CS224d
Deep NLP

Lecture 4:
Word Window Classification and Neural Networks

Richard Socher

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**)

Refresher: Classification setup and notation

- Generally we have a training dataset consisting of samples

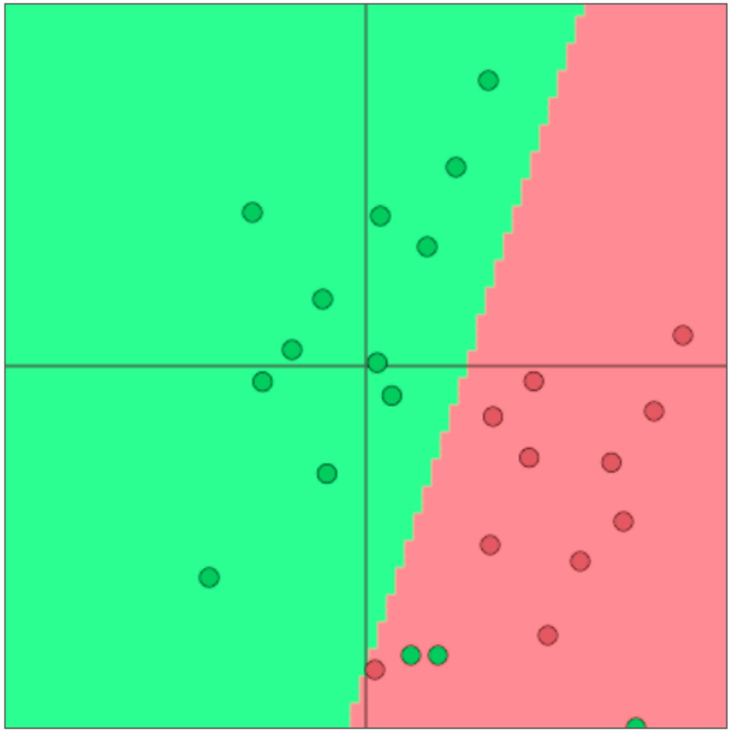
$$\{x_i, y_i\}^N_{i=1}$$

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

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

Classification intuition

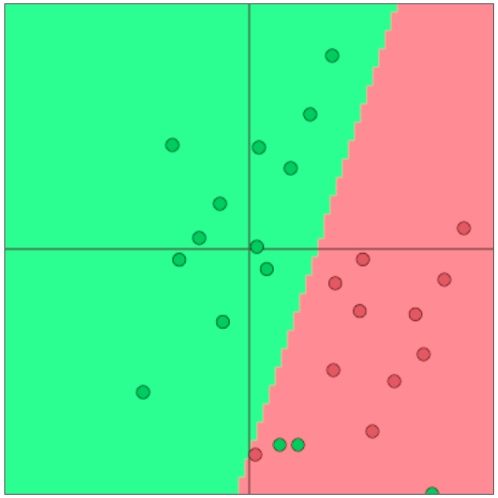
- Training data: $\{x_i, y_i\}^N_{i=1}$
- Simple illustration case:
  - Fixed 2d word vectors to classify
  - Using logistic regression
  - → linear decision boundary →
- General ML: assume x is fixed and only train logistic regression weights W and only modify the decision boundary



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

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

Classification notation



- General ML: only train logistic regression weights and hence only modify the decision boundary

- Loss function over dataset $\{x_i,y_i\}^N_{i=1}$

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

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

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

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

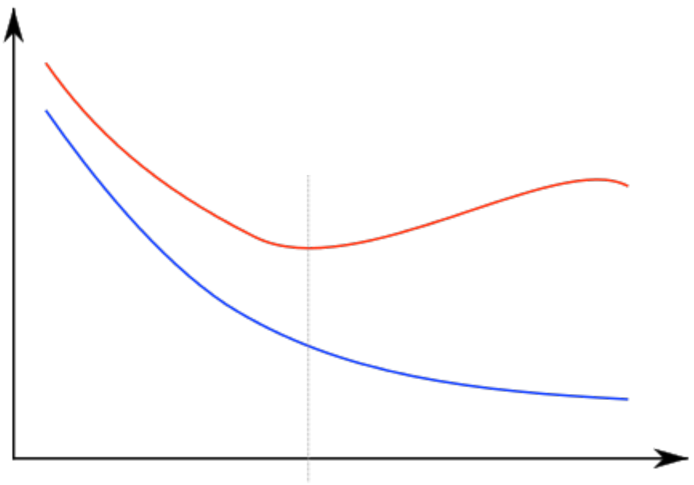
$$f = Wx$$

Classification: Regularization!

