Natural Language Processing with Deep Learning

CS224N/Ling284

Lecture 4: Word Window Classification and Neural Networks

Christopher Manning and **Richard Socher**

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 lecture will help a lot with PSet1 :)

Classification setup and notation

- Generally we have a training dataset consisting of samples

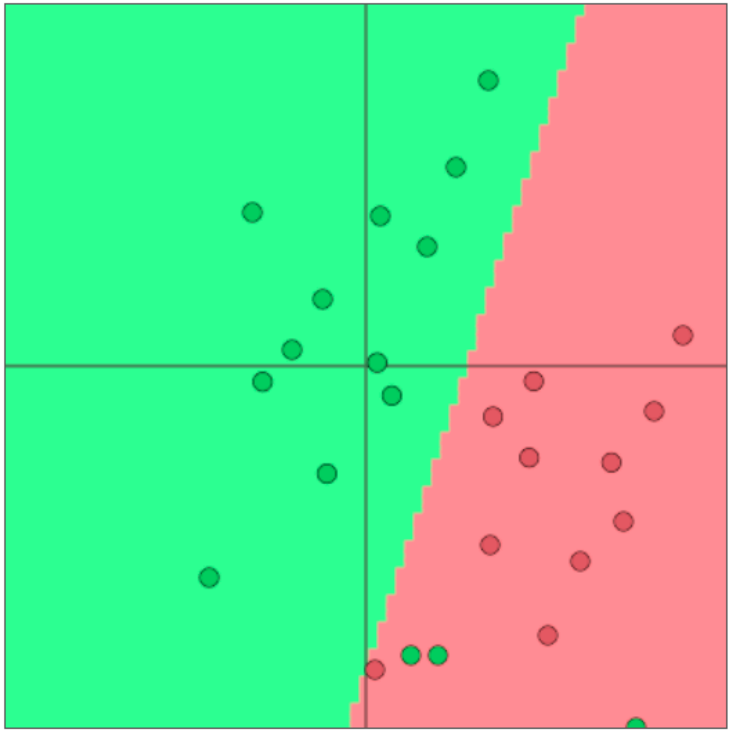
$$\{x_i, y_i\}^N_{i=1}$$

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

- y_i - labels we try to predict, for example
  - class: 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 →
- General ML: assume x is fixed,
  train logistic regression weights W
  → only modify the decision boundary
- Goal: predict for each x: $p(y|x) = \dfrac{\exp(W_y.x)}{\sum_{c=1}^{C} \exp(W_c.x)}$
  where $W \in \mathbb{R}^{C \times d}$



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

Details of the softmax

- We can tease apart into two steps:

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

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

$$W_y.x = \sum_{i=1}^{d} W_{yi} x_i = f_y$$

   Compute all f_c for c=1,…,C

2. Normalize to obtain probability with softmax function:

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

The 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

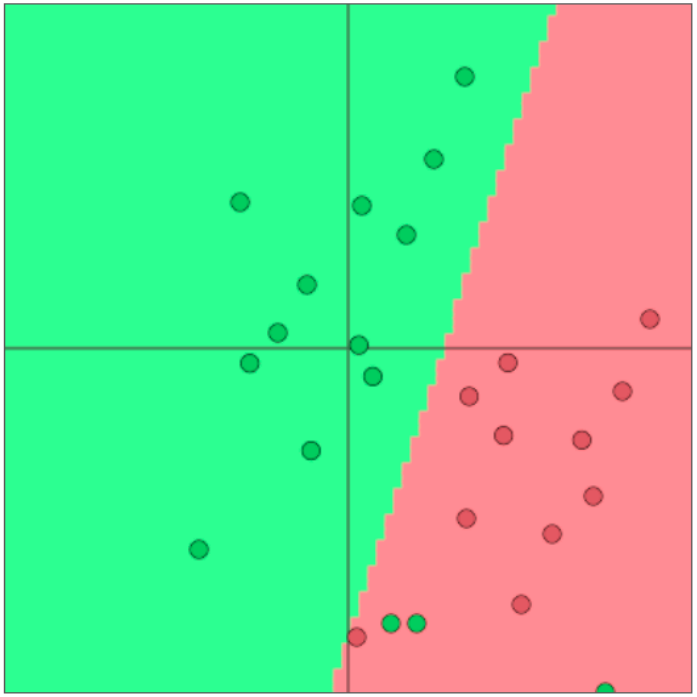
- Cross entropy loss function over full dataset $\{x_i, y_i\}^N_{i=1}$

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

- 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

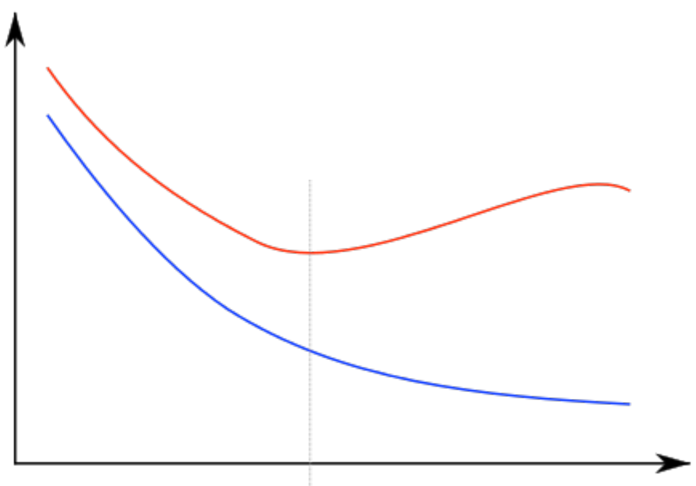


Classification: Regularization!

