Deep Learning Lessons

Word Classification, NER, Key Concepts in Deep Learning

Marta R. Costa-jussà

with slides from Christopher Manning, Stanford University

Outline

Classification Task with Neural Networks

Named Entity Recognition as an example of Classification Task

Deep Learning Computation Key Concepts

What to read

The Deep Learning book, Ian Goodfellow and Yoshua Bengio and Aaron Courville, MIT Press

CS224n: Natural Language Processing with Deep Learning
Stanford / Winter 2020

Next: Classification Task with Neural Networks

Classification setup and notation

- Generally we have a training dataset consisting of samples

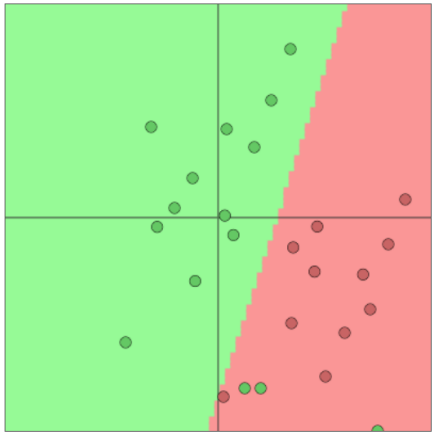
$$\{x_i, y_i\}_{i=1}^{N}$$

- x_i are inputs, e.g. words (indices or vectors!), 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: $\{x_i, y_i\}^N_{i=1}$
- Simple illustration case:
  - Fixed 2D word vectors to classify
  - Using softmax/logistic regression
  - Linear decision boundary



Visualizations with ConvNetJS by Karpathy!

http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

- **Traditional ML/Stats approach:** assume x_i are fixed, train (i.e., set) softmax/logistic regression weights $W \in \mathbb{R}^{C \times d}$ to determine a decision boundary (hyperplane) as in the picture
- **Method**: 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 $c = 1, \ldots, C$

2. Apply softmax function to get normalized probability:

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

Training with softmax and cross-entropy loss

- For each training example (x,y), our objective is to maximize the probability of the correct class *y*

- This is equivalent to minimizing the negative log probability of that class:

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

  - Using log probability converts our objective function to sums, which is easier to work with on paper and in implementation.

Background: What is “cross entropy” loss/error?

- Concept of “cross entropy” is from information theory
- Let the true probability distribution be *p*
- Let our computed model probability be *q*
- The cross entropy is:

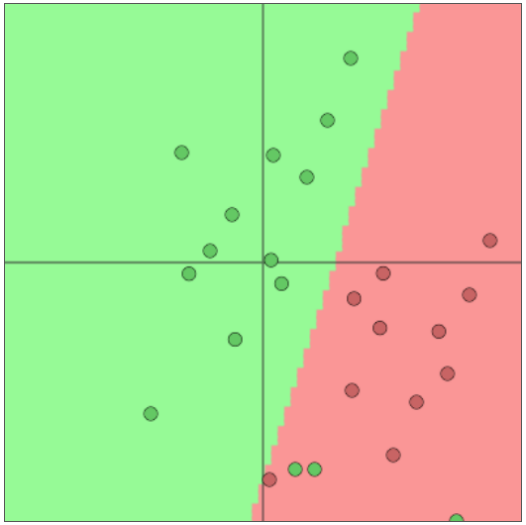
$$H(p,q) = -\sum_{c=1}^{C} p(c) \log q(c)$$

- Assuming a ground truth (or true or gold or target) probability distribution that is 1 at the right class and 0 everywhere else:
  p = [0,…,0,1,0,…0] then:
- **Because of one-hot *p*, the only term left is the negative log probability of the true class**

Classification over a full dataset

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

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



- 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:

$$f = Wx$$

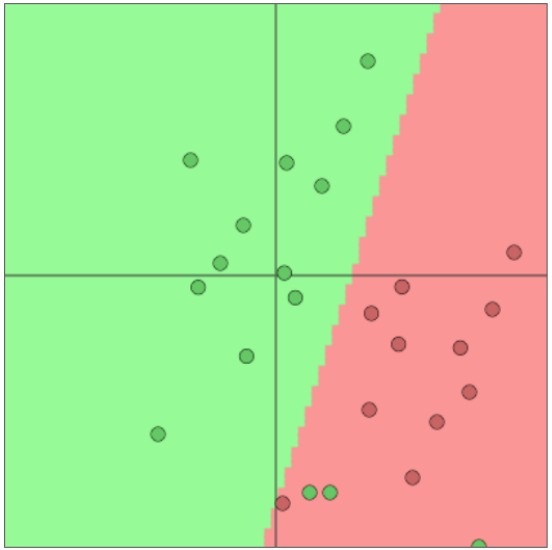
Traditional ML optimization

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

$$\theta = \begin{bmatrix} W_{1\cdot} \\ \vdots \\ W_{C\cdot} \end{bmatrix} = W \in \mathbb{R}^{Cd}$$

