Natural Language Processing with Deep Learning

CS224N/Ling284

Lecture 4: Word Window Classification and Neural Networks

Richard Socher

Organization

- Main midterm: Feb 13
- Alternative midterm: Friday Feb 9 – tough choice

- Final project and PSet4 can both have groups of 3.
- Highly discouraged (especially for PSet4) and need more results so higher pressure for you to coordinate and deliver.

- Python Review session tomorrow (Friday) 3-4:20pm at nVidia auditorium
- Coding session: Next Monday, 6-9pm

- Project website is up

Overview Today:

- Classification background
- Updating word vectors for classification
- Window classification & cross entropy error derivation tips
- A single layer neural network!
- Max-Margin loss and **backprop**

- This will be a tough lecture for some! → OH

Classification setup and notation

- Generally we have a training dataset consisting of samples

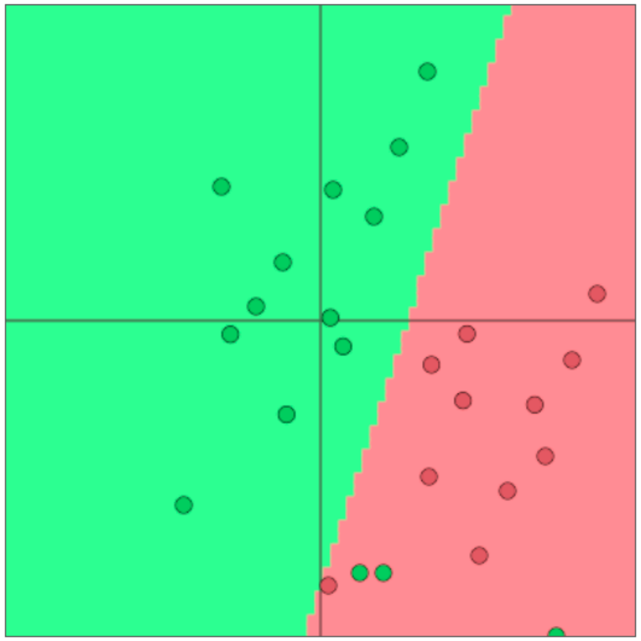
$$\{x_i, y_i\}^N_{i=1}$$

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

- y_i are labels (one of *C* classes) we try to predict, for example:
  - classes: sentiment, named entities, buy/sell decision
  - other words
  - later: multi-word sequences

Classification intuition

- Training data: $\{x_i, y_i\}_{i=1}^{N}$
- 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 approach: assume x_i are fixed, train logistic regression weights $W \in \mathbb{R}^{C \times d}$ (only modifies the decision boundary)

- Goal: For each x, predict:

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

Details of the softmax classifier

$$p(y|x) = \frac{\exp(W_{y\cdot}x)}{\sum_{c=1}^{C}\exp(W_{c\cdot}x)}$$

We can tease apart the prediction function into two steps:

1. Take the y'th row of W and multiply that row with x:

$$W_{y\cdot}x = \sum_{i=1}^{d} W_{yi}x_i = f_y$$

   Compute all f_c for c=1,…,C

2. Apply softmax function to get normalized probability:

$$p(y|x) = \frac{\exp(f_y)}{\sum_{c=1}^{C}\exp(f_c)} = softmax(f)_y$$

Training with softmax and cross-entropy error

- For each training example {x,y}, our objective is to maximize the probability of the correct class y

- Hence, we minimize the negative log probability of that class:

$$-\log p(y|x) = -\log\left(\frac{\exp(f_y)}{\sum_{c=1}^{C}\exp(f_c)}\right)$$

Background: Why “Cross entropy” error

- Assuming a ground truth (or gold or target) probability distribution that is 1 at the right class and 0 everywhere else: p = [0,…,0,1,0,…0] and our computed probability is q, then the cross entropy is:

$$H(p,q) = -\sum_{c=1}^{C} p(c) \log q(c)$$

- **Because of one-hot p, the only term left is the negative log probability of the true class**

Sidenote: The KL divergence

- Cross-entropy can be re-written in terms of the entropy and *Kullback-Leibler* divergence between the two distributions:

$$H(p,q) = H(p) + D_{KL}(p||q)$$

- Because H(p) is zero in our case (and even if it wasn't it would be fixed and have no contribution to gradient), to minimize this is equal to minimizing the KL divergence between p and q

- The KL divergence is **not a distance** but a non-symmetric measure of the difference between two probability distributions *p* and *q*

$$D_{KL}(p||q) = \sum_{c=1}^{C} p(c) \log \frac{p(c)}{q(c)}$$

Classification over a full dataset

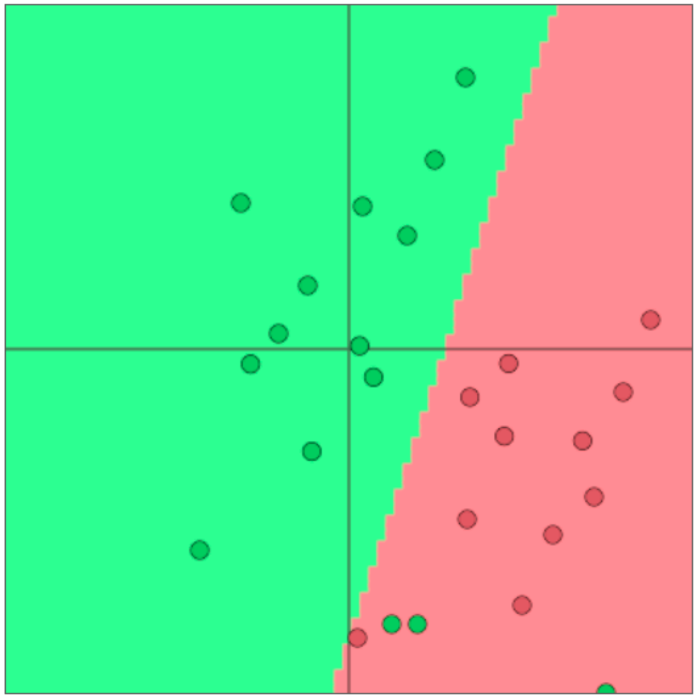
- Cross entropy loss function over full dataset $\{x_i, y_i\}^N_{i=1}$

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

- Instead of

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

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

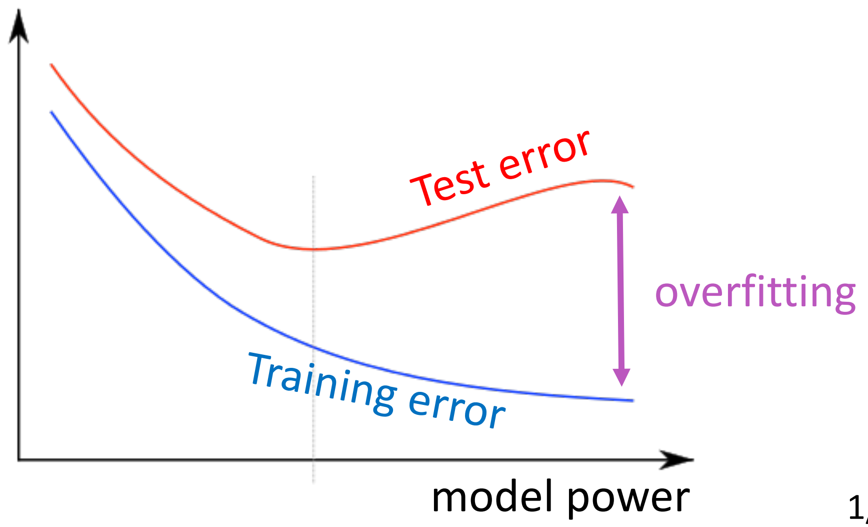


Classification: Regularization!