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

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

- 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



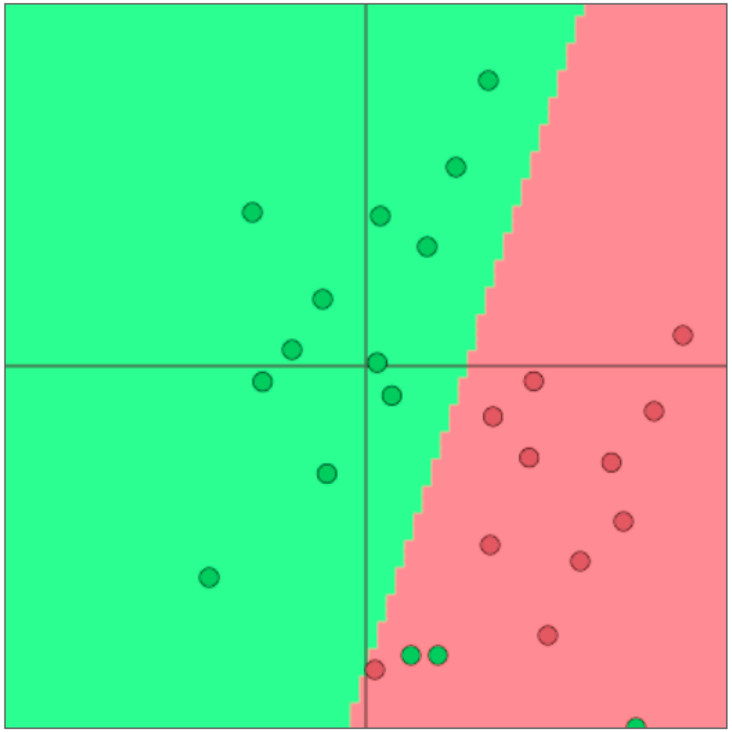
Details: General ML optimization

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

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

- So we only update the decision boundary

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



Visualizations with ConvNetJS by Karpathy

Classification difference with word vectors

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

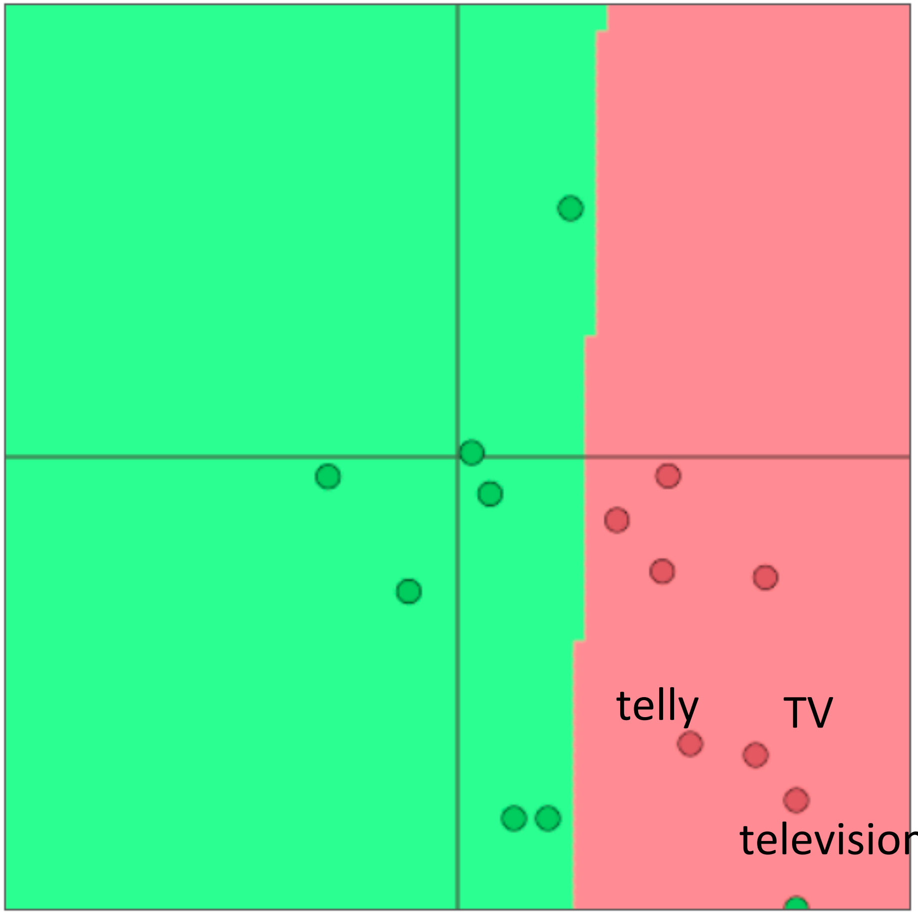
$$\nabla_\theta J(\theta) = \begin{bmatrix} \nabla_{W_{\cdot 1}} \\ \vdots \\ \nabla_{W_{\cdot d}} \\ \nabla_{x_{aardvark}} \\ \vdots \\ \nabla_{x_{zebra}} \end{bmatrix} \in \mathbb{R}^{Cd + \boxed{Vd}}$$

Very large!

Overfitting Danger!

Losing generalization by re-training word vectors

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

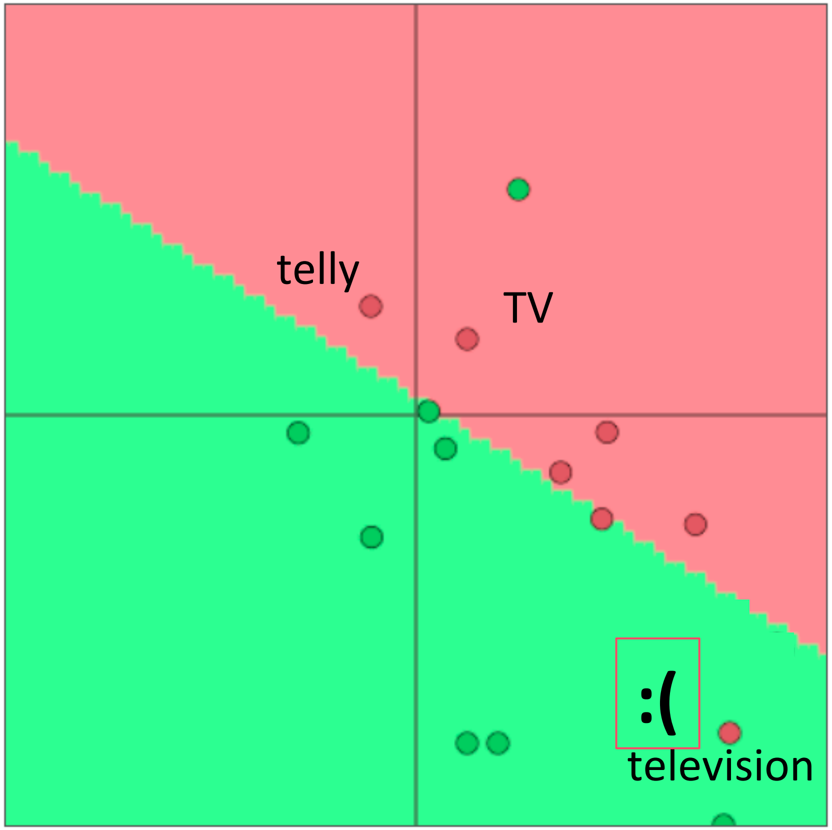


- What happens when we train the word vectors?