- So we only update the decision boundary via

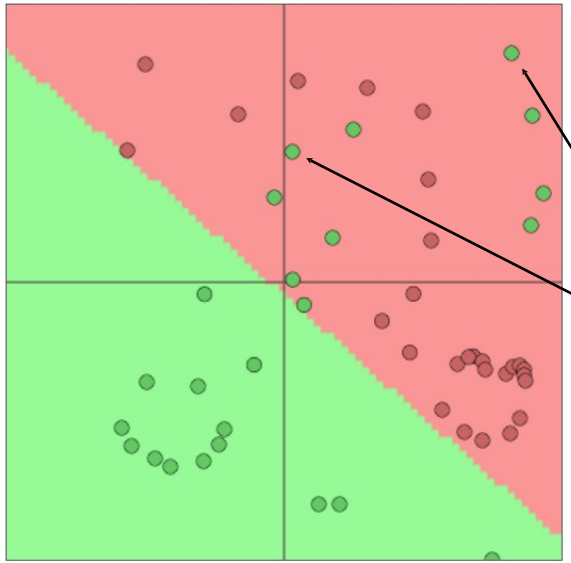
$$\nabla J(\theta) = \begin{bmatrix} \nabla W_{1\cdot} \\ \vdots \\ \nabla W_{C\cdot} \end{bmatrix} \in \mathbb{R}^{Cd}$$



Visualizations with ConvNetJS by Karpathy

Neural Network Classifiers

- Softmax (≈ logistic regression) alone not very powerful
- Softmax gives only linear decision boundaries



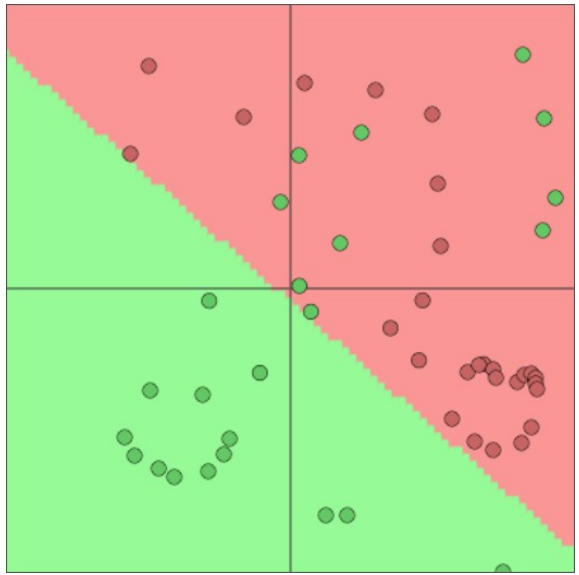
This can be quite limiting

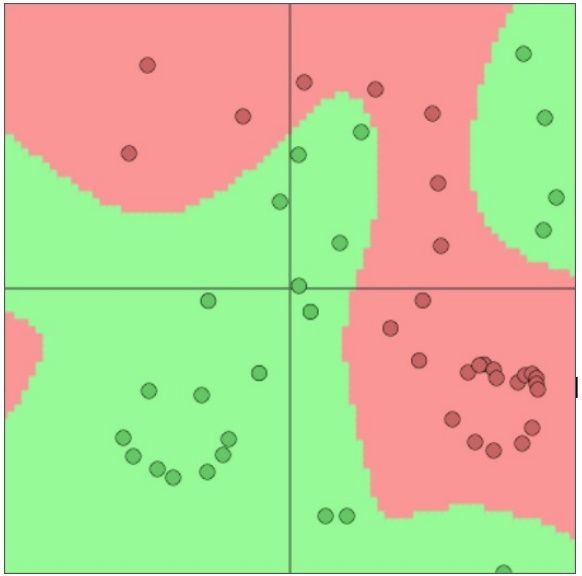
Unhelpful when a problem is complex

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!