- Really full loss function in practice includes **regularization** over all parameters θ:

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

- Regularization prevents overfitting when we have a lot of features (or later a very powerful/deep model,++)



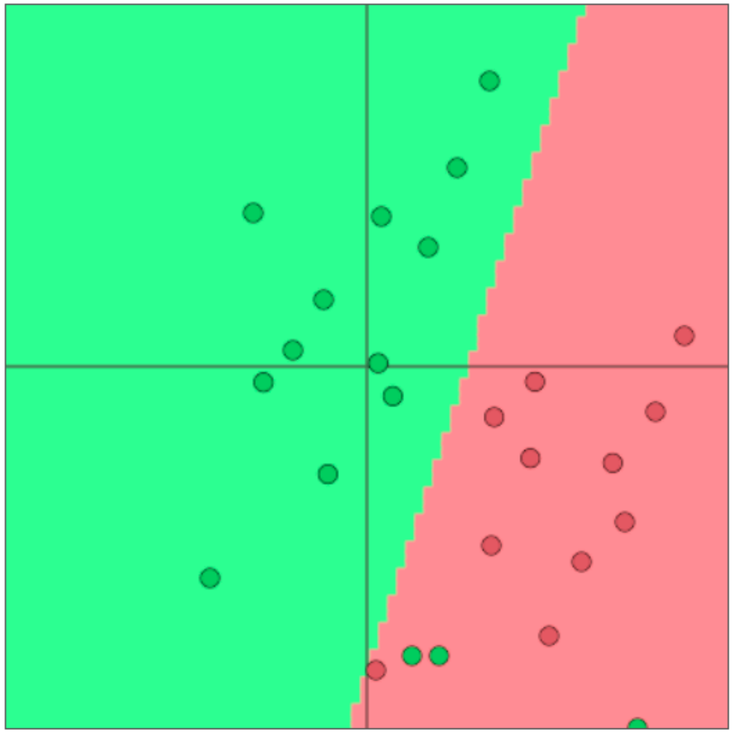
Details: General ML optimization

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

$$\theta = \left[\begin{array}{c} W_{\cdot 1} \\ \vdots \\ W_{\cdot d} \end{array} \right] = W(:) \in \mathbb{R}^{Cd}$$

- So we only update the decision boundary

$$\nabla_\theta J(\theta) = \left[\begin{array}{c} \nabla_{W_{\cdot 1}} \\ \vdots \\ \nabla_{W_{\cdot d}} \end{array} \right] \in \mathbb{R}^{Cd}$$



Visualizations with ConvNetJS by Karpathy

Classification difference with word vectors

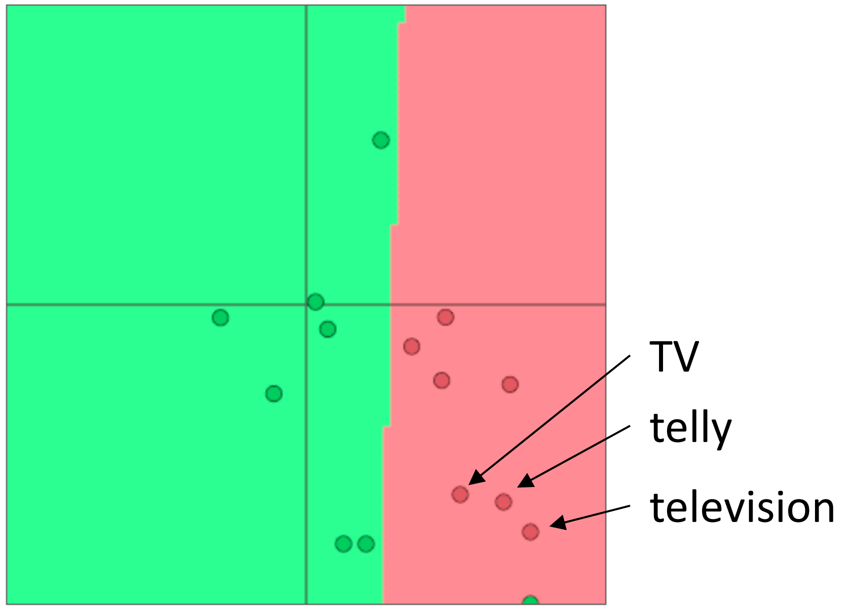
- Common in deep learning:
  - Learn both W and word vectors x

Very large!
Danger of overfitting!

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

A pitfall when retraining word vectors

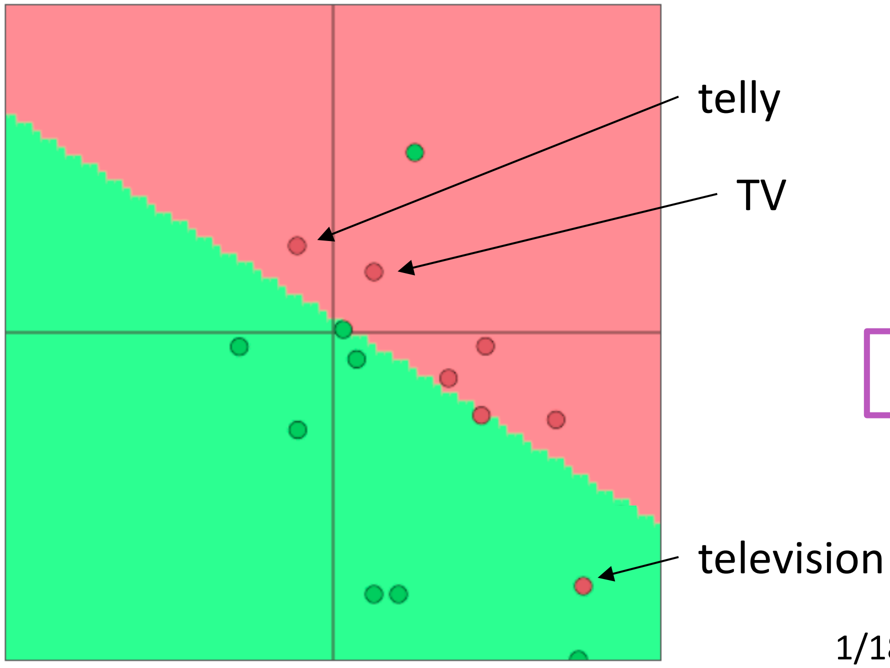
- Setting: We are training a logistic regression classification model for movie review sentiment using single words.
- In the training data we have “TV” and “telly”
- In the testing data we have “television”
- The pre-trained word vectors have all three similar:



- Question: **What happens when we retrain the word vectors?**

A pitfall when retraining word vectors

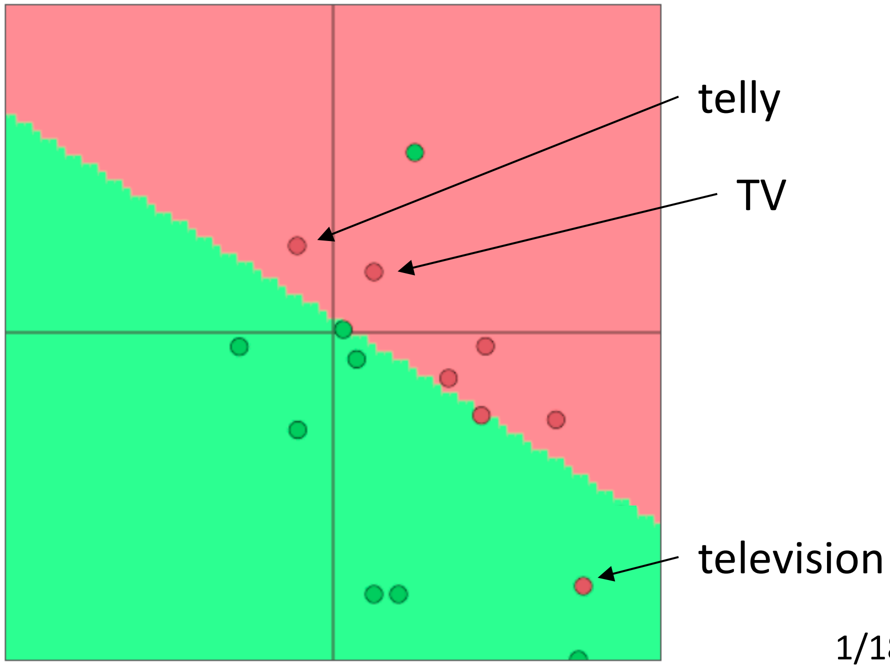
- Question: What happens when we train the word vectors?
- Answer:
  - Those that are in the training data move around
    - “TV” and “telly”
  - Words not in the training data stay
    - “television”



This is bad!