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”
- Only in testing data: “television”

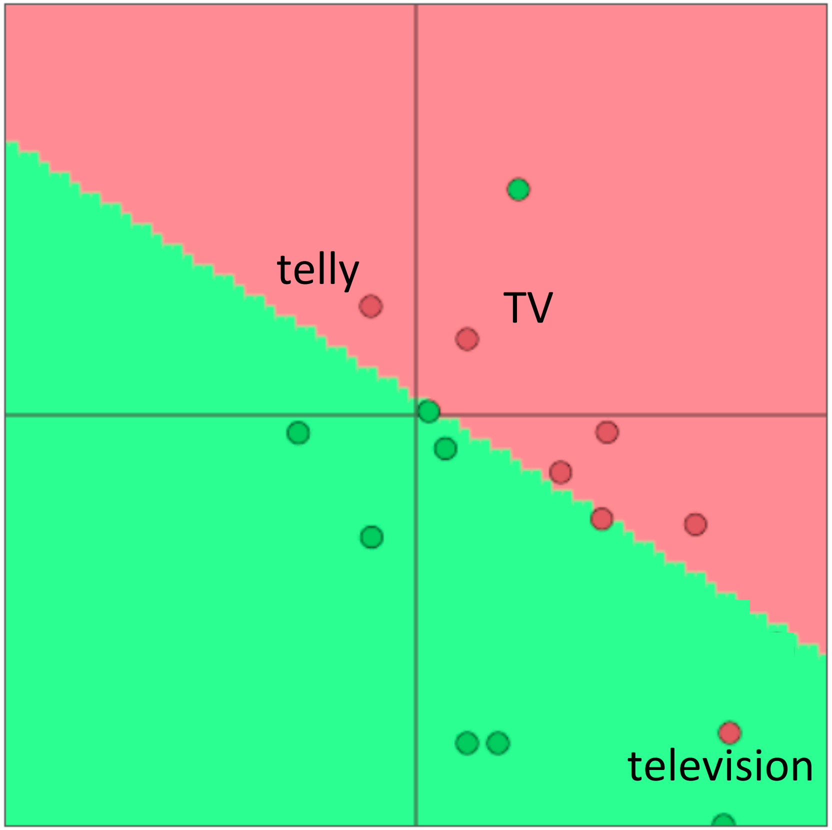


Losing generalization by re-training word vectors

- Take home message:

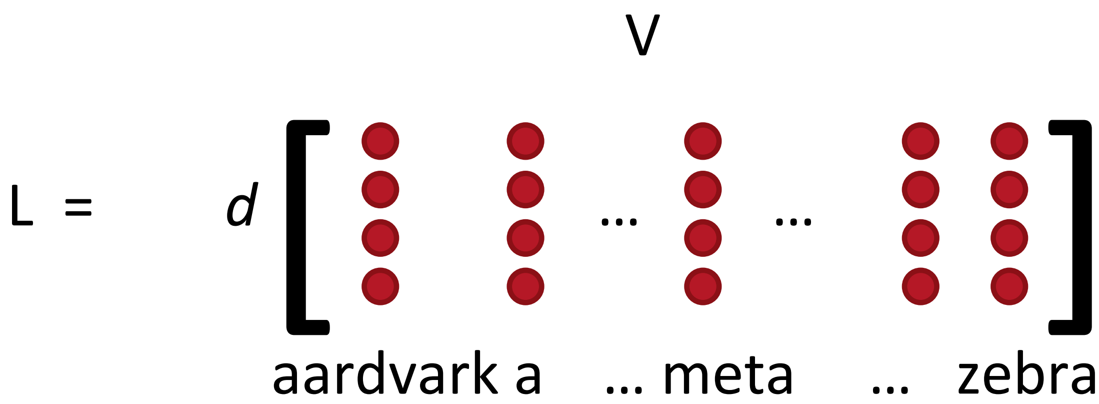
If you only have a small training data set, don't train the word vectors.

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



Side note on word vectors notation

- The word vector matrix L is also called lookup table
- Word vectors = word embeddings = word representations (mostly)
- Mostly 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 into 4 classes:
  - Person, location, organization, none

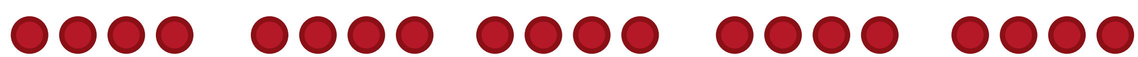
- Many possibilities exist for classifying one word in context, e.g. averaging all the words in a window but that looses position information

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

Simplest window classifier: Softmax

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

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

predicted model output probability

- 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?

Updating concatenated word vectors

- 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 :)

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: Chain rule! If *y* = *f*(*u*) and *u* = *g*(*x*), i.e. *y* = *f*(*g*(x)), then:
- Simple example: $\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx} = \frac{df(u)}{du}\frac{dg(x)}{dx}$

$$\frac{dy}{dx} = \frac{d}{dx}5(x^3+7)^4$$

$$y = f(u) = 5u^4 \qquad\qquad u = g(x) = x^3 + 7$$

$$\frac{dy}{du} = 20u^3 \qquad\qquad \frac{du}{dx} = 3x^2$$

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

Updating concatenated word vectors

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

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

- 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≠ 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:

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

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

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

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

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

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}

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

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

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

- 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!