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

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

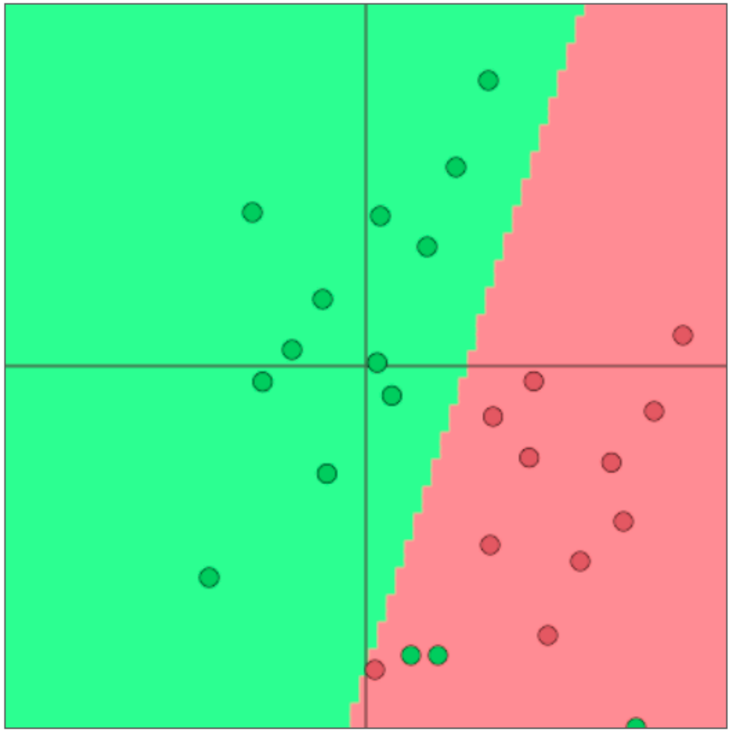
- Regularization will prevent overfitting when we have a lot of features (or later a very powerful/deep model)
  - x-axis: more powerful model or more training iterations
  - Blue: training error, red: test error



Classification difference with word vectors

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

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



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

- For general ML θ usually only consists of columns of W
- Additionally common in deep learning:
  - Learn both W and word vectors x

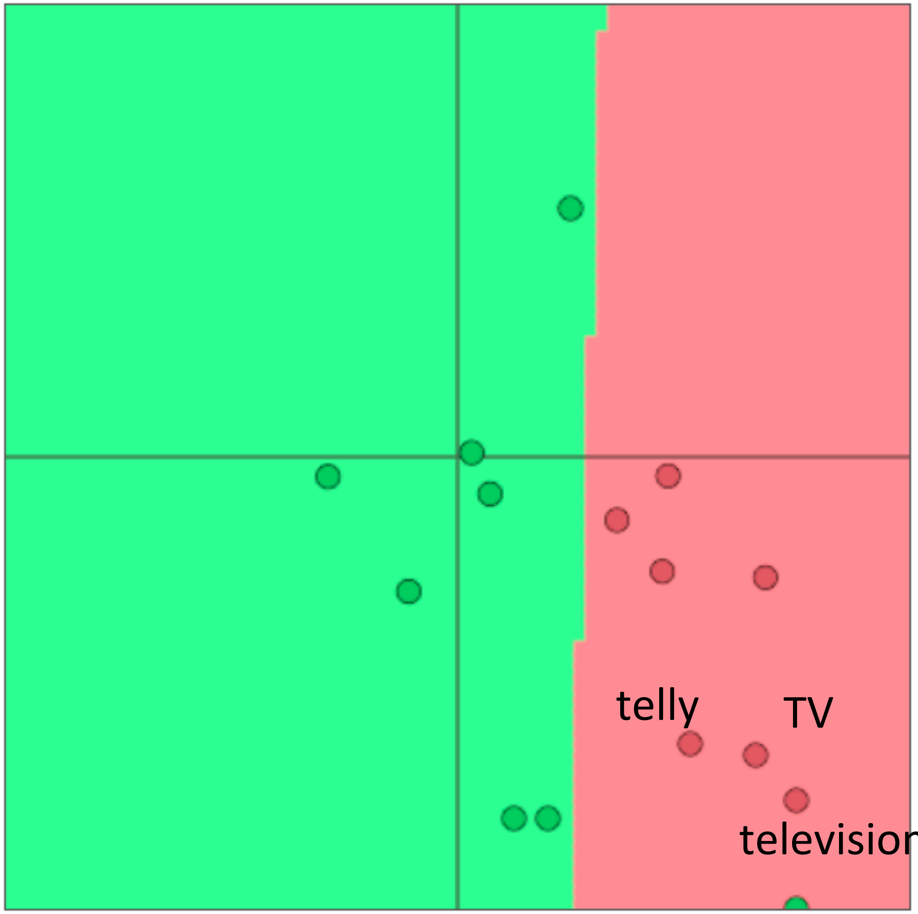
$$\nabla_\theta J(\theta) = \begin{bmatrix} \nabla_{W_{\cdot 1}} \\ \vdots \\ \nabla_{W_{\cdot d}} \\ \nabla_{x_{aardvark}} \\ \vdots \\ \nabla_{x_{zebra}} \end{bmatrix} \in \mathbb{R}^{Cd + \boxed{Vd}}$$

Very large!

Overfitting Danger!