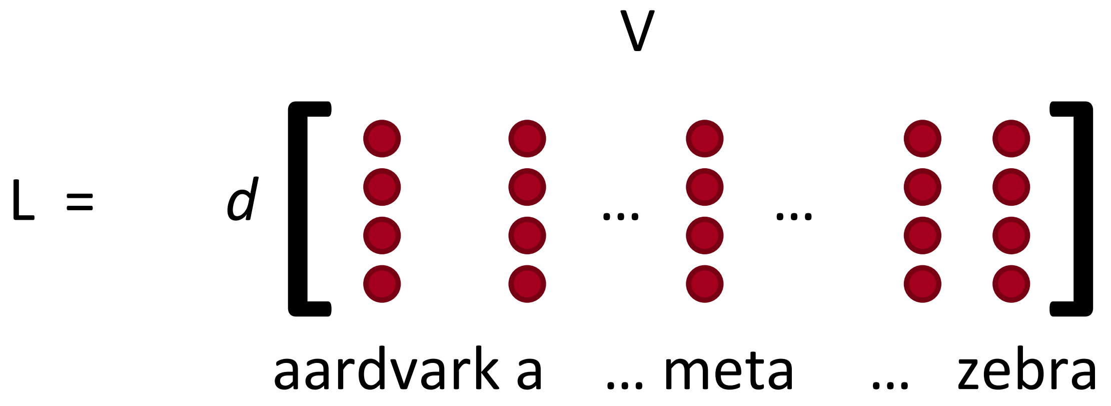
So what should I do?

- **Question**: Should I train my own word vectors?
- **Answer**:
  - 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.



Side note on word vectors notation

- The word vector matrix L is also called *lookup table*
- Word vectors = word embeddings = word representations (mostly)
- Usually obtained from methods like word2vec or Glove



- These are the word features x_{word} from now on

- New development (later in the class): character models :o

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.

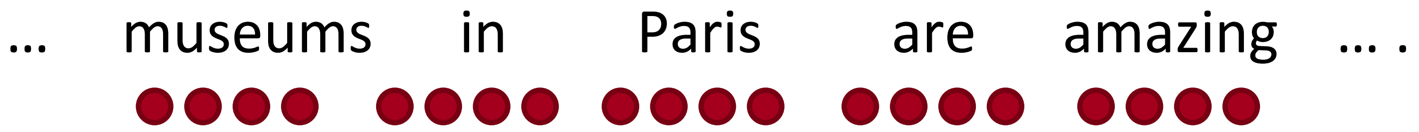
- For example, **Named Entity Recognition** is a 4-way classification task:
  - Person, Location, Organization, None

- There are many ways to classify a single word in context
  - For example: average all the words in a window
  - Problem: that would lose position information

Window classification

- Train softmax classifier to classify a center word by taking concatenation of all word vectors surrounding it

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



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

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

Simplest window classifier: Softmax

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

predicted model output probability

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

- With cross entropy error as before:

same

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

- But how do you update the word vectors?

Deriving gradients for window model

- Short answer: Just take derivatives as before
- Long answer: Let's go over steps together (helpful for PSet 1)
- 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 :)

Deriving gradients for window model

- **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**: Chain rule! If *y* = *f*(*u*) and *u* = *g*(*x*), i.e. *y* = *f*(*g*(x)), then:

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx} = \frac{df(u)}{du}\frac{dg(x)}{dx}$$

- Example repeated:

$$\boxed{\begin{aligned} y &= f(u) = 5u^4 \\ \frac{dy}{du} &= 20u^3 \end{aligned}}$$

$$\boxed{\begin{aligned} u &= g(x) = x^3 + 7 \\ \frac{du}{dx} &= 3x^2 \end{aligned}}$$

$$\frac{dy}{dx} = 20(x^3+7)^3 . 3x^2$$

Deriving gradients for window model

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

$$\hat{y} \quad t \qquad W \in \mathbb{R}^{C \times 5d}$$

- Tip 2 continued: **Know thy chain rule**
- Don't forget which variables depend on what and that x appears inside all elements of f's

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

- **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$\neq$ y (all the incorrect classes)

Deriving gradients for window model

- **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:

$$\hat{y} \qquad t$$

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

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

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

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

Deriving gradients for window model

- **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}

$$\hat{y} \qquad t$$

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

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

- **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!

Deriving gradients for window model

- 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:

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

Deriving gradients for window model

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

- We have

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

Deriving gradients for window model

- 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

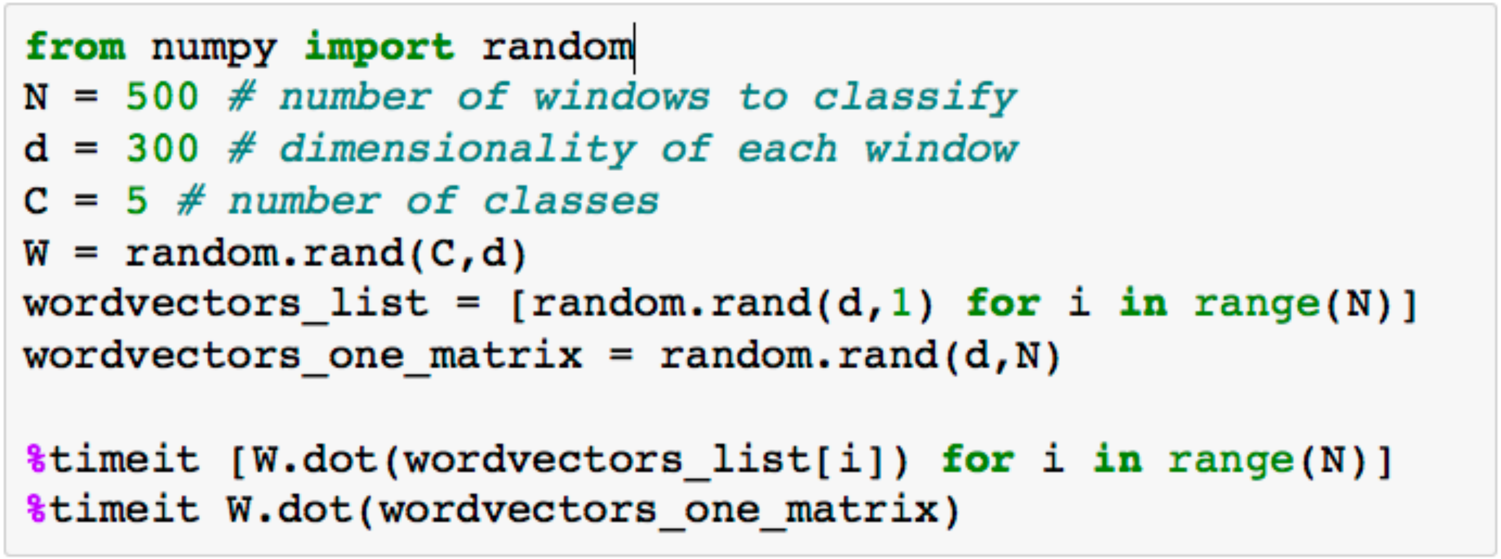
$$\nabla_\theta J(\theta) = \begin{bmatrix} \nabla_{W_{\cdot 1}} \\ \vdots \\ \nabla_{W_{\cdot d}} \\ \nabla_{x_{aardvark}} \\ \vdots \\ \nabla_{x_{zebra}} \end{bmatrix} \in \mathbb{R}^{Cd+Vd}$$