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 to a large 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)
```

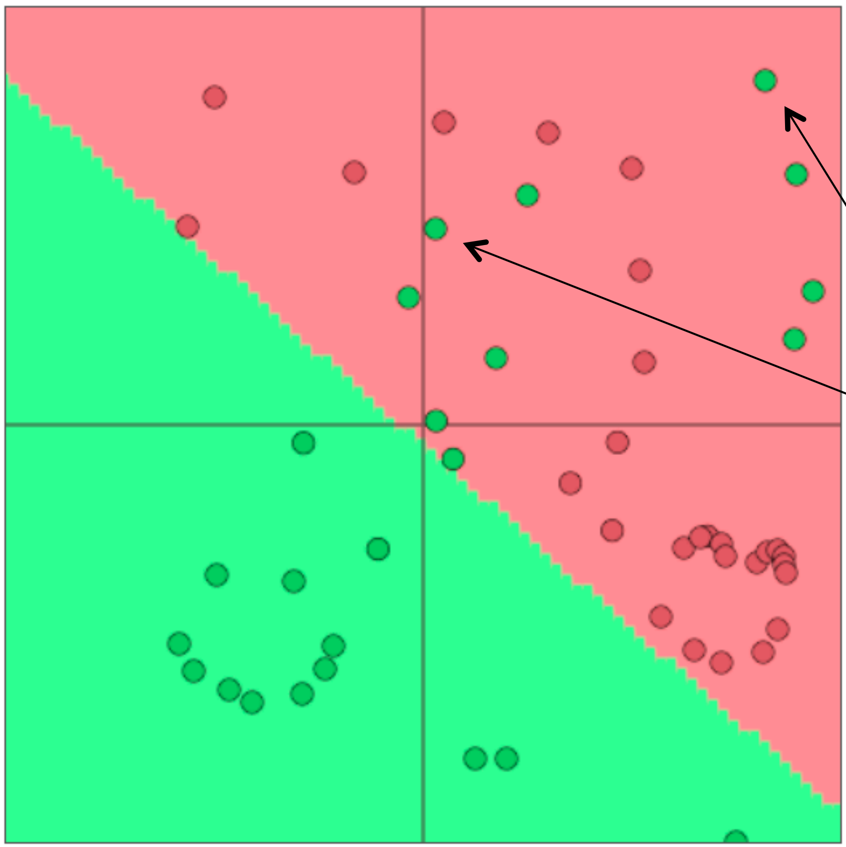
- Result of faster method is a C x N matrix:
  - Each column is an f(x) in our notation (unnormalized class scores)
- Matrices are awesome!
- You should speed test your code a lot too

Softmax (= logistic regression) 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

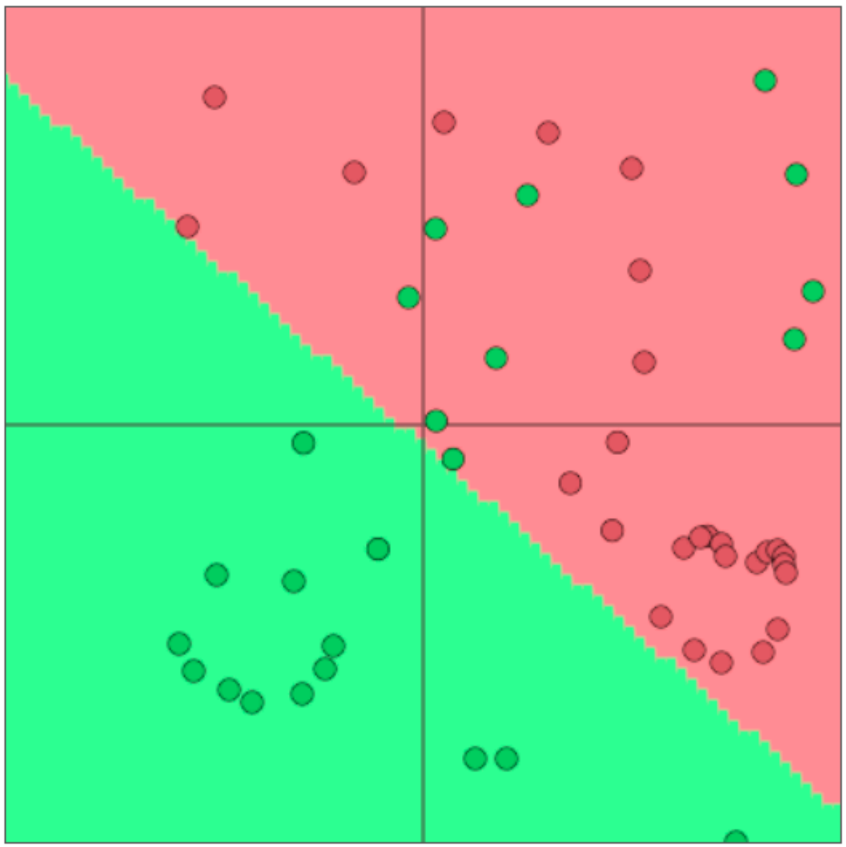


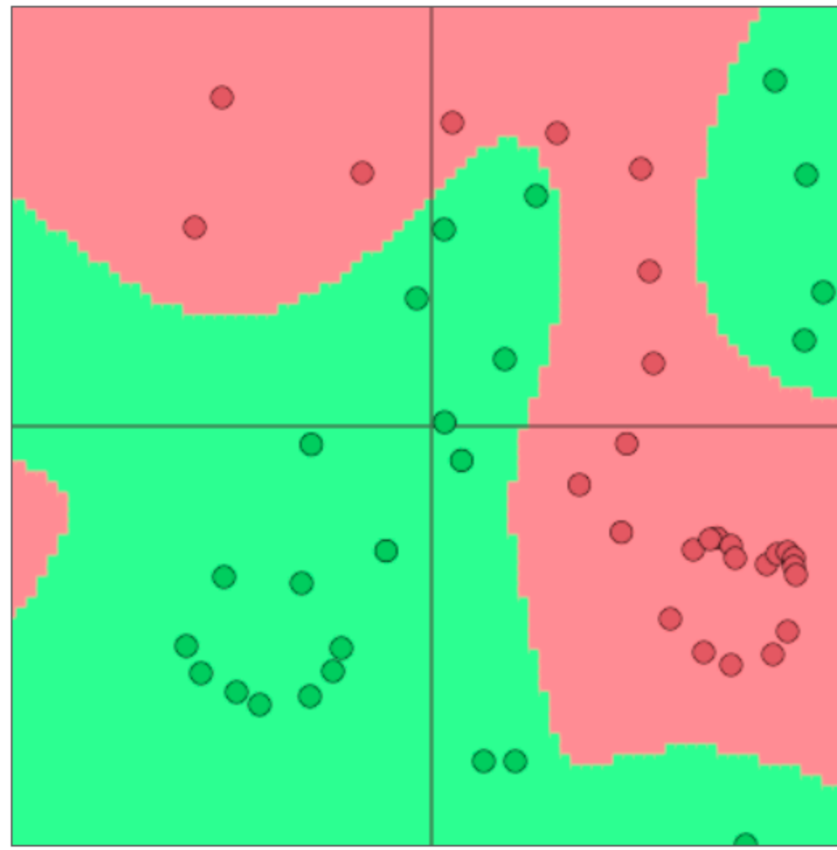
→ Lame when problem is complex

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

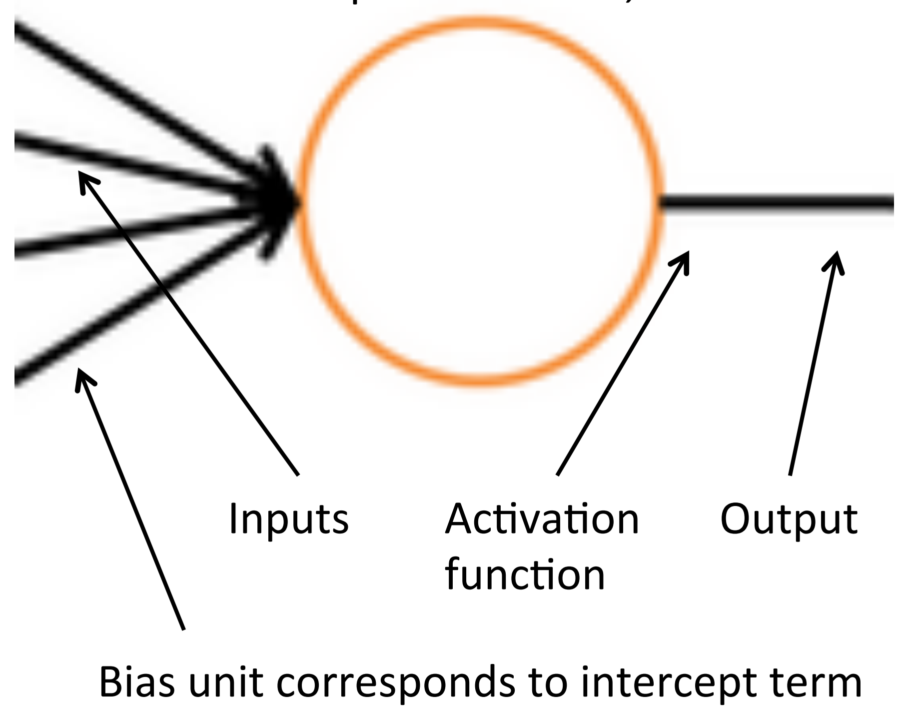
Neural networks come with their own terminological baggage

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*

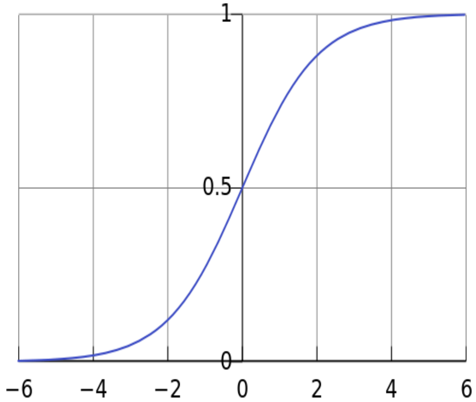


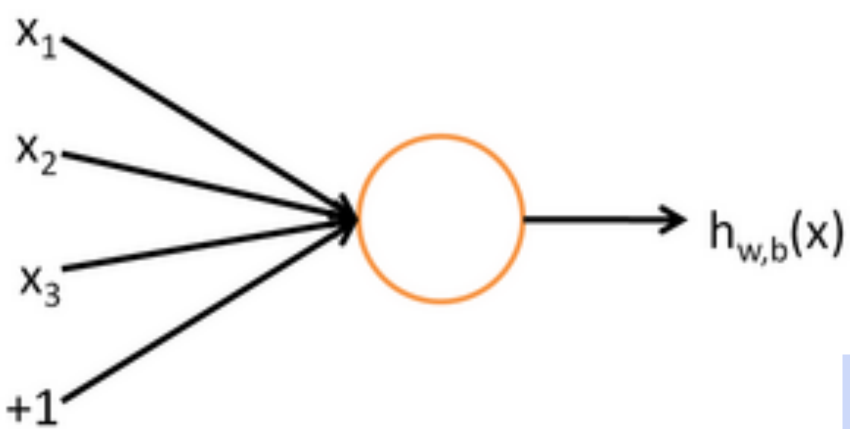
A neuron is essentially a binary logistic regression unit

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

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

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

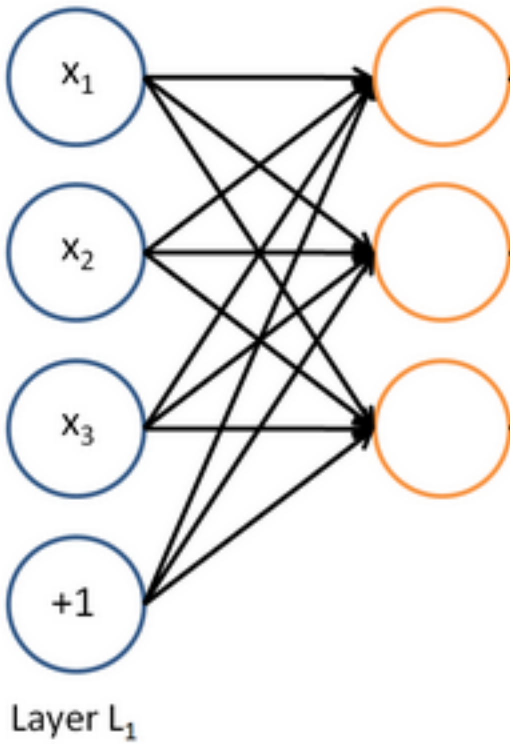




