Natural Language Processing with Deep Learning

Word Window Classification, Neural Networks, and Matrix Calculus
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: \(\{x_i,y_i\}_{i=1}^N\)

- Simple illustration case:
  - Fixed 2D word vectors to classify
  - Using softmax/logistic regression
  - Linear decision boundary

- **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

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

We can tease apart the prediction function into two steps:

1. Take the \( y^{th} \) row of \( W \) and multiply that row with \( x \):

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

Compute all \( f_c \) for \( c = 1, ..., C \)

2. Apply softmax function to get normalized probability:

\[ p(y|x) = \frac{\exp(f_y)}{\sum_{c=1}^{C} \exp(f_c)} = \text{softmax}(f_y) \]
Training with softmax and cross-entropy loss

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

- Or we can 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: 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: \( p = [0,\ldots,0,1,0,\ldots] \) 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:

\[
f = W \cdot x
\]
Traditional ML optimization

- For general machine learning $\theta$ usually only consists of columns of $W$:
  \[
  \theta = \begin{bmatrix}
  W_{.1} \\
  \vdots \\
  W_{.d}
  \end{bmatrix} = W(:,) \in \mathbb{R}^{Cd}
  \]

- So we only update the decision boundary via
  \[
  \nabla_{\theta} J(\theta) = \begin{bmatrix}
  \nabla W_{.1} \\
  \vdots \\
  \nabla W_{.d}
  \end{bmatrix} \in \mathbb{R}^{Cd}
  \]
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!
  • In original space
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 via layer $x = Le$

\[
\nabla_\theta J(\theta) = \begin{bmatrix}
\nabla W_1 \\
\vdots \\
\nabla W_d \\
\nabla x_{aardvark} \\
\vdots \\
\nabla x_{zebra}
\end{bmatrix} \in \mathbb{R}^{Cd+Vd}
\]

Very large number of parameters!
An artificial neuron

- Neural networks come with their own terminological baggage
- But if you understand how softmax models work, then you can easily understand the operation of a neuron!
A neuron can be a binary logistic regression unit

\[ h_{w,b}(x) = f(w^T x + b) \]

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

\( f \): 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

\[ z = Wx + b \]
\[ a = f(z) \]

Activation \( f \) 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!
The European Commission said on Thursday it disagreed with German advice. Only France and Britain backed Fischler’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 (NFU) chairman John Lloyd Jones said on BBC 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.

<table>
<thead>
<tr>
<th>Foreign</th>
<th>ORG</th>
<th>}</th>
<th>B-ORG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ministry</td>
<td>ORG</td>
<td>}</td>
<td>I-ORG</td>
</tr>
<tr>
<td>spokesman</td>
<td>O</td>
<td></td>
<td>O</td>
</tr>
<tr>
<td>Shen</td>
<td>PER</td>
<td>}</td>
<td>B-PER</td>
</tr>
<tr>
<td>Guofang</td>
<td>PER</td>
<td>}</td>
<td>I-PER</td>
</tr>
<tr>
<td>told</td>
<td>O</td>
<td></td>
<td>O</td>
</tr>
<tr>
<td>Reuters</td>
<td>ORG</td>
<td>}</td>
<td>B-ORG</td>
</tr>
<tr>
<td>that</td>
<td>O</td>
<td></td>
<td>O</td>
</tr>
</tbody>
</table>

👨‍💼 BIO encoding
Why might NER be hard?

- Hard to work out boundaries of entity
- Hard to know if something is an entity
- Hard to know class of unknown/novel entity:

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

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

  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
Binary word window classification

• In general, 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: resolving linking of ambiguous named entities:
  • Paris → Paris, France vs. Paris Hilton vs. Paris, Texas
  • Hathaway → Berkshire Hathaway vs. Anne Hathaway
Window classification

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

• 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 classification: 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:

\[
\begin{align*}
\text{... museums in Paris are amazing ... .} \\
\mathbf{x}_{\text{window}} &= [ x_{\text{museums}} \quad x_{\text{in}} \quad x_{\text{Paris}} \quad x_{\text{are}} \quad x_{\text{amazing}} ]^T
\end{align*}
\]

• Resulting vector \( \mathbf{x}_{\text{window}} \in \mathbb{R}^{5d} \), a column vector!
Simplest window classifier: Softmax

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

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

- With cross entropy error as before:

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

- How do you update the word vectors?
- Short answer: Just take derivatives and optimize
Binary classification with unnormalized scores

Method used by Collobert & Weston (2008, 2011)

• Just recently won ICML 2018 Test of time award

• For our previous example:
  \[ X_{\text{window}} = [ \ x_{\text{museums}} \ x_{\text{in}} \ x_{\text{Paris}} \ x_{\text{are}} \ x_{\text{amazing}} \ ] \]

• 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

Use neural activation $a$ simply to give an unnormalized score

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

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

- $s = \text{score}("museums in Paris are amazing")$

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

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

$$x_{\text{window}} = [x_{\text{museums}}, x_{\text{in}}, x_{\text{Paris}}, x_{\text{are}}, x_{\text{amazing}}]$$
Main intuition for extra layer

The middle layer learns \textit{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
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)
- $s = \text{score}(\text{museums in Paris are amazing})$
- $s_c = \text{score}(\text{Not all museums in Paris})$
- **Minimize**
  $$J = \max(0, 1 - s + s_c)$$
- This is not differentiable but it is continuous $\rightarrow$ we can use SGD.
Max-margin loss

- Objective for a single window:

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

- Each window with an NER location at its center should have a score +1 higher than any window without a location at its center

- For full objective function: Sample several corrupt windows per true one. Sum over all training windows.

- Similar to negative sampling in word2vec
Simple net for score

\[ s = u^T h \]

\[ h = f(Wx + b) \]

\[ x \quad (input) \]

\[ x = [x_{\text{museums}}, x_{\text{in}}, x_{\text{paris}}, x_{\text{are}}, x_{\text{amazing}}] \]
Remember: Stochastic Gradient Descent

- Update equation:
  \[ \theta^{new} = \theta^{old} - \alpha \nabla_{\theta} J(\theta) \]
  \(\alpha = \text{step size or learning rate}\)

- How do we compute \(\nabla_{\theta} J(\theta)\)?
  - By hand
  - Algorithmically: the backpropagation algorithm
Computing Gradients by Hand

- Review of multivariable derivatives
- Matrix calculus: Fully vectorized gradients
  - Much faster and more useful than non-vectorized gradients
  - But doing a non-vectorized gradient can be good practice
Backpropagation

Backpropagation - automatic differentiation

- Computing gradients algorithmically and efficiently
- Converting what we do by hand into an algorithm
- Used by deep learning software frameworks (TensorFlow, PyTorch, Chainer, etc.)
Derivative wrt a weight matrix

• Let’s look carefully at computing \( \frac{\partial s}{\partial W} \)

• Using the chain rule:

\[
\frac{\partial s}{\partial W} = \frac{\partial s}{\partial h} \frac{\partial h}{\partial z} \frac{\partial z}{\partial W}
\]

\[
s = u^T h
\]

\[
h = f(z)
\]

\[
z = Wx + b
\]

\[
x = [x_{\text{museums}} \ x_{\text{in}} \ x_{\text{Paris}} \ x_{\text{are}} \ x_{\text{amazing}}]
\]
Deriving gradients for backprop

• For this function (following on from last time):

\[
\frac{\partial s}{\partial W} = \delta \frac{\partial z}{\partial W} = \delta \frac{\partial}{\partial W} Wx + b
\]

• Let’s consider the derivative of a single weight \( W_{ij} \)

• \( W_{ij} \) only contributes to \( z_i \)
  • For example: \( W_{23} \) is only used to compute \( z_2 \) not \( z_1 \)

\[
\frac{\partial z_i}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}} W_i.x + b_i = \frac{\partial}{\partial W_{ij}} \sum_{k=1}^{d} W_{ik}x_k = x_j
\]
Deriving gradients for backprop

- So for derivative of single $W_{ij}$:
  \[
  \frac{\partial s}{\partial W_{ij}} = \delta_i x_j
  \]

- We want gradient for full $W$ – but each case is the same

- Overall answer: Outer product:
  \[
  \frac{\partial s}{\partial W} = \delta^T \cdot x^T \quad [n \times m] \cdot [n \times 1] = [1 \times m]
  \]
Deriving gradients: Tips