A note on matrix implementations

- There are two expensive operations in the softmax classifier:
  - The matrix multiplication $f = Wx$ and the exp
- A large matrix multiplication is always more efficient than a for loop!
  - Example code on next slide →

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

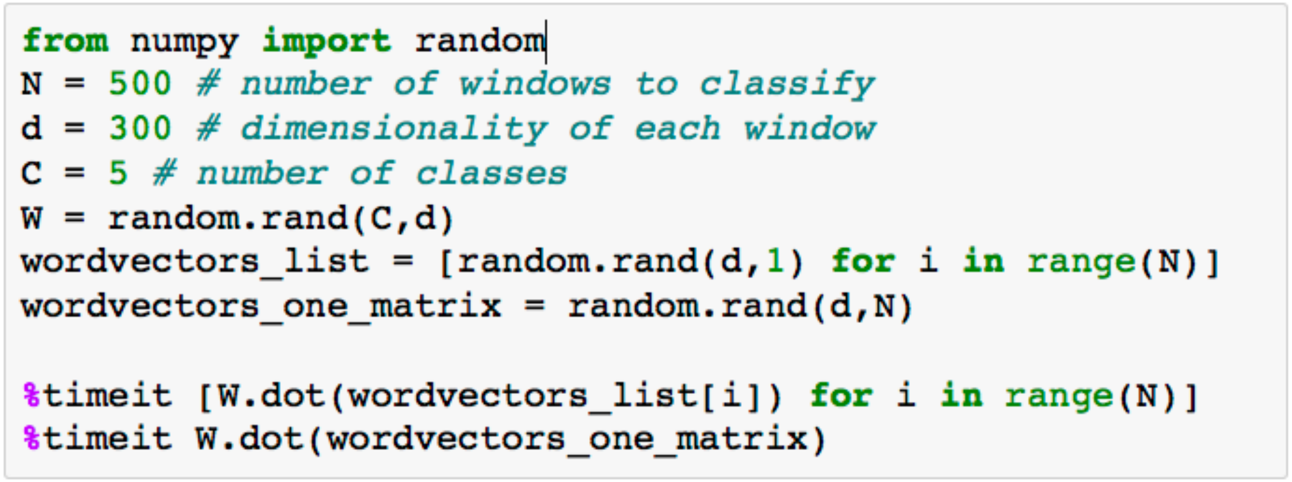


```
from numpy import random
N = 500 # number of windows to classify
d = 300 # dimensionality of each window
C = 5 # number of classes
W = random.rand(C,d)
wordvectors_list = [random.rand(d,1) for i in range(N)]
wordvectors_one_matrix = random.rand(d,N)

%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)
```

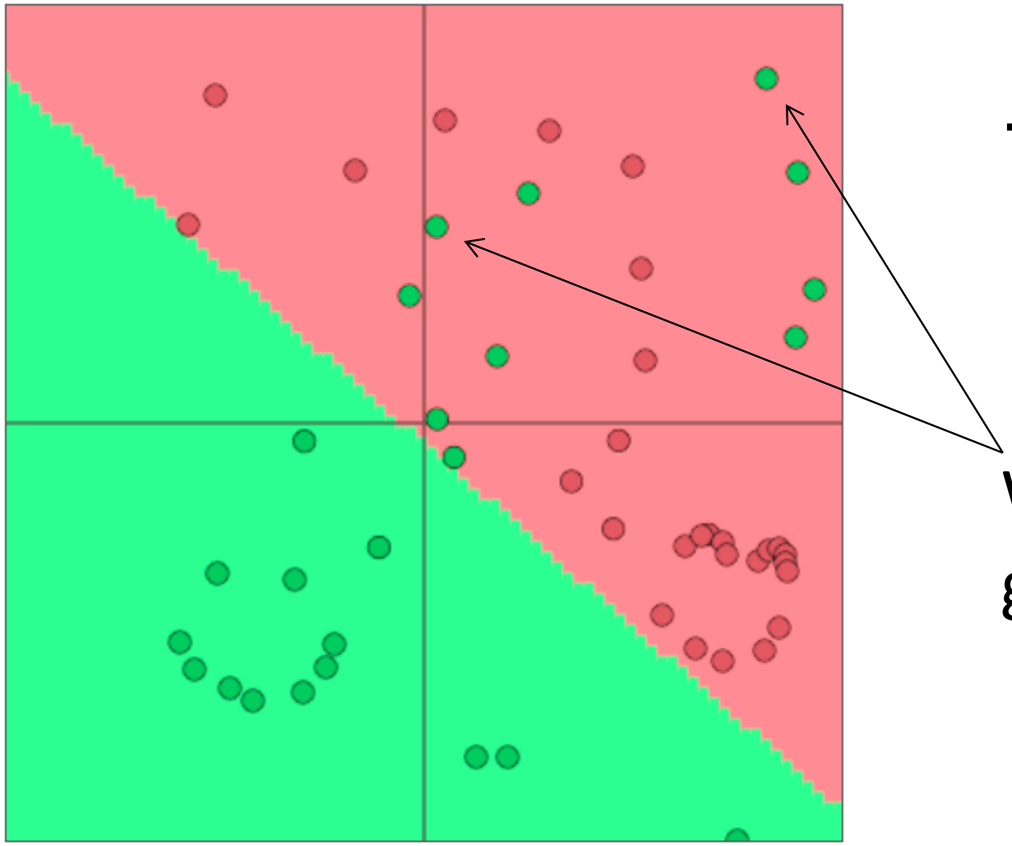
- Result of faster method is a C x N matrix:
  - Each column is an f(x) in our notation (unnormalized class scores)
- You should speed-test your code a lot too
- Tl;dr: Matrices are awesome! Matrix multiplication is better than for loop

Softmax (= logistic regression) alone 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