w, *b* are the parameters of this neuron
i.e., this logistic regression model

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

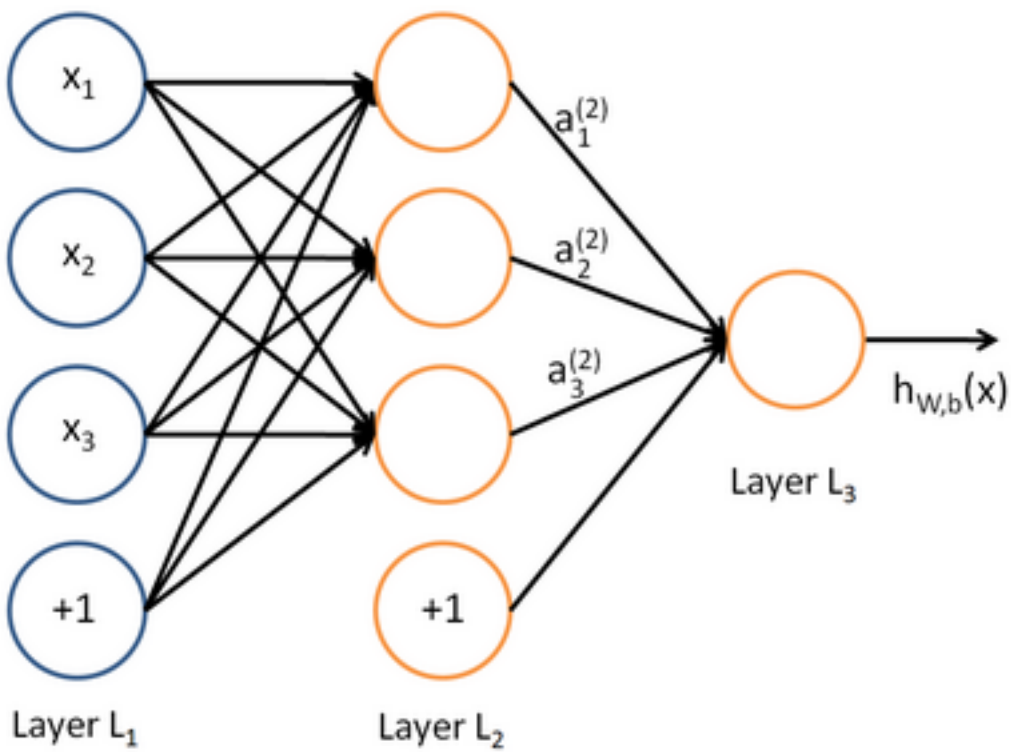
If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs …



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

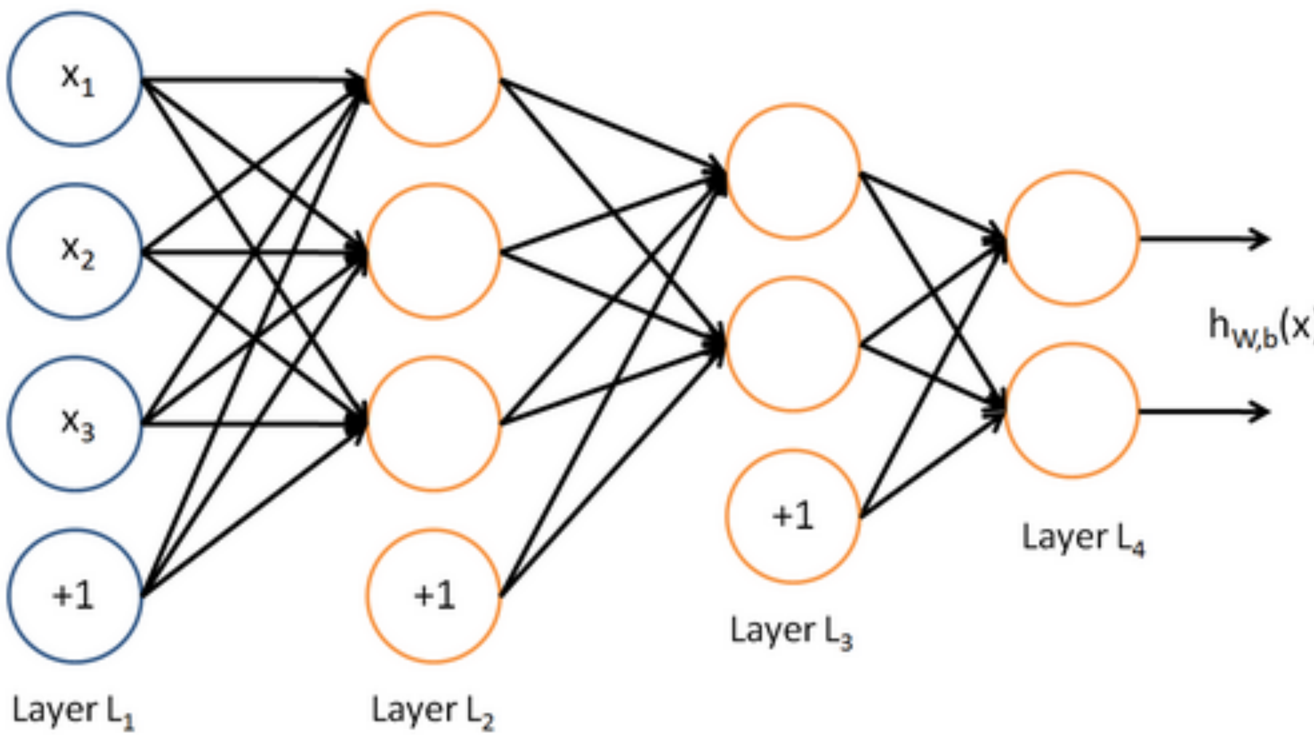
… which we can feed into another logistic regression function



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

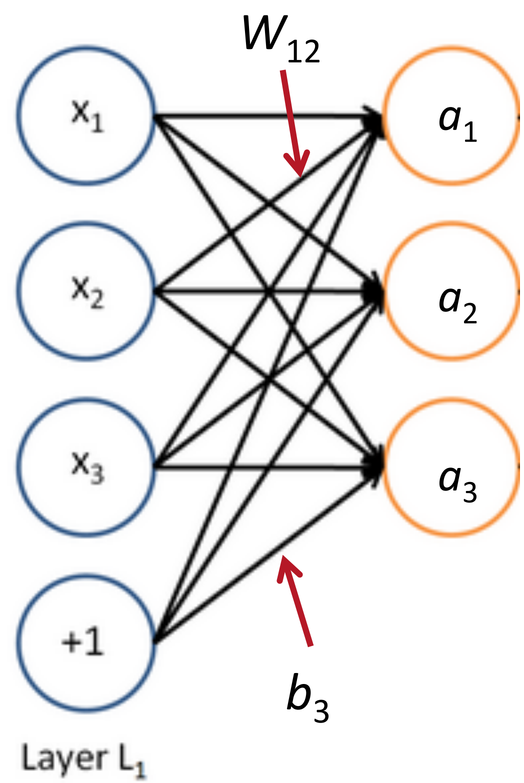
$$a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1)$$
$$a_2 = f(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2)$$
etc.

In matrix notation

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

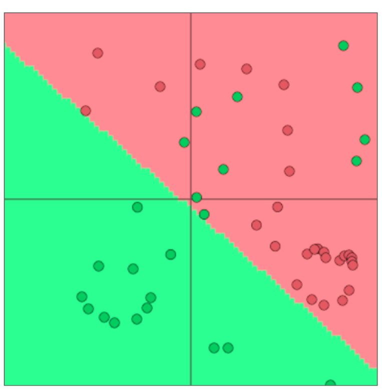
where *f* is applied element-wise:

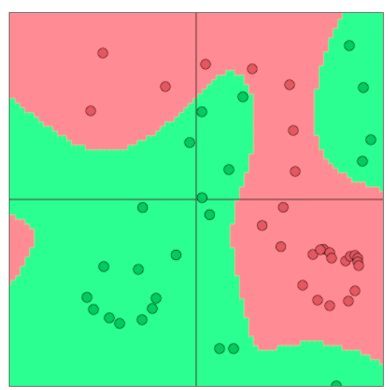
$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$

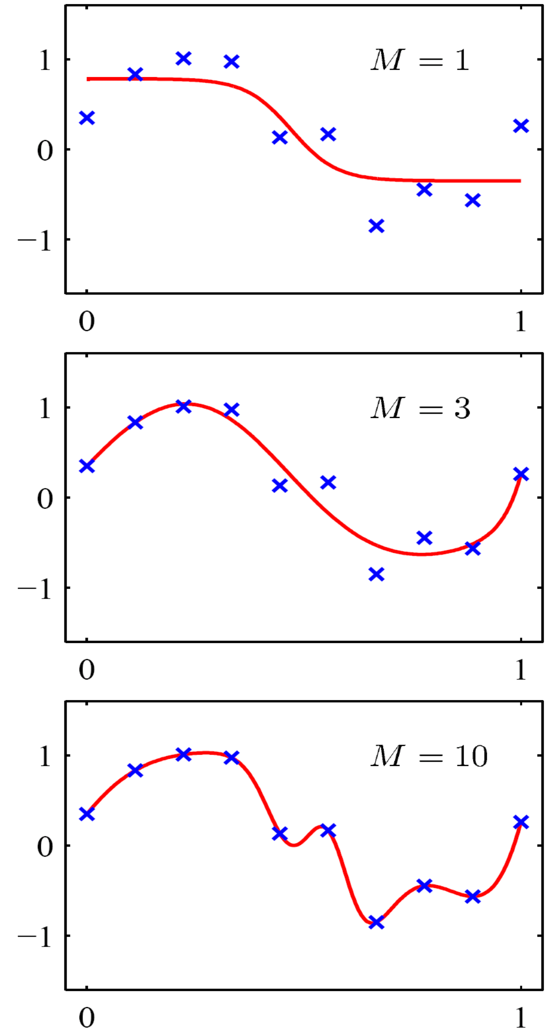


Non-linearities (f): Why they're needed

- 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!







A more powerful, neural net 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:
$$\begin{aligned} z &= Wx + b \\ a &= f(z) \end{aligned}$$