Loosing generalization by re-training word vectors

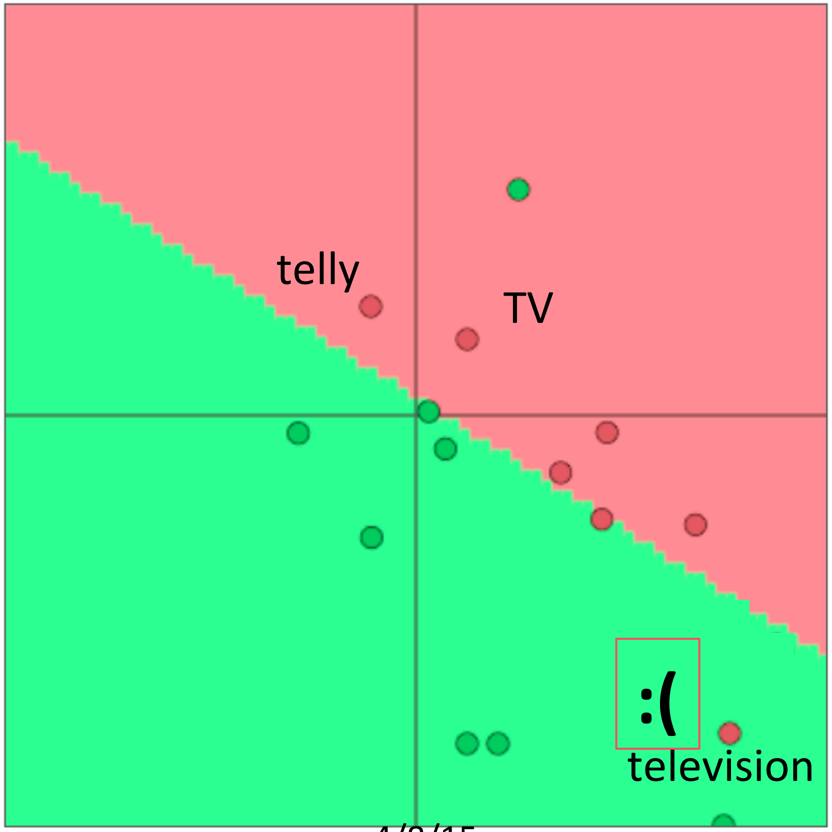
- Setting: Training logistic regression for movie review sentiment and in the training data we have the word
  - “TV” and “telly”
- In the testing data we have
  - “television”
- Originally they were all similar (from pre-training word vectors)
- What happens when we train the word vectors?



Loosing 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"

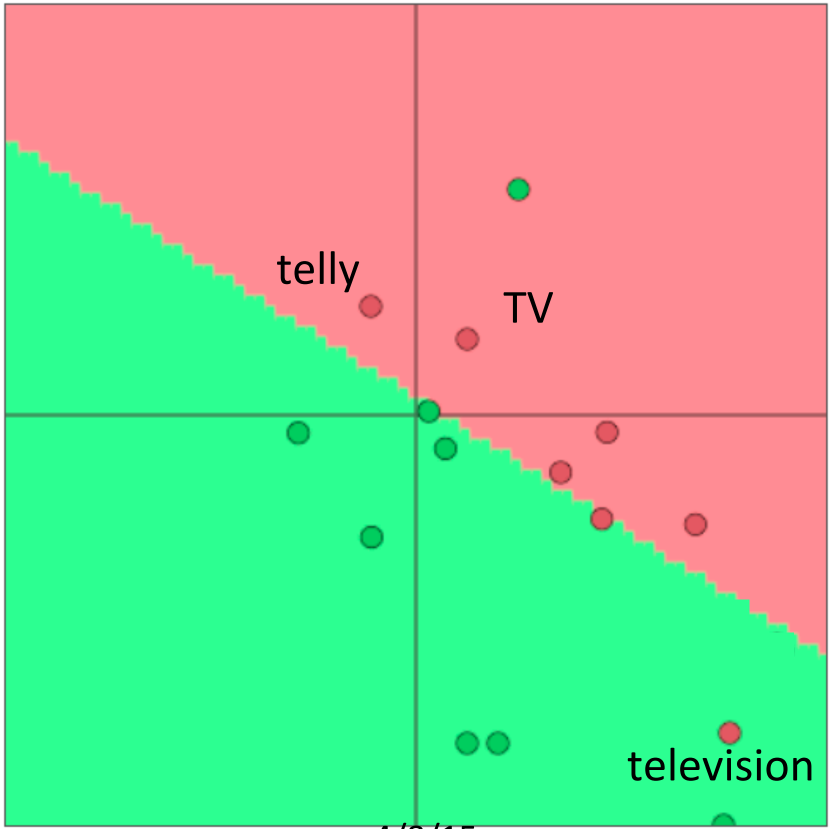


Loosing generalization by re-training word vectors

- Take home message:

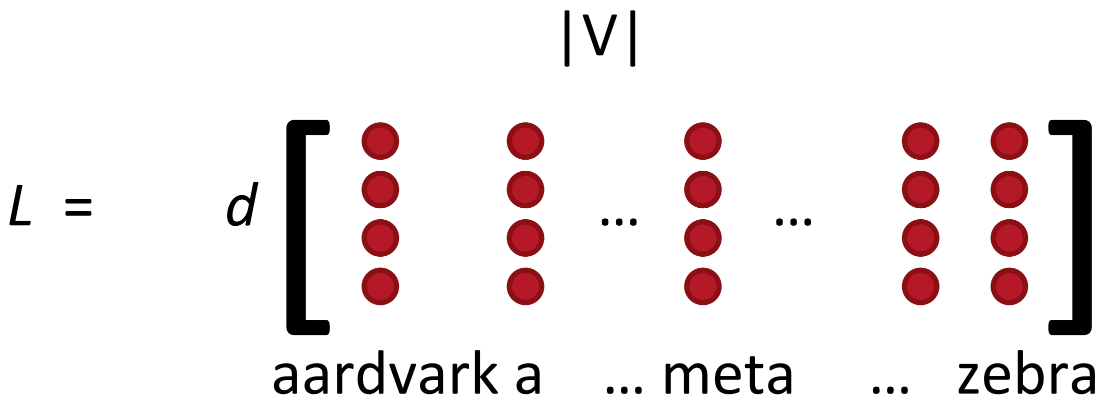
If you only have a small training data set, don't train the word vectors.