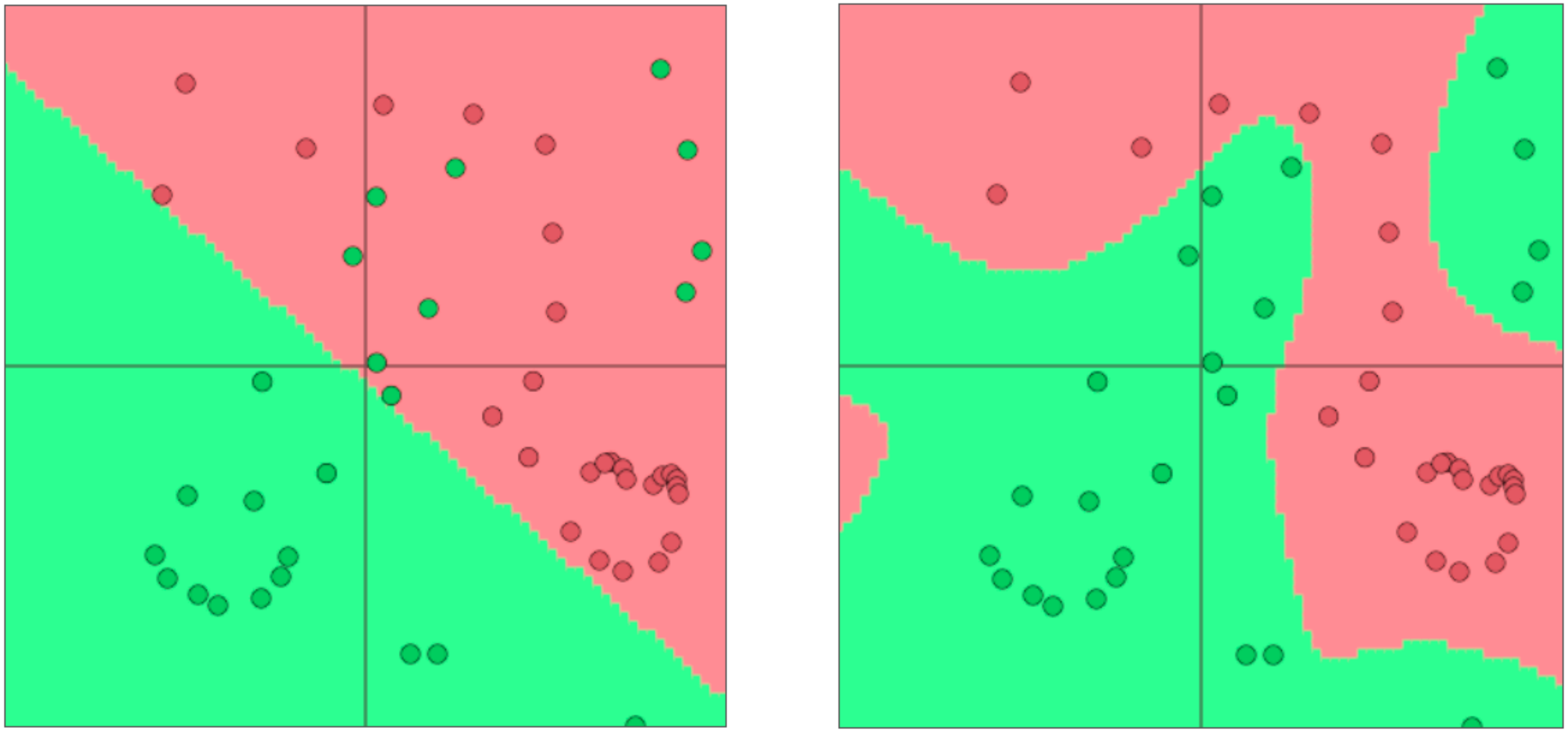


→ Unhelpful when problem is complex

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

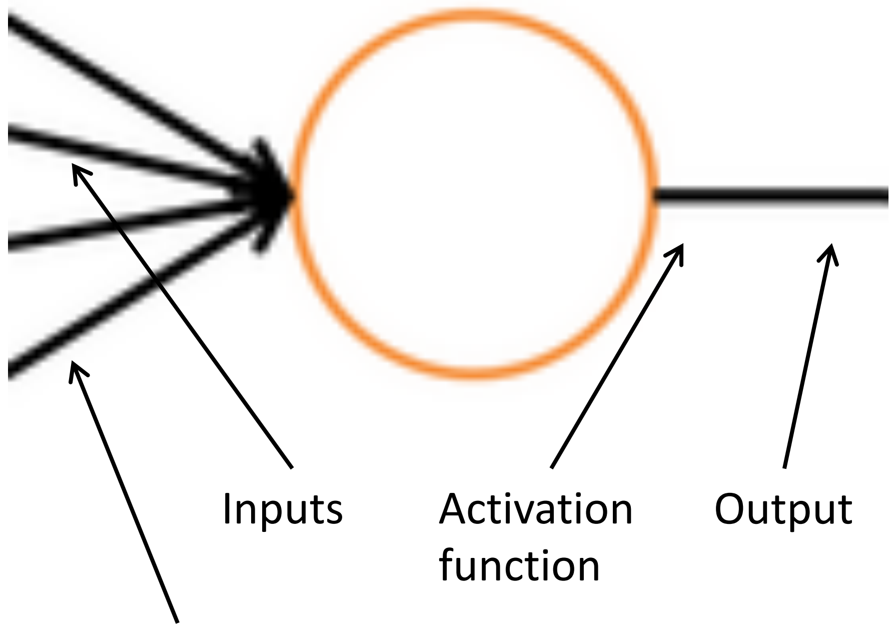
Demystifying neural networks

Neural networks come with their own terminological baggage

But if you understand how softmax models work

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

A single neuron
A computational unit with *n* (3) inputs and 1 output
and parameters *W, b*



Bias unit corresponds to intercept term

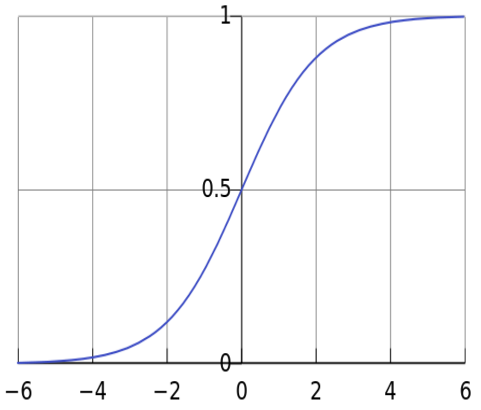
A neuron is essentially a binary logistic regression unit

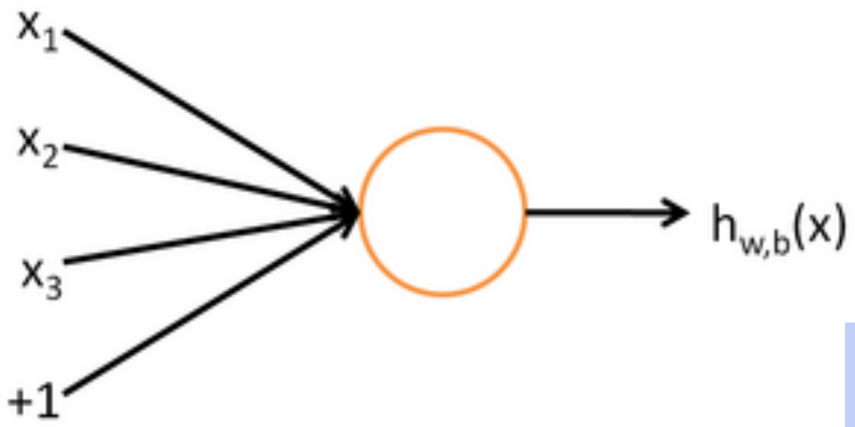
f = nonlinear activation fct. (e.g. sigmoid), w = weights, b = bias, h = hidden, x = inputs