13

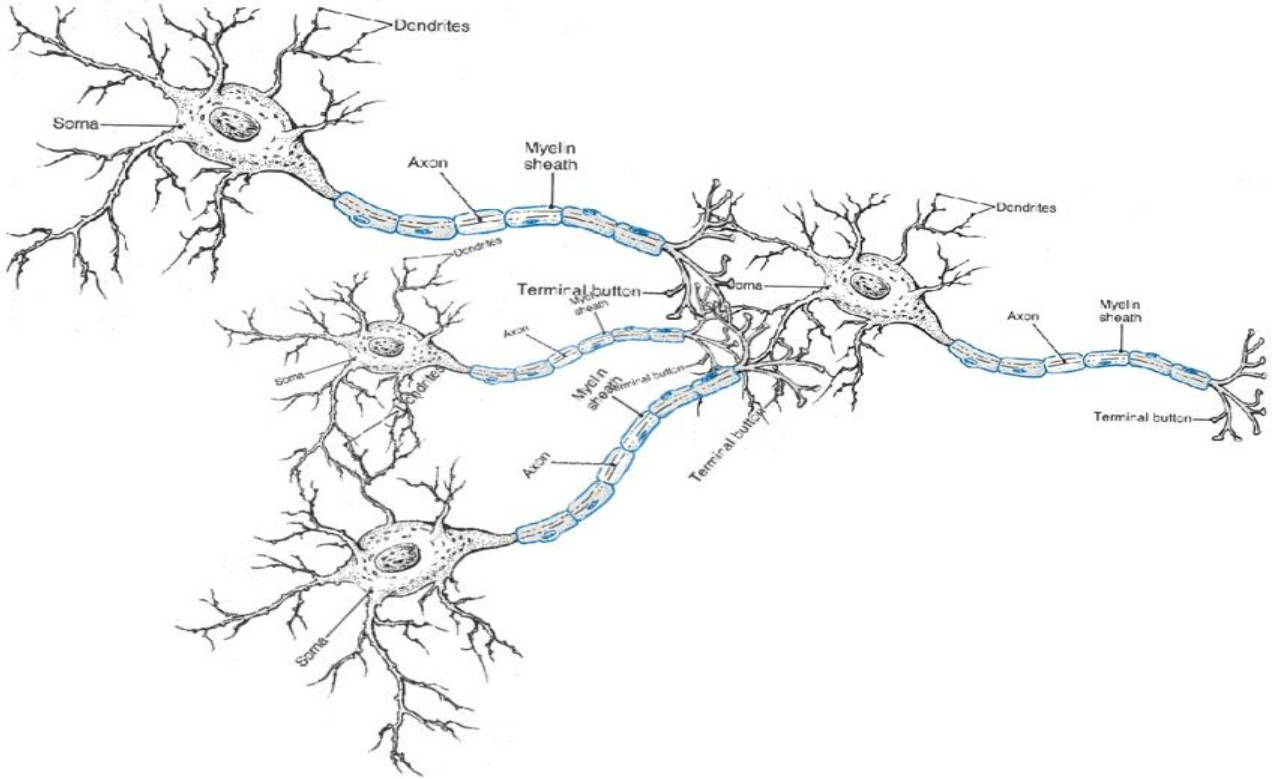
Classification difference with word vectors

- Commonly in NLP deep learning:
  - We learn **both** *W* **and** word vectors ***x***
  - We learn **both** conventional parameters **and** representations
  - The word vectors re-represent one-hot vectors—move them around in an intermediate layer vector space—for easy classification with a (linear) softmax classifier

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

Very large number of parameters!

Neural computation



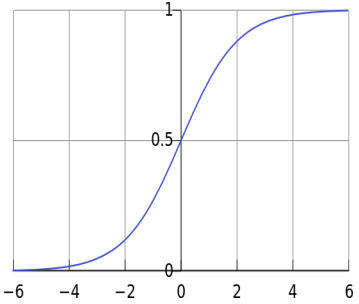
A neuron can be a binary logistic regression unit

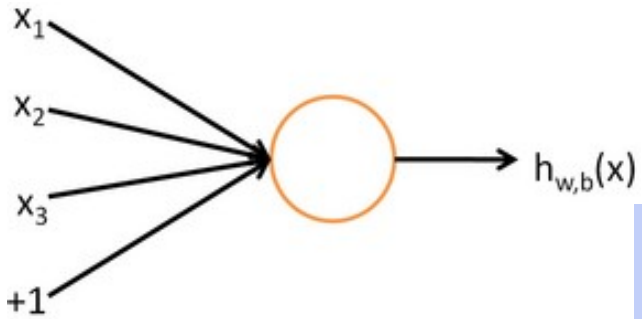
f = nonlinear activation fct. (e.g. sigmoid), *w* = weights, *b* = bias, *h* = hidden, *x* = inputs