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

- Conceptually you get a word's vector by left multiplying a one-hot vector *e* by *L*: $x = Le \in d \times V \cdot V \times 1$

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 Hilton vs Paris, France
  - Hathaway → Berkshire Hathaway, Anne Hathaway

Window classification

- Idea: Instead of classifying a single word, just classify a word together with its context window of neighboring words.

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

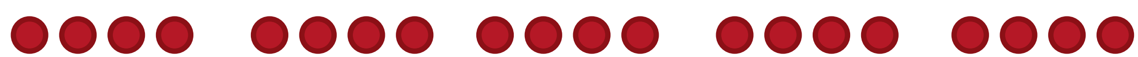
- Many possibilities exist for classifying one word in context, e.g. averaging all the words in a window but that looses position information

Window classification

- Most commonly used technique to classify a word in a window
- Train 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 … .



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

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

Simplest window classifer: Softmax

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

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

= predicted model output probability

same

- With cross entropy error as before:

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

- 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 the steps together (you'll have to fill in the details in 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)
  - $f = Wx \in \mathbb{R}^C$ 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 in which variables other variables are being used:

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

- 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, think of your results in terms of vector operations and define new, single index-able vectors:

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

Updating concatenated word vectors

- Tip 5: When you start with the chain rule, first use explicit sums and look at partial derivatives of e.g. x_i or W_{ij}

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

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

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

- Tip 6: To clean it up for even more complex functions later: Know dimensionality of variables &simplify into matrix notation

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

- Tip 7: Write this out in full sums if it's not clear!

Updating concatenated word vectors

- Tip 5: When you start with the chain rule, first use explicit sums and look at partial derivatives of e.g. x_i or W_{ij}

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

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

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

- Tip 6: To clean it up for even more complex functions later: Know dimensionality of variables &simplify into matrix notation

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

- Tip 7: Write this out in full sums if it's not clear!

Updating concatenated word vectors

- What is the dimensionality of

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

- X is the entire window of 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 $\nabla_x J = W^T \delta = \delta_{window}$
- With $x_{window} = [\ x_{museums} \quad x_{in} \quad x_{Paris} \quad x_{are} \quad x_{amazing}\ ]$
- We have

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

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 that does the same mathematical operation!