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

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

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

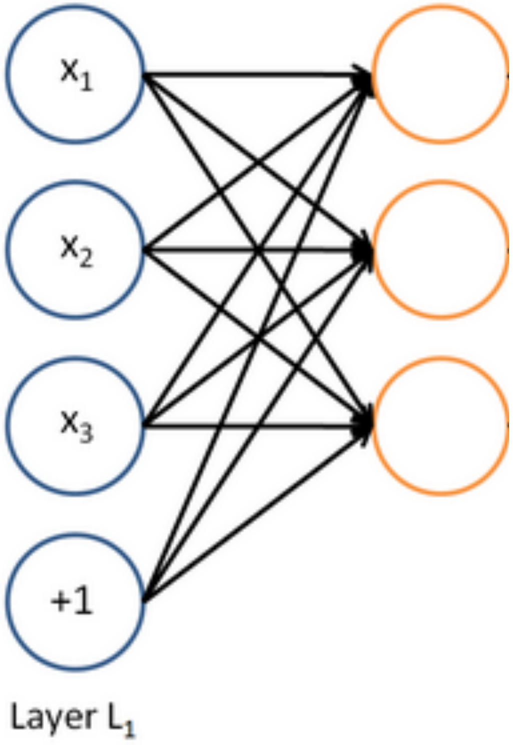




w, *b* are the parameters of this neuron i.e., this logistic regression model

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

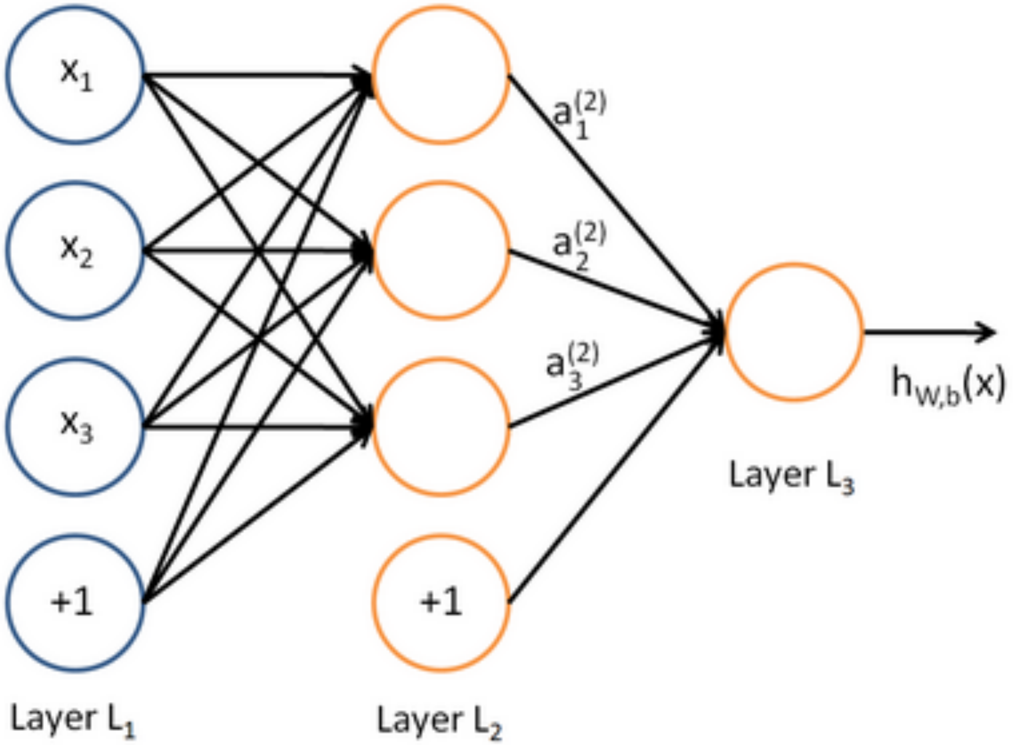
If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs …



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

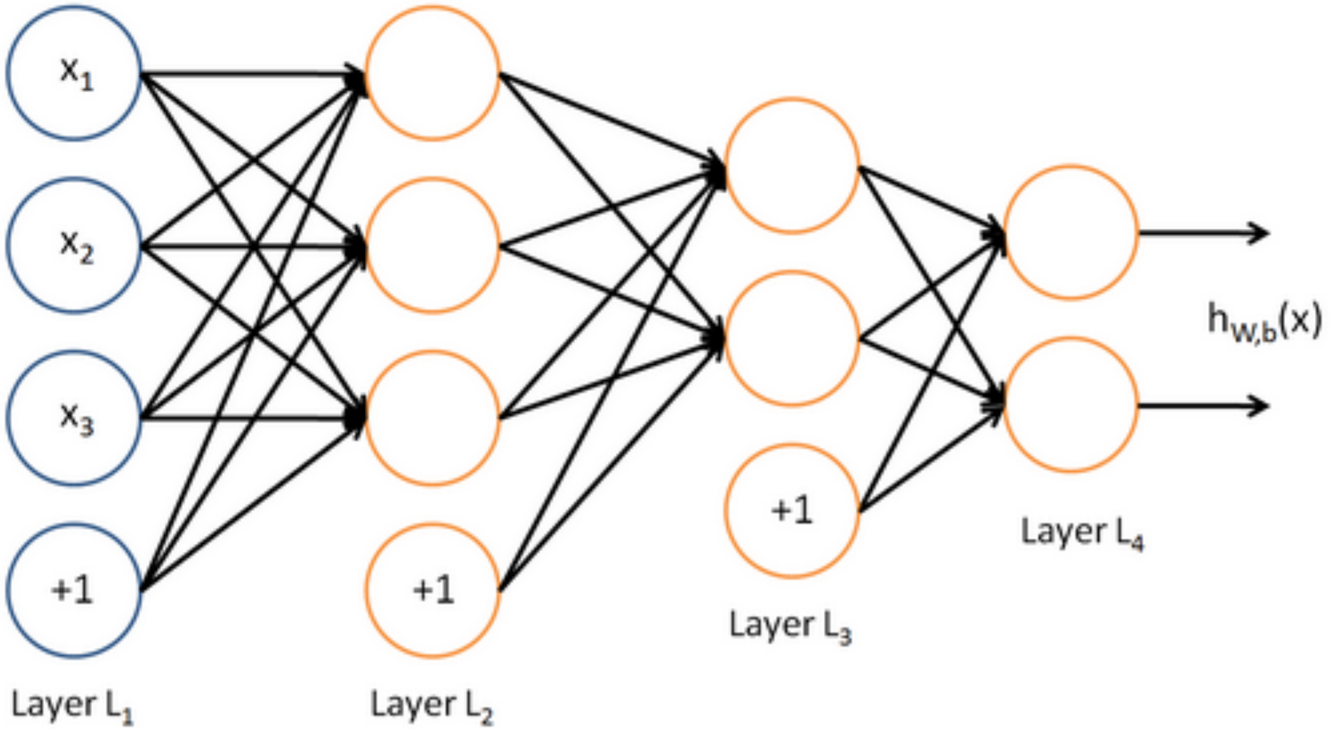
… which we can feed into another logistic regression function



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

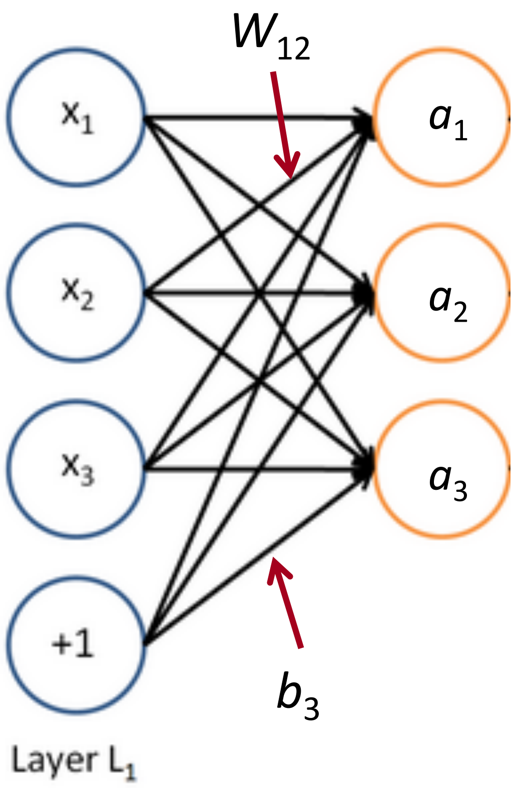
$$a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1)$$
$$a_2 = f(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2)$$
etc.

In matrix notation

$$z = Wx + b$$
$$a = f(z)$$

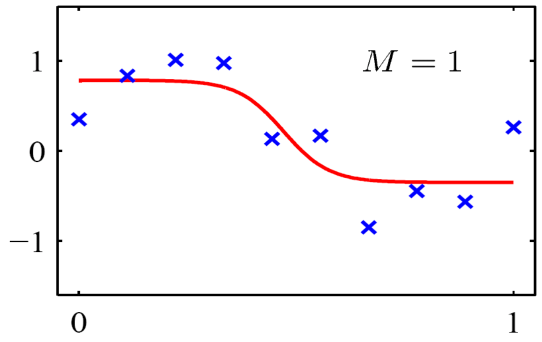
where *f* is applied element-wise:

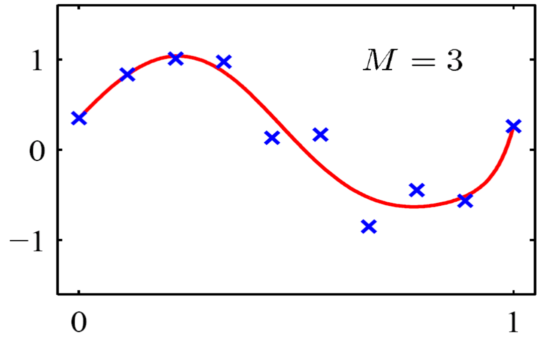
$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$

