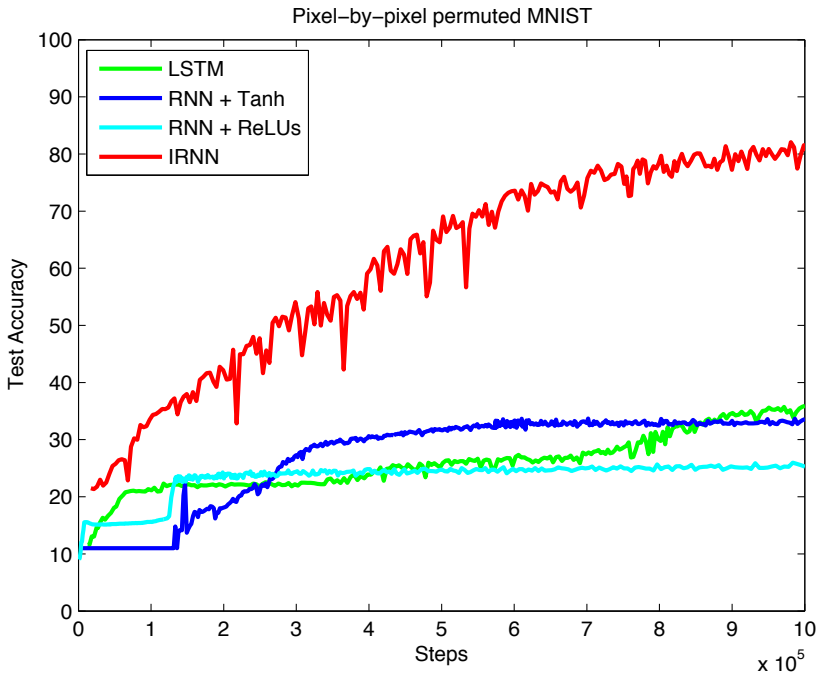
- Multiply the same matrix at each time step during backprop

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

- This can become very small or very large quickly [Bengio et al 1994], and the locality assumption of gradient descent breaks down. → **Vanishing or exploding gradient**

Initialization trick for RNNs!

- Initialize $W^{(hh)}$ to be the identity matrix I and
  f(z) = $\text{rect}(z) = \max(z, 0)$
- → Huge difference!



- Initialization idea first introduced in *Parsing with Compositional Vector Grammars*, Socher et al. 2013
- New experiments with recurrent neural nets last week (!) in *A Simple Way to Initialize Recurrent Networks of Rectified Linear* Units, Le et al. 2015

Long-Term dependencies and clipping trick

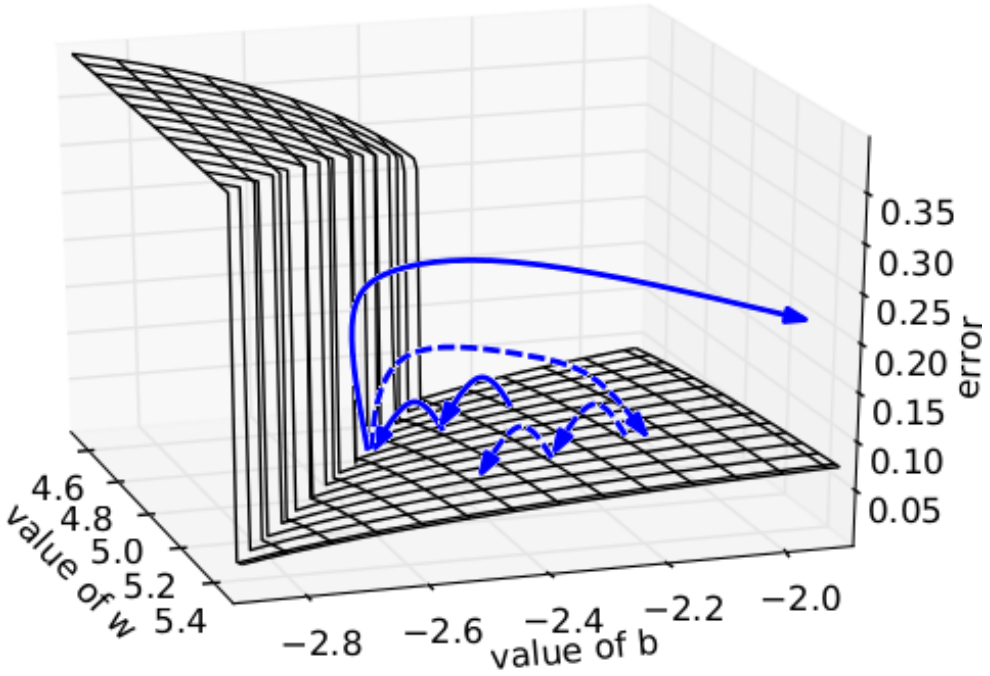
- The solution first introduced by Mikolov is to clip gradients to a maximum value.

Algorithm 1 Pseudo-code for norm clipping the gradients whenever they explode

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$
end if

- Makes a big difference in RNNs.

Gradient clipping intuition



On the difficulty of training Recurrent Neural Networks, Pascanu et al. 2013

- Error surface of a single hidden unit RNN,
- High curvature walls
- Solid lines: standard gradient descent trajectories
- Dashed lines gradients rescaled to fixed size

Summary

Tips and tricks to become a deep neural net ninjia

Introduction to Recurrent Neural Network

Next week:

→ More RNN details and variants (LSTMs and GRUs)

→ Exciting times!