- 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)
```

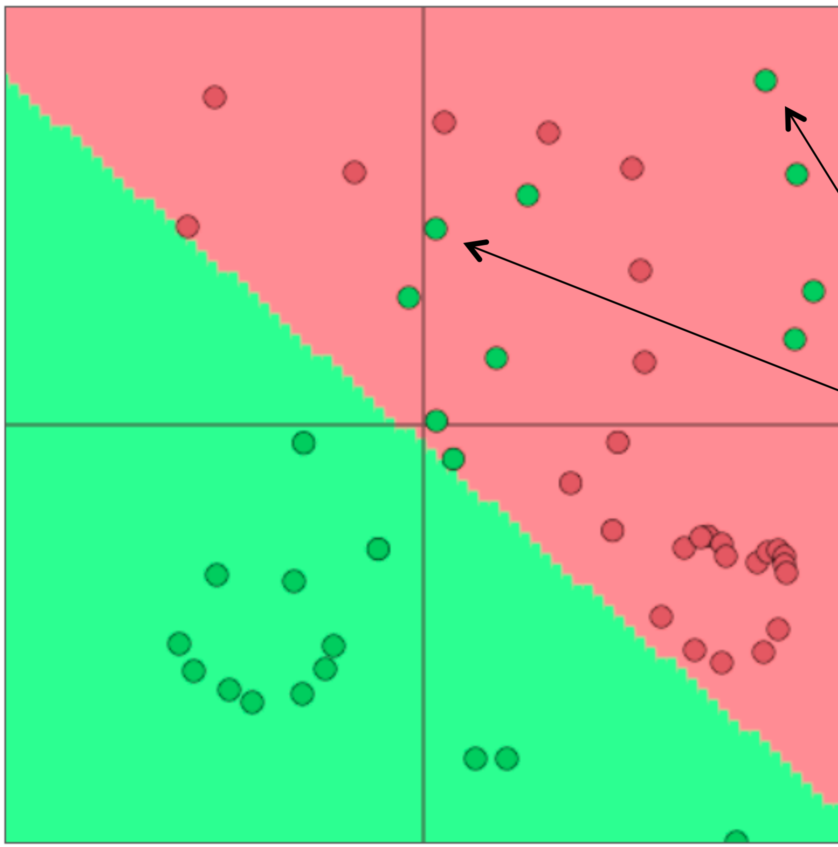
- Result of faster method is a C x N matrix:
  - Each column is an f(x) in our notation (unnormalized class scores)
- Matrices are awesome!
- You should speed test your code a lot too

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

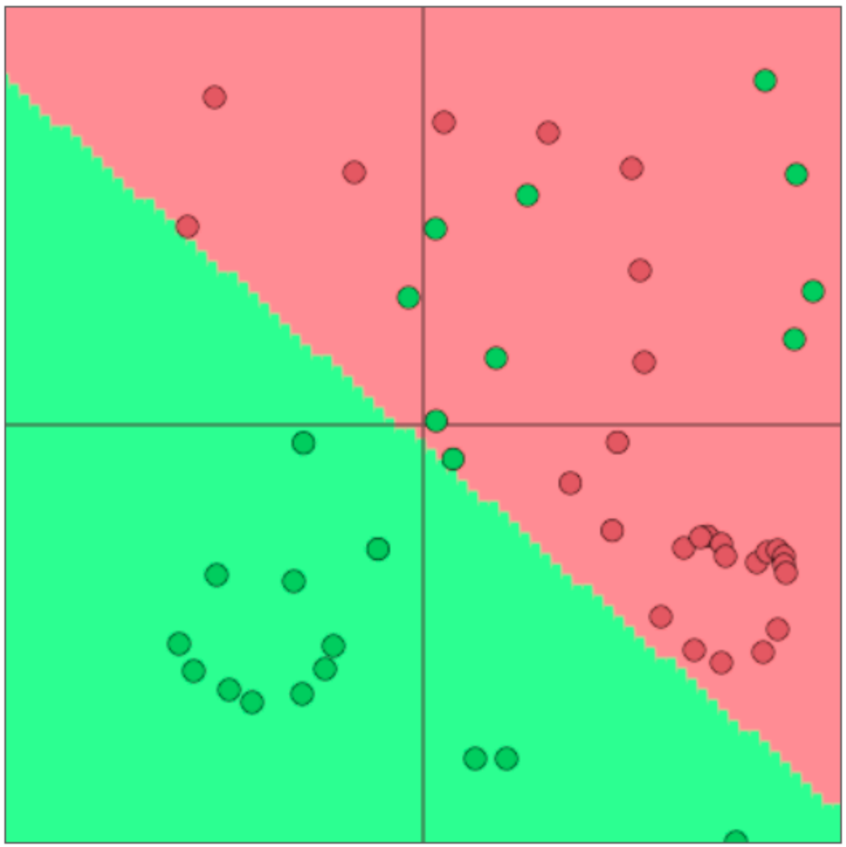


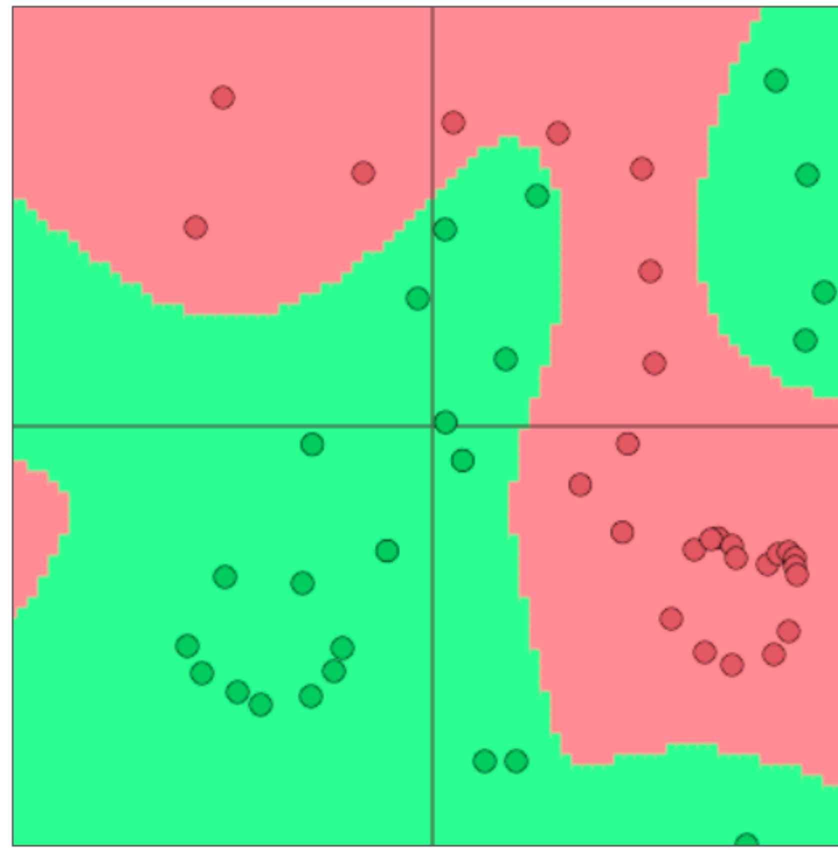
→ Lame when problem is complex

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

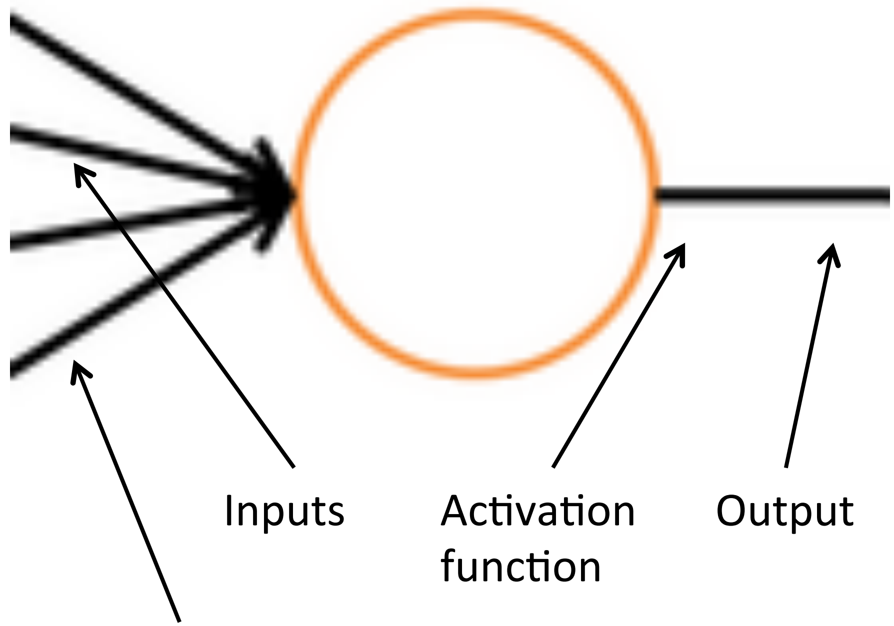
… just like SVMs

But if you understand how softmax models work

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