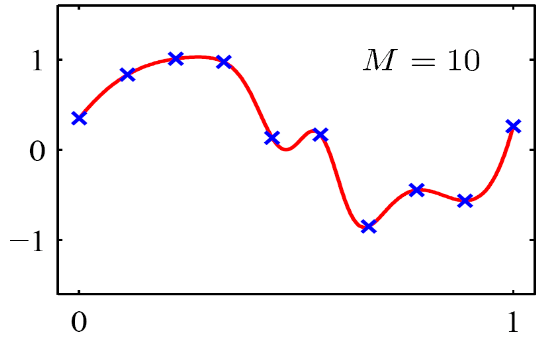


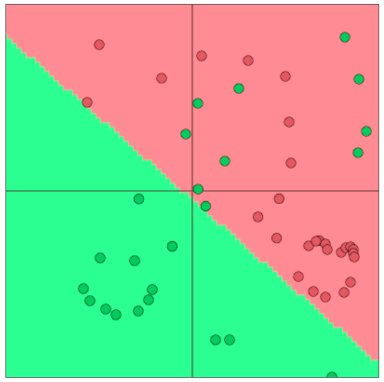
Non-linearities (aka “f”): Why they’re needed

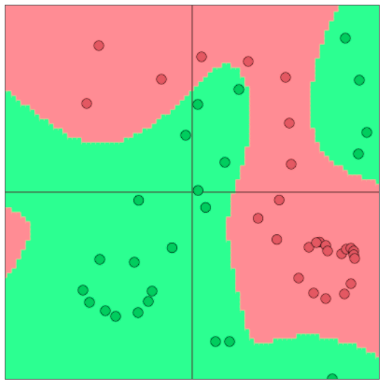
- Example: function approximation, e.g., regression or classification
  - Without non-linearities, deep neural networks can’t do anything more than a linear transform
  - Extra layers could just be compiled down into a single linear transform: $W_1 W_2 x = Wx$
  - With more layers, they can approximate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











Binary classification with unnormalized scores

- Revisiting our previous example:
  $X_{window} = [\ x_{museums} \quad x_{in} \quad x_{Paris} \quad x_{are} \quad x_{amazing}\ ]$
- Assume we want to classify whether the center word is a Location (Named Entity Recognition)

- Similar to word2vec, we will go over all positions in a corpus. But this time, it will be supervised and only some positions should get a high score.
- The positions that have an actual NER location in their center are called "true" positions.

Binary classification for NER

- Example: Not all museums in Paris are amazing.
- Here: one true window, the one with Paris in its center and all other windows are “corrupt” in terms of not having a named entity location in their center.
- “Corrupt“ windows are easy to find and there are many: Any window that isn’t specifically labeled as NER location in our corpus

A Single Layer Neural Network

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

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

- The neural activations *a* can then be used to compute some output.

  - For instance, a probability via softmax:

$$p(y|x) = \text{softmax}(Wa)$$

  - Or an unnormalized score (even simpler):

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

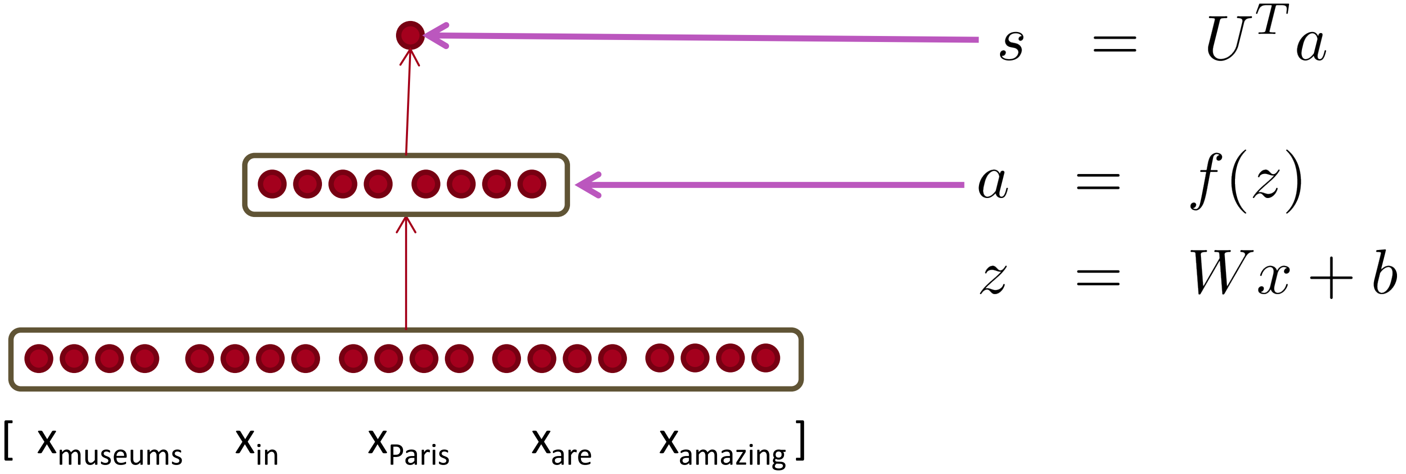
Summary: Feed-forward Computation

We compute a window's score with a 3-layer neural net:

- *s = score*("museums in Paris are amazing")

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

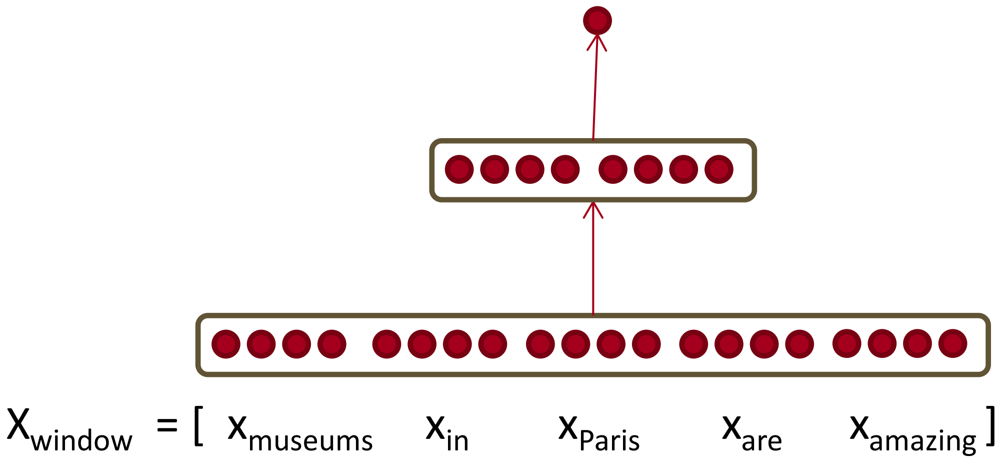
$$x \in \mathbb{R}^{20\times 1}, W \in \mathbb{R}^{8\times 20}, U \in \mathbb{R}^{8\times 1}$$



$s = U^T a$
$a = f(z)$
$z = Wx + b$
x_{window} = [$x_{museums}$ x_{in} x_{Paris} x_{are} $x_{amazing}$]

Main intuition for extra layer

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



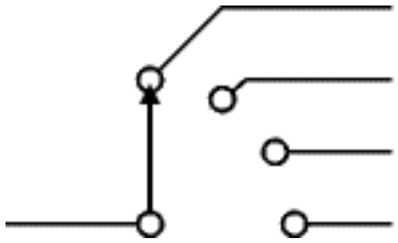
$X_{window} = [\ x_{museums}\quad x_{in}\quad x_{Paris}\quad x_{are}\quad x_{amazing}\ ]$

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

The max-margin loss

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

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



- This is not differentiable but it is continuous --> we can use SGD.