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

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

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

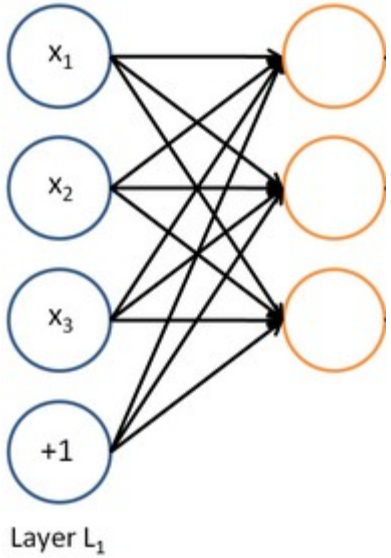




w, *b* are the parameters of this neuron
i.e., this logistic regression model

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

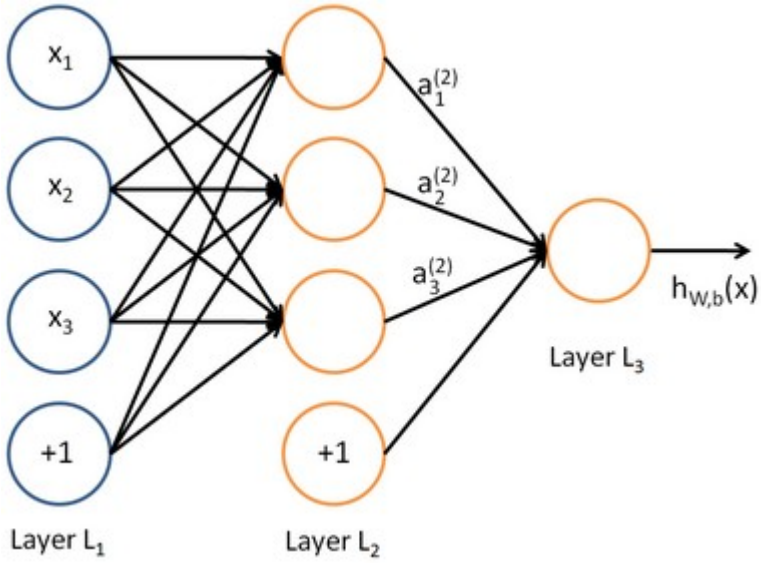
If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs …



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

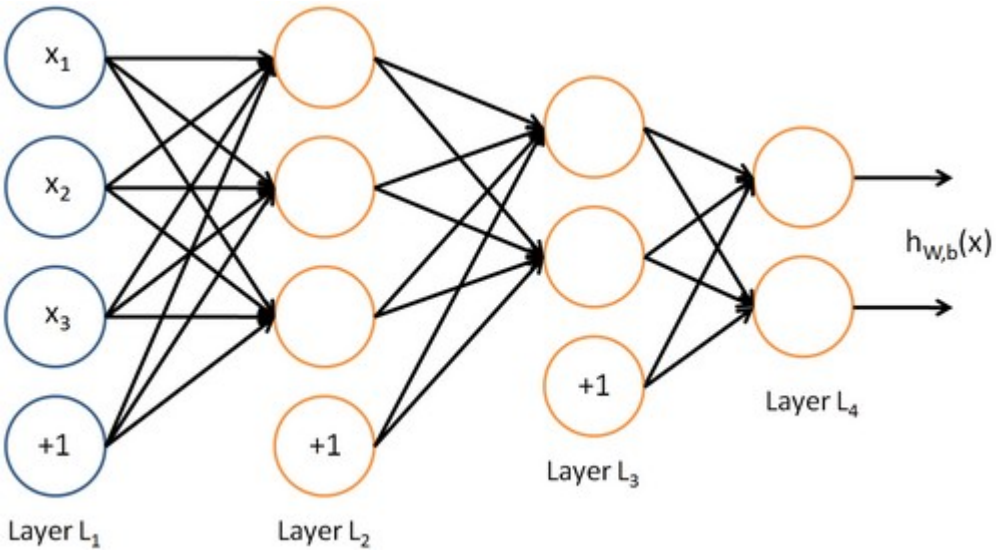


It is the loss function that will direct what the intermediate hidden variables should be, so as to do a good job at predicting the targets for the next layer, etc.

18

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

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



19

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.

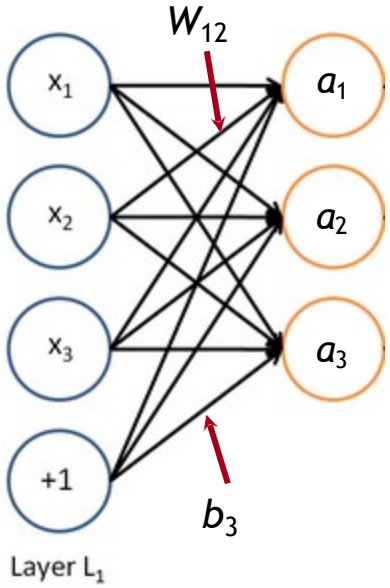
In matrix notation

$$z = Wx + b$$

$$a = f(z)$$

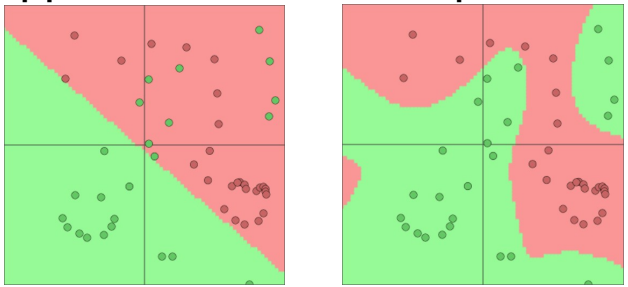
Activation f is applied element-wis

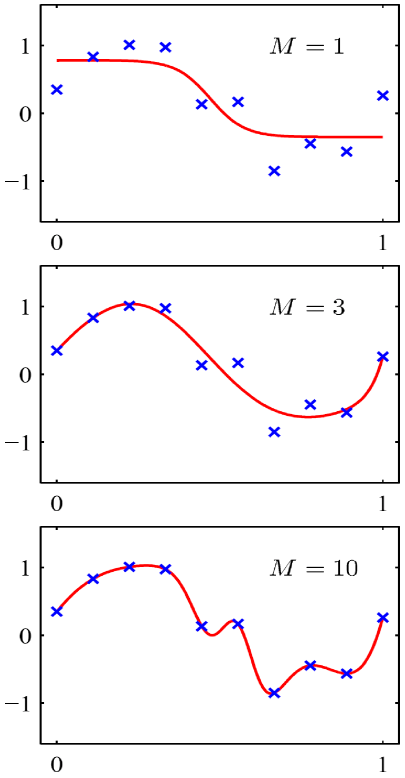
$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$



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!