- The neural activations *a* can then be used to compute some output
- For instance, a probability via softmax
$pyx = \mathrm{softmax}(Wa)$
- Or an unnormalized score (even simpler)

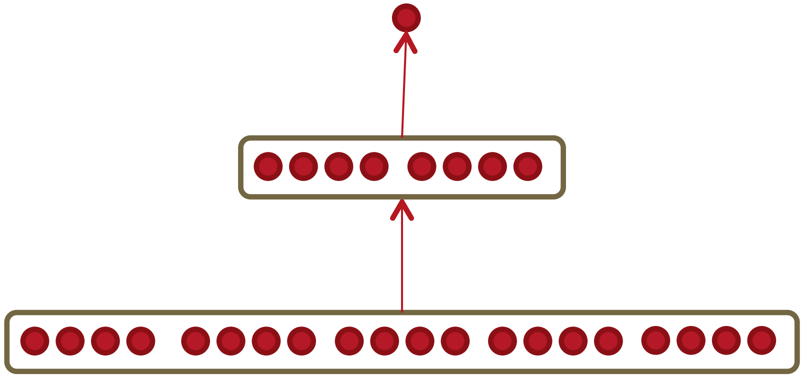
$$score(x) = U^T a \in \mathbb{R}$$

Summary: Feed-forward Computation

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

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

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

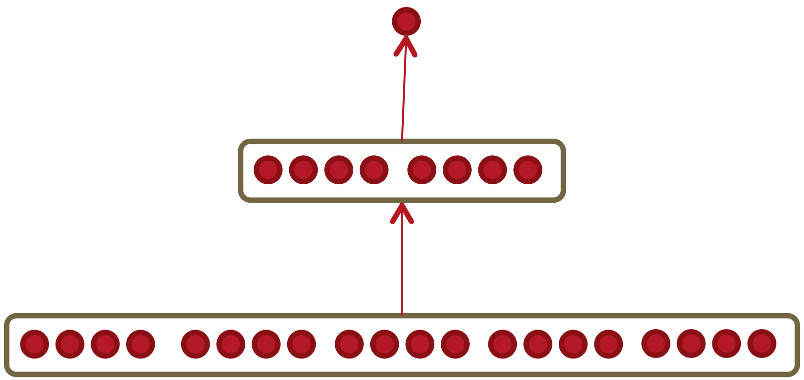


x_{window} = [$x_{museums}$ x_{in} x_{Paris} x_{are} $x_{amazing}$]

Main intuition for extra layer

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

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

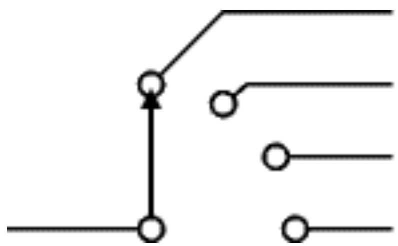


x_{window} = [$x_{museums}$ x_{in} x_{Paris} x_{are} $x_{amazing}$]

The max-margin loss

- 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 corrupt window's score lower (until they're good enough): minimize

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



- This is continuous --> we can use 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: Sample several corrupt windows per true one. Sum over all training windows