A single neuron

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



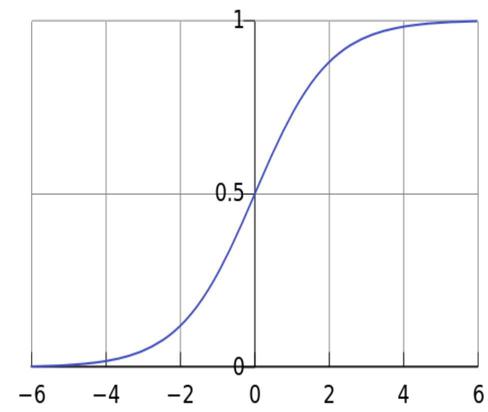
Bias unit corresponds to intercept term

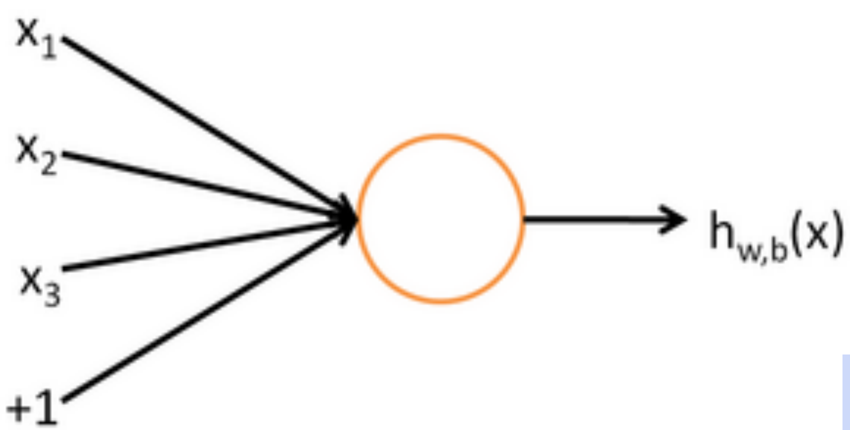
A neuron is essentially a binary logistic regression unit

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

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

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

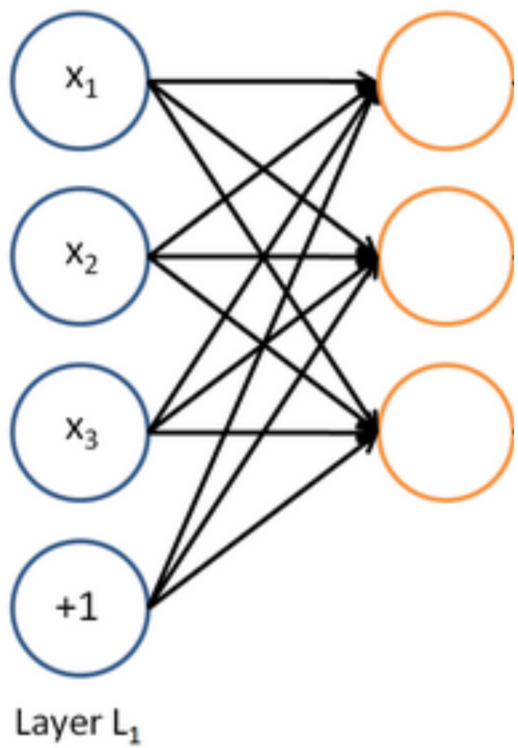




w, *b* are the parameters of this neuron i.e., this logistic regression model

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

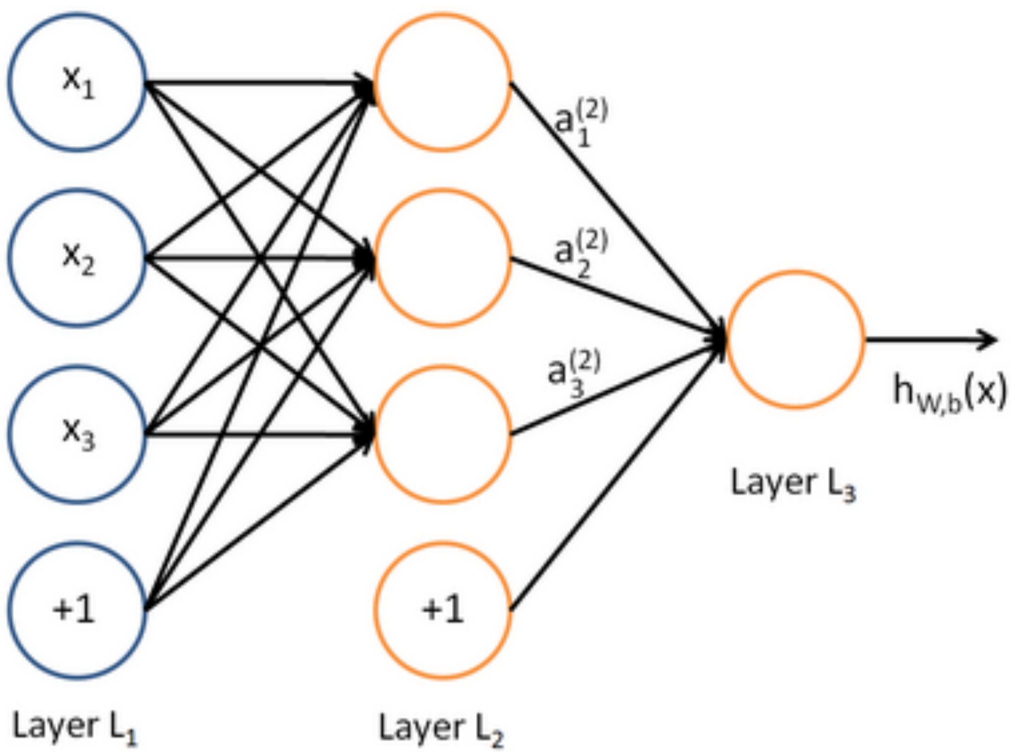