Each option is continuous

Max-margin loss

- Objective for a single window:

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

- Each window with an NER location at its center should have a score +1 higher than any window without a location at its center
- xxx $|\leftarrow$ 1 $\rightarrow|$ ooo

- For full objective function: Sample several corrupt windows per true one. Sum over all training windows.
- Similar to negative sampling in word2vec

Deriving gradients for backprop

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

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

Assuming cost *J* is > 0,

compute the derivatives of *s* and s_c wrt all the involved variables: *U, W, b, x*

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

Deriving gradients for backprop

- For this function:

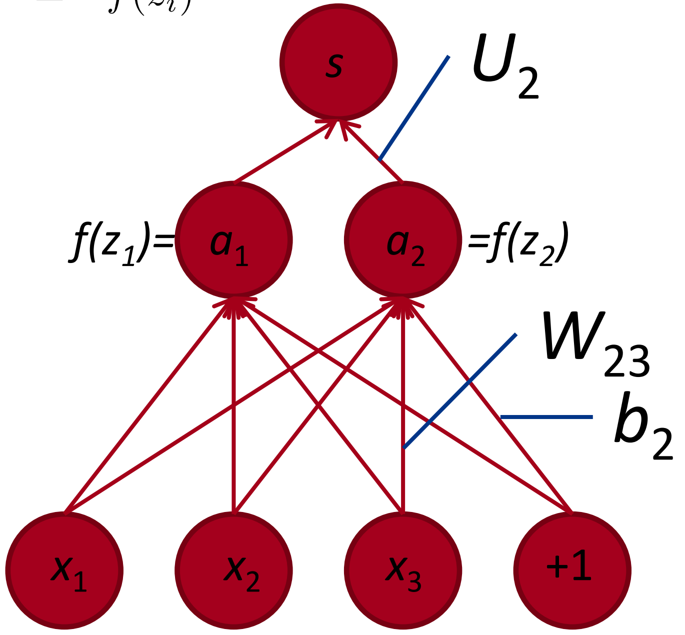
$$\frac{\partial s}{\partial W} = \frac{\partial}{\partial W} U^T a = \frac{\partial}{\partial W} U^T f(z) = \frac{\partial}{\partial W} U^T f(Wx + b)$$

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

- W_{ij} only appears inside a_i

- For example: W_{23} is only used to compute a_2 not a_1

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



Deriving gradients for backprop

Derivative of single weight W_{ij}:

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

Ignore constants in which W_{ij} doesn't appear

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

Pull out U_i since it's constant, apply chain rule

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

Apply definition of a

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

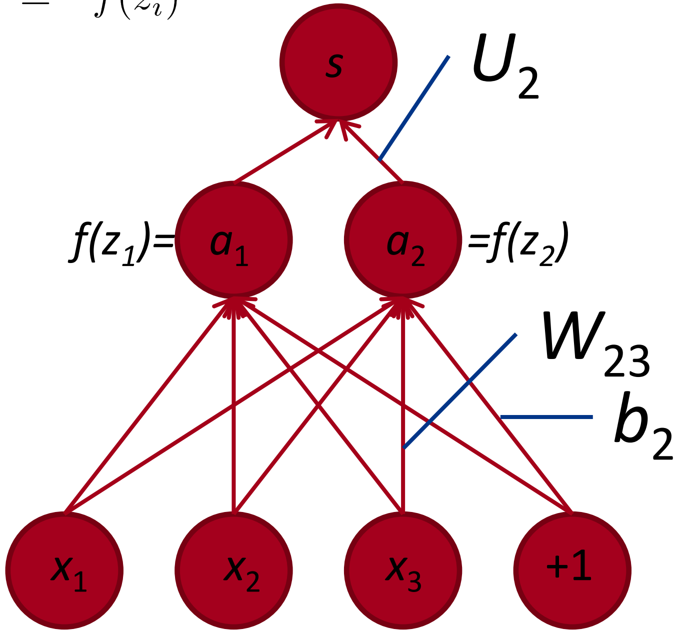
Just define derivative of f as f'

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

Plug in definition of z

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

$$a_i \quad = \quad f(z_i)$$



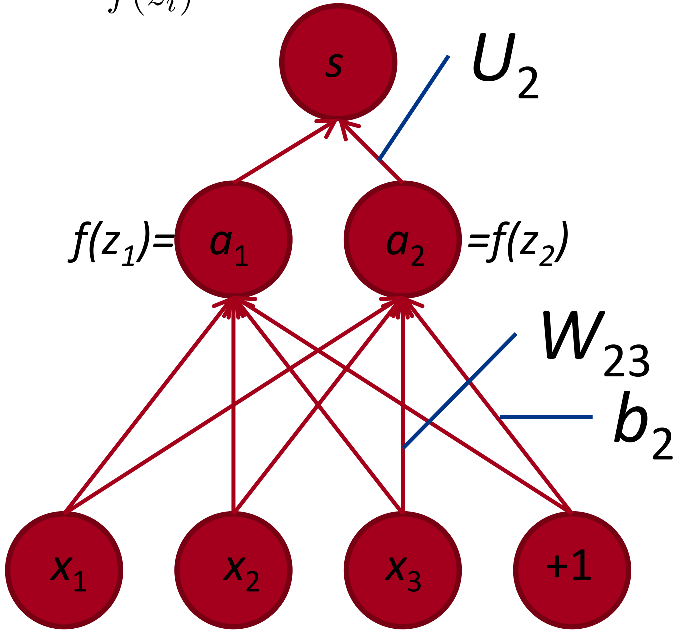
Deriving gradients for backprop

Derivative of single weight *W*$_{ij}$ continued:

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

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

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



Deriving gradients for backprop

- So far, derivative of single W_{ij} only , but we want gradient for full *W*.

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

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

- Solution: Outer product: $$\frac{\partial s}{\partial W} = \delta x^T$$

where $\delta \in \mathbb{R}^{2\times 1}$ is the "responsibility" or error signal coming from each activation *a*

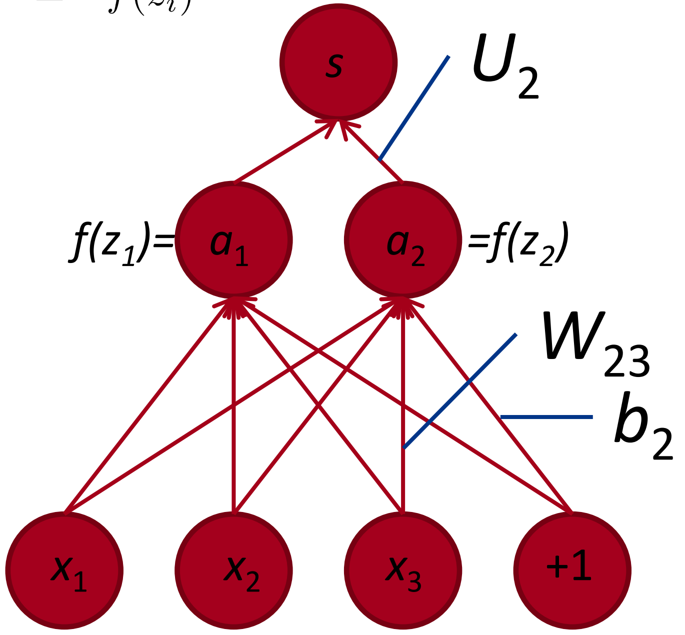
Deriving gradients for backprop

- How to derive gradients for biases b?

$$\frac{\partial s}{\partial\, b} = a$$

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

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



Training with Backpropagation

That's almost backpropagation

It's 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:

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

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 δ 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:

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

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

- For example: gradient for just U:

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

Summary

Congrats! Super useful basic components and real model

- Word vector training
- Windows
- Softmax and cross entropy error $\rightarrow$ PSet1
- Scores and max-margin loss
- Neural network $\rightarrow$ PSet1

Next lecture:

Taking more and **deeper derivatives** → Full **Backprop**

High level tips for easier derivations

Then we have all the basic tools in place to learn about and have fun with more complex and deeper models :)