Next: Named Entity Recognition
(as an example of Classification Task)

Named Entity Recognition (NER)

- The task: find and classify names in text, for example:

  The European Commission [ORG] said on Thursday it disagreed with German [MISC] advice.

  Only France [LOC] and Britain [LOC] backed Fischler [PER] 's proposal .

  "What we have to be extremely careful of is how other countries are going to take Germany 's lead", Welsh National Farmers ' Union [ORG] (NFU [ORG]) chairman John Lloyd Jones [PER] said on BBC [ORG] radio .

- Possible purposes:
  - Tracking mentions of particular entities in documents
  - For question answering, answers are usually named entities
  - A lot of wanted information is really associations between named entities
  - The same techniques can be extended to other slot-filling classifications
- Often followed by Named Entity Linking/Canonicalization into Knowledge Base

Named Entity Recognition on word sequences

We predict entities by classifying words in context and then extracting entities as word subsequences

Why might NER be hard?

- Hard to work out boundaries of entity

  First National Bank Donates 2 Vans To Future School Of Fort Smith

  POSTED 3:43 PM, JANUARY 11, 2019, BY 5NEWS WEB STAFF

  Is the first entity “First National Bank” or “National Bank”

- Hard to know if something is an entity

  Is there a school called “Future School” or is it a future school?

- Hard to know class of unknown/novel entity:

To find out more about Zig Ziglar and read features by other Creators Syndicate writers and

  What class is “Zig Ziglar”? (A person.)

- Entity class is ambiguous and depends on context

  “Charles Schwab” is PER not ORG here!

👉 where Larry Ellison and Charles Schwab can live discreetly amongst wooded estates. And

Word-Window classification

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

  "Museums in Paris are amazing"

  to classify whether or not the center word *"Paris"* is a named-entity

- For example, **Named Entity Classification** of a word in context:
  - Person, Location, Organization, None

- A simple way to classify a word in context might be to average the word vectors in a window and to classify the average vector
  - Problem: that would lose position information

Window classificaiton: softmax

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

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

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

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

Simplest window classifier: Softmax

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

predicted model output probability

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

- With cross entropy error:

same

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

- How do you update the word vectors?
- Short answer: Just take derivatives and optimize

Slightly more complex: Multilayer Perceptron

- Introduce an additional layer in our softmax classifier with a non-linearity.
- MLPs are fundamental building blocks of more complex neural systems!

Binary classification with unnormalized scores

Method used by Collobert & Weston (2008, 2011)

- For 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

- 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.

- E.g., the positions that have an actual NER Location in their center are “true” positions and get a high score

Binary classification for NER Location

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.

museums in Paris are amazing

• “Corrupt“ windows are easy to find and there are many: Any window whose center word isn’t specifically labeled as NER location in our corpus

Not all museums in Paris

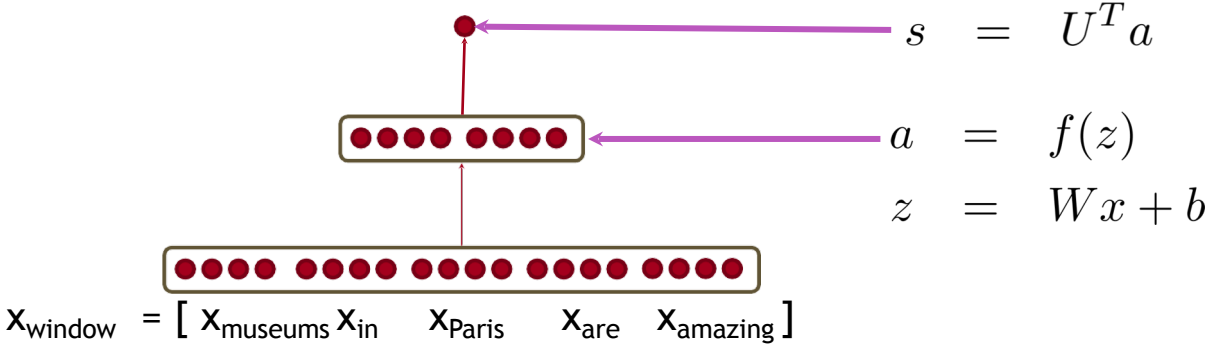
Neural Network Feed-forward Computation

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

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

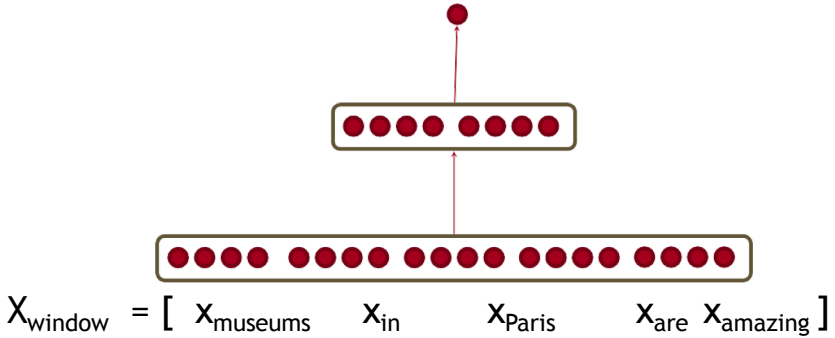
- *s* = *score*("museums in Paris are amazing")

$$s = U^T f(Wx + b)$$
$$x \in \mathbb{R}^{20\times 1}, W \in \mathbb{R}^{8x20}, U \in \mathbb{R}^{8x2}$$