If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs …



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

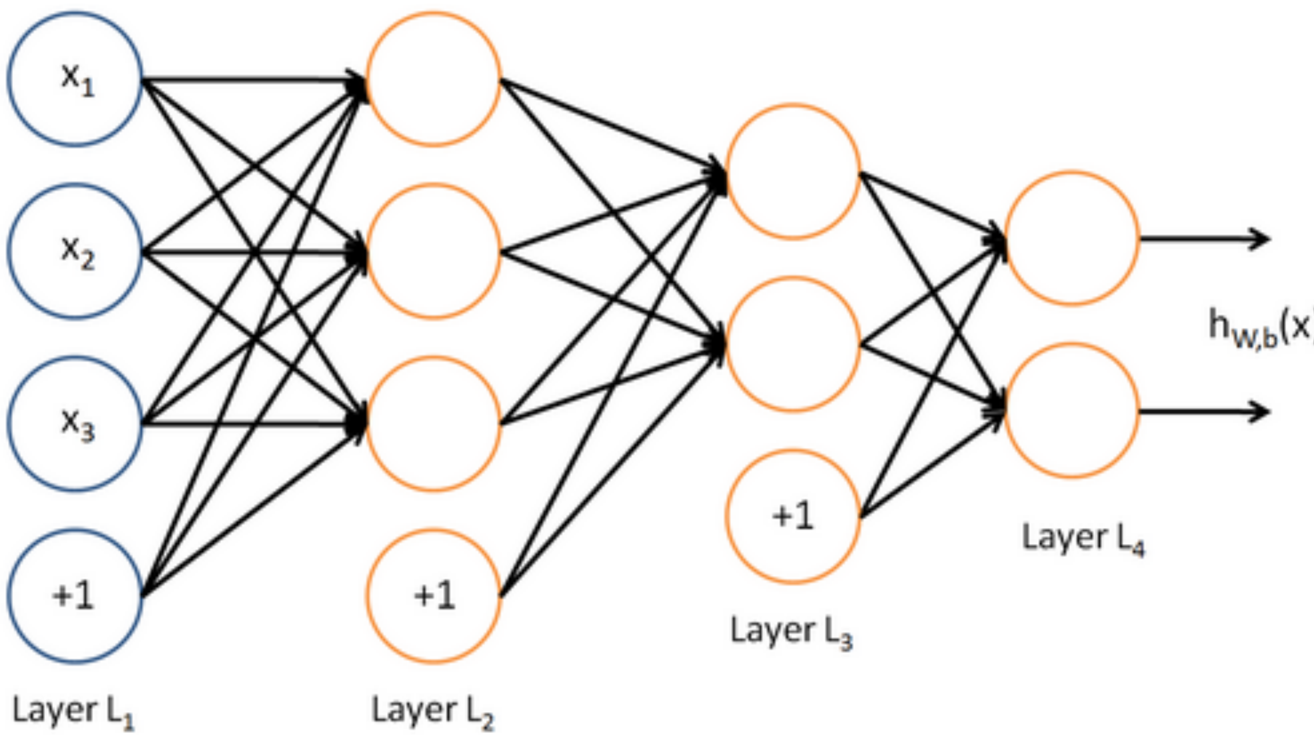
… which we can feed into another logistic regression function



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.

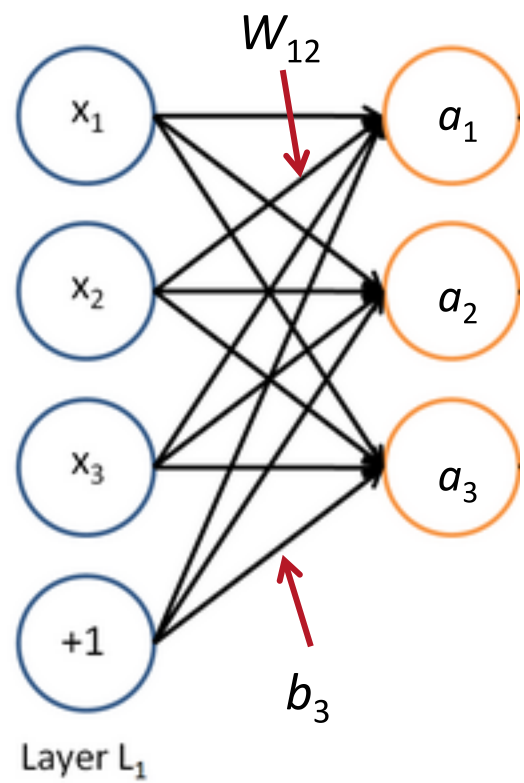
In matrix notation

$$z = Wx + b$$

$$a = f(z)$$

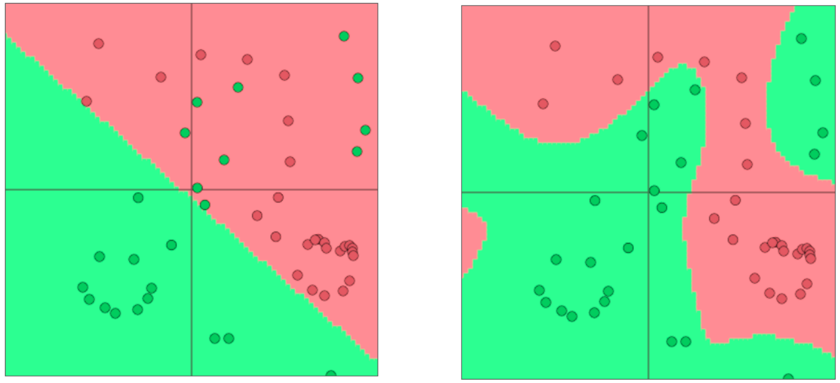
where *f* is applied element-wise:

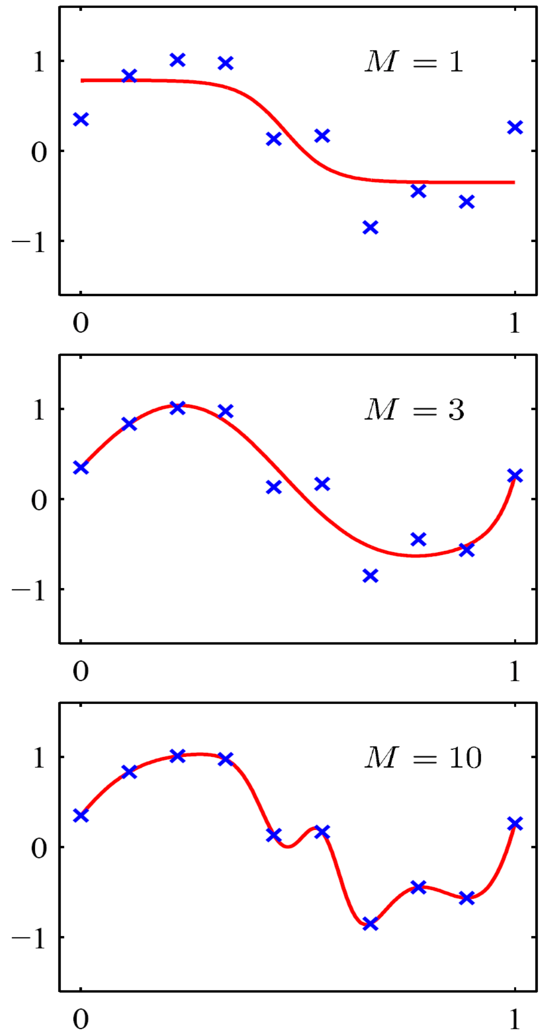
$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$



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 window classifier

- Revisiting

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