Training with Backpropagation

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

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

Assuming cost *J* is > 0,

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

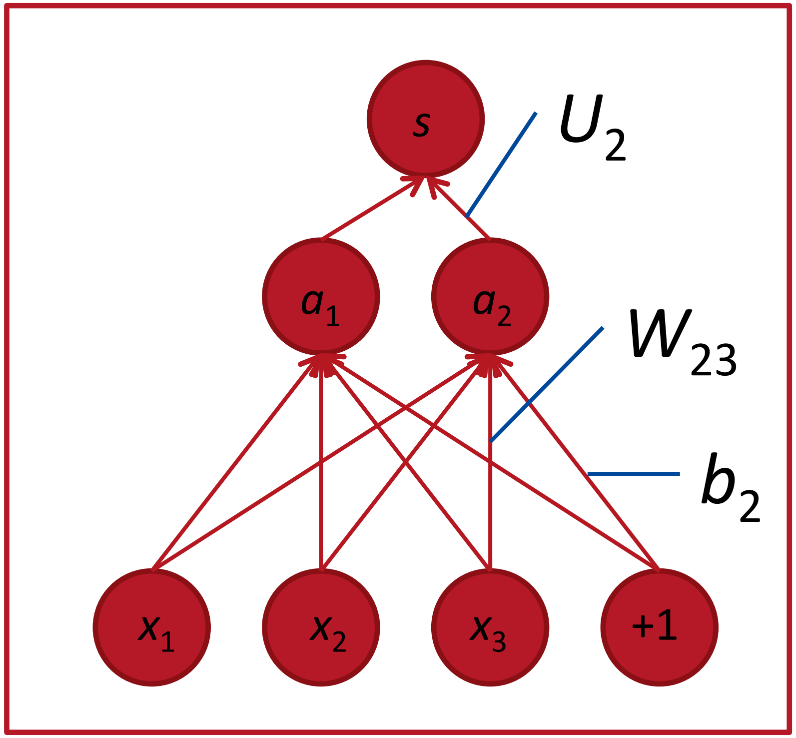
$$\frac{\partial s}{\partial U} = \frac{\partial}{\partial U} U^T a \qquad \frac{\partial s}{\partial U} = a$$

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

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

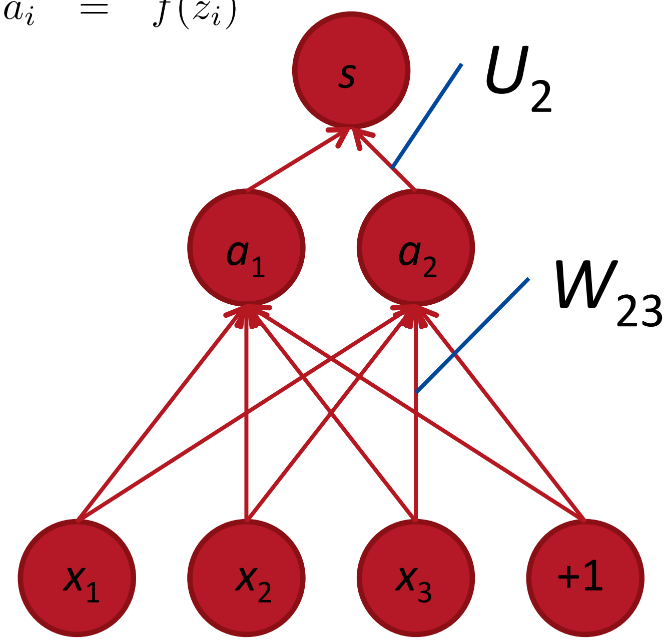
Derivative of weight W_{ij}:

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

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

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

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



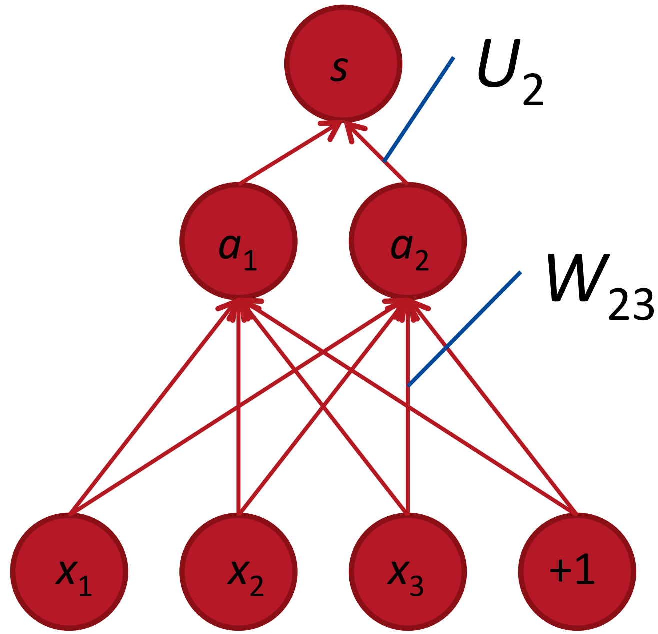
Training with Backpropagation

Derivative of single weight W_{ij} :

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

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

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