Main intuition for extra layer

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



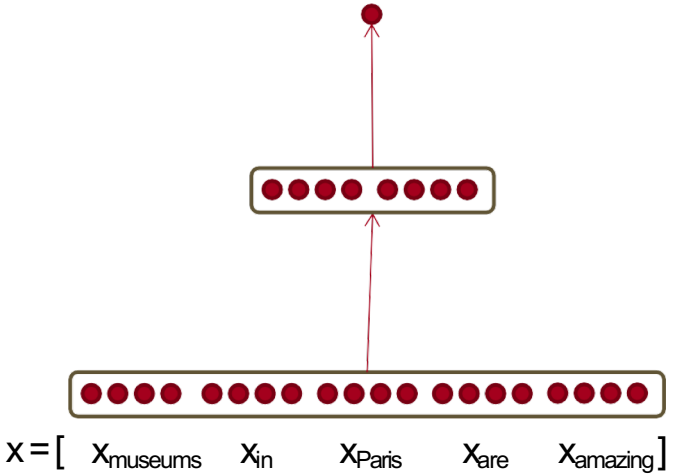
Example: only if "*museums*" is first vector should it matter that "*in*" is in the second position

Main model

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

$$\boldsymbol{x} \quad \text{(input)}$$



Next: Deep Learning Computation – Key Concepts

Deep Learning Computation: Some Key Concepts

Backpropagation

Regularization

Vectorization

Non-linearities

Initialization

Optimizers

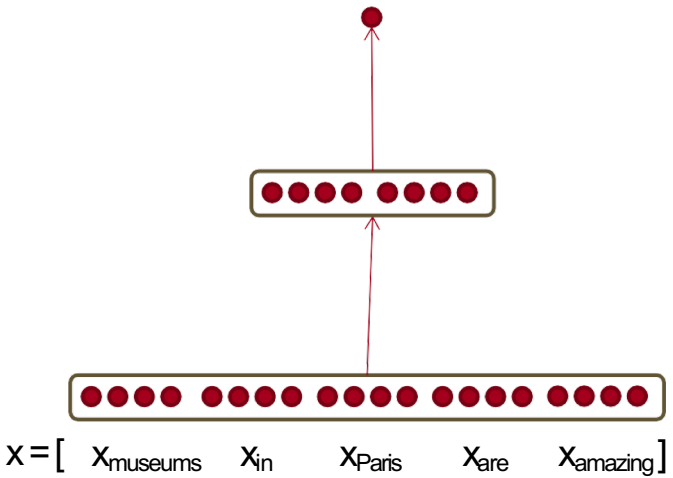
Learning Rates

...

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

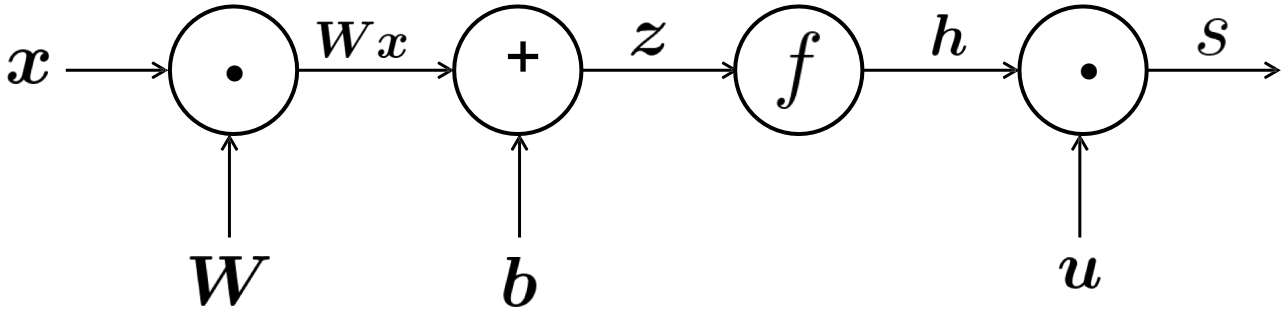
$$\boldsymbol{x} \quad (\text{input})$$



Computation Graphs and Backpropagation

- We represent our neural net equations as a graph
  - Source nodes: inputs
  - Interior nodes: operations
  - Edges pass along result of the operation