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 function
- For instance, a softmax probability or an unnormalized score or a we care about:

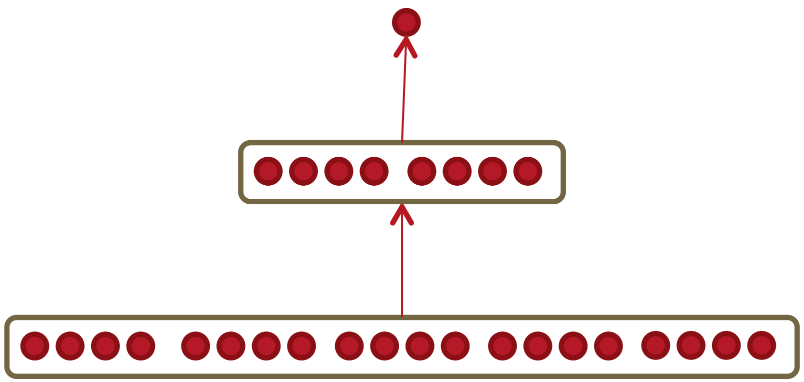
$$score(x) = U^T a \in \mathbb{R}$$

Summary: Feed-forward Computation

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

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

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



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

Next lecture:

Training a window-based neural network.

Taking more **deeper derivatives** → **Backprop**

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