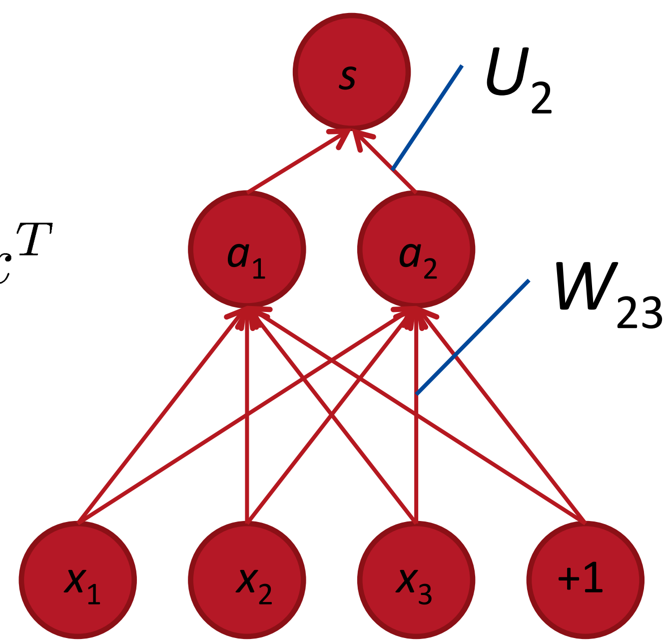
Training with Backpropagation

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

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

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

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

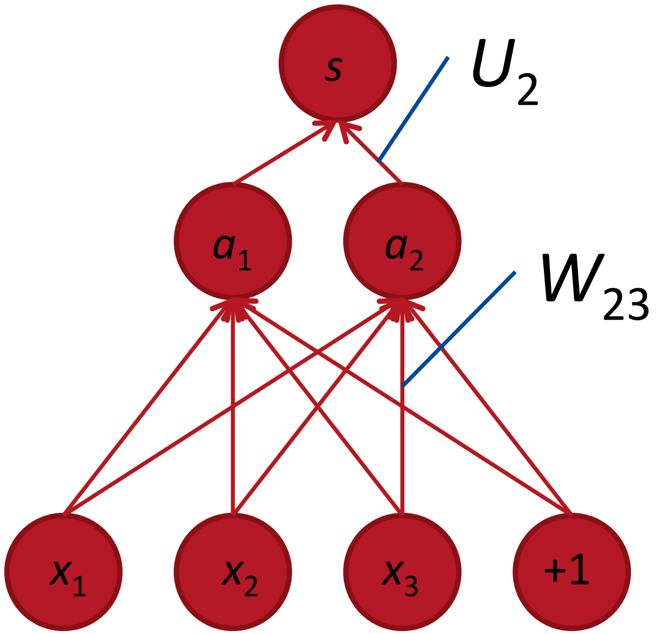


Training with Backpropagation

- For biases b, we get:

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

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



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:

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

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? →

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

- 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 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 → PSet1
- Scores and max-margin loss
- Neural network → PSet1

One more half of a math-heavy lecture

Then the rest will be easier and more applied :)

Next lecture:

Project advice

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

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