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad (\text{input})$$



Computation Graphs and Backpropagation

- We represent our neural net equations as a graph

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

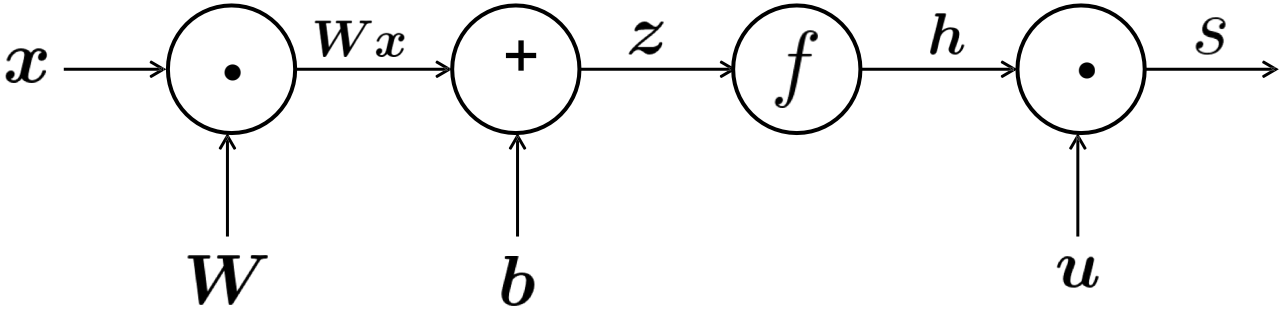
$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$+ \boldsymbol{b}$$

$$\text{ut})$$

"Forward Propagation"

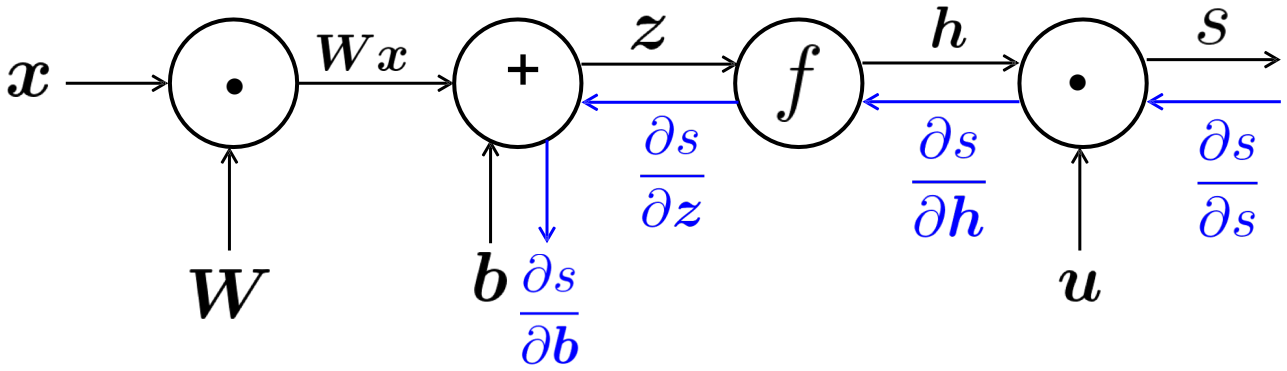
  - Edges pass along result of the operation



Backpropagation

- Go backwards along edges
  - Pass along **gradients**

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



Backprop key concepts

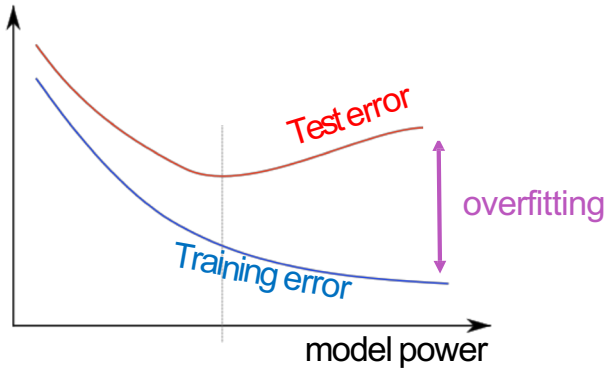
- Backpropagation: recursively (and hence efficiently) apply the chain rule along computation graph
  - [downstream gradient] = [upstream gradient] x [local gradient]
- Forward pass: compute results of operations and save intermediate values
- Backward pass: apply chain rule to compute gradients

We have models with many params: Regularization

- Really a full loss function in practice includes **regularization** over all parameters !, e.g., L2 regularization:

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

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



"Vectorization"

- E.g., looping over word vectors versus 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

“Vectorization”

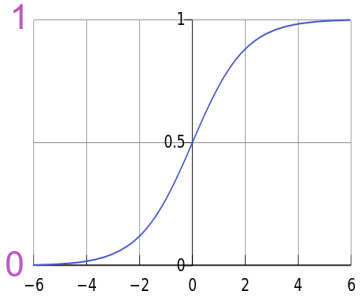
```
from numpy import random
N = 500 # number of windows to classify
d = 300 # dimensionality of each window
C = 5 # number of classes
W = random.rand(C,d)
wordvectors_list = [random.rand(d,1) for i in range(N)]
wordvectors_one_matrix = random.rand(d,N)

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

- The (10x) faster method is using a Cx N matrix
- Always try to use vectors and matrices rather than for loops!
- You should speed-test your code a lot too!!
- tl;dr: Matrices are awesome!!!

Non-linearities: The starting points

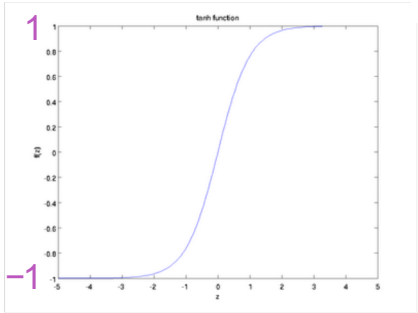
logistic ("sigmoid")

$$f(z) = \frac{1}{1 + \exp(-z)}.$$



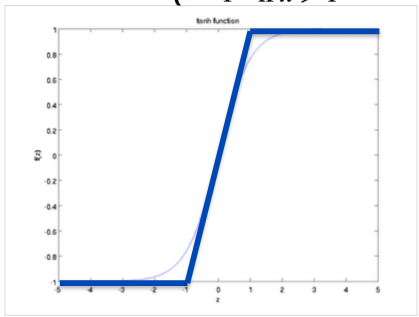
tanh

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



hard tanh

$$\text{HardTanh}(x) = \begin{cases} -1 & \text{if } x < -1 \\ x & \text{if } -1 <= x <= 1 \\ 1 & \text{if } x > 1 \end{cases}$$



tanh is just a rescaled and shifted sigmoid (2 x as steep, [−1,1]):

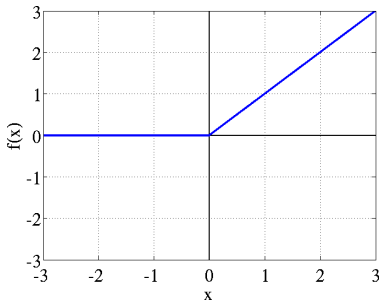
$$\tanh(z) = 2\text{logistic}(2z) - 1$$

Both logistic and tanh are still used in particular uses, but are no longer the defaults for making deep networks

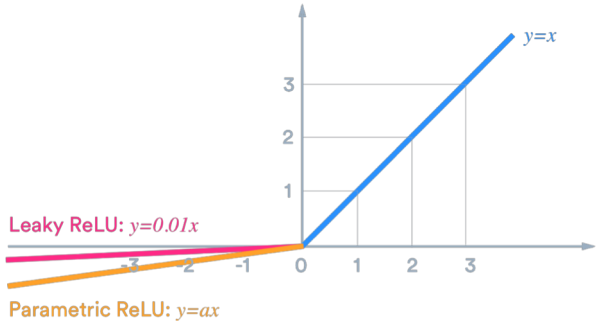
Non-linearities: The new world order

ReLU (rectified linear unit)

$$\text{rect}(z) = \max(z, 0)$$



Leaky ReLU



Parametric ReLU

- For building a feed-forward deep network, the first thing you should try is ReLU—it trains quickly and performs well due to good gradient backflow

Parameter Initialization

- You normally must initialize weights to small random values
  - To avoid symmetries that prevent learning/specialization
- 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 **all other weights** ~ Uniform(–*r*, *r*), with *r* chosen so numbers get neither too big or too small
- Xavier initialization has variance inversely proportional to fan-in n_{in} (previous layer size) and fan-out n_{out} (next layer size):

$$\mathrm{Var}(W_i) = \frac{2}{n_{\mathrm{in}} + n_{\mathrm{out}}}$$

Optimizers

- Usually, plain SGD will work just fine
  - However, getting good results will often require hand-tuning the learning rate (next slide)
- For more complex nets and situations, or just to avoid worry, you often do better with one of a family of more sophisticated "adaptive" optimizers that scale the parameter adjustment by an accumulated gradient.
  - These models give per-parameter learning rates
    - Adagrad
    - RMSprop
    - Adam A fairly good, safe place to begin in many cases
    - SparseAdam
    - …

Learning Rates

- You can just use a constant learning rate. Start around $lr = 0.001$?
  - It must be order of magnitude right – try powers of 10
    - Too big: model may diverge or not converge
    - Too small: your model may not have trained by the deadline
- Better results can generally be obtained by allowing learning rates to decrease as you train
  - By hand: halve the learning rate every *k* epochs
    - An epoch = a pass through the data (shuffled or sampled)
  - By a formula: $\eta_t = \eta_0 \alpha^t$, for epoch *t*
  - There are fancier methods like cyclic learning rates (q.v.)
- Fancier optimizers still use a learning rate but it may be an initial rate that the optimizer shrinks – so may be able to start high

Summary

Classification Tasks can successfully be addressed with Neural Networks because they are able to capture non-linearities

Named Entity Recognition can be addressed as a Classification Task

Deep Learning Computation is complex and full of tricks and details