- **Tip 1:** Carefully define your variables and keep track of their dimensionality!
- **Tip 2:** Chain rule! If $y = f(u)$ and $u = g(x)$, i.e., $y = f(g(x))$, then:
  \[
  \frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \frac{\partial u}{\partial x}
  \]
  Keep straight what variables feed into what computations
- **Tip 3:** For the top softmax part of a model: First consider the derivative wrt $f_c$ when $c = y$ (the correct class), then consider derivative wrt $f_c$ when $c \neq y$ (all the incorrect classes)
- **Tip 4:** Work out element-wise partial derivatives if you’re getting confused by matrix calculus!
- **Tip 5:** The error message $\delta$ that arrives at a hidden layer has the same dimensionality as that hidden layer
Deriving gradients wrt words 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 $x_{window} = [ x_{museums} \ x_{in} \ x_{Paris} \ x_{are} \ x_{amazing} ]$

- 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 word gradients in window model

- This will push word vectors around so that they will (in principle) be more helpful in determining named entities.

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

![Diagram showing the similarity of TV, telly, and television in word vectors]

- **Question:** What happens when we update the word vectors?
A pitfall when retraining word vectors

• **Question:** What happens when we update the word vectors?
• **Answer:**
  • Those words that are in the training data move around
    • “TV” and “telly”
  • Words not in the training data stay where they were
    • “television”

This can be bad!
So what should I do?

**Question:** Should I use available “pre-trained” word vectors?

**Answer:**
- Almost always, yes!
- They are trained on a huge amount of data, and so they will know about words not in your training data and will know more about words that are in your training data.

**Question:** Should I update (“fine tune”) 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 large dataset, it probably will work better to train = update = fine-tune word vectors to the task.
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

\[
s = u^T h \\
h = f(z) \\
z = Wx + b \\
x \quad \text{(input)}
\]
Backpropagation

• Go backwards along edges
  • Pass along gradients

\[
s = u^T h \\
h = f(z) \\
z = Wx + b \\
x \text{ (input)}
\]
Backpropagation: Single Node

- Each node has a **local gradient**
- The gradient of its output with respect to its input

\[
h = f(z)
\]

\[
\frac{\partial s}{\partial z} = \frac{\partial s}{\partial h} \cdot \frac{\partial h}{\partial z}
\]

- [downstream gradient] = [upstream gradient] x [local gradient]
Backpropagation: Single Node

- Multiple inputs → multiple local gradients

\[ z = Wx \]

\[ \frac{\partial s}{\partial W} = \frac{\partial s}{\partial z} \frac{\partial z}{\partial W} \]

\[ \frac{\partial s}{\partial x} = \frac{\partial s}{\partial z} \frac{\partial z}{\partial x} \]

- Downstream gradients
- Local gradients
- Upstream gradient
Gradients sum at outward branches

\[ a = x + y \]
\[ b = \max(y, z) \]
\[ f = ab \]

\[ \frac{\partial f}{\partial y} = \frac{\partial f}{\partial a} \frac{\partial a}{\partial y} + \frac{\partial f}{\partial b} \frac{\partial b}{\partial y} \]
An Example

Forward prop steps
\[ a = x + y \]
\[ b = \max(y, z) \]
\[ f = ab \]

Local gradients
\[ \frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1 \]
\[ \frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0 \]
\[ \frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3 \]

\[ f(x, y, z) = (x + y) \max(y, z) \]
\[ x = 1, y = 2, z = 0 \]

---

Graph:
- \( x \) to \( + \)
- \( y \) to \( + \)
- \( z \) to \( \text{max} \)
- \( + \) to \( * \)
- \( * \) to output

Edges:
- 1 from \( x \) to \( + \)
- 2 from \( y \) to \( + \)
- 3 from \( z \) to \( \text{max} \)
- 3 from \( \text{max} \) to \( * \)
- 6 from \( * \) to output
Node Intuitions

\[ f(x, y, z) = (x + y) \max(y, z) \]
\[ x = 1, y = 2, z = 0 \]

- + “distributes” the upstream gradient
- max “routes” the upstream gradient
- * “switches” the upstream gradient

\[ x \]
\[ y \]
\[ z \]
Efficiency: compute all gradients at once

- Correct way:
  - Compute all the gradients at once
  - Analogous to using $\delta$ when we computed gradients by hand

\[ s = u^T h \]
\[ h = f(z) \]
\[ z = Wx + b \]
\[ x \text{ (input)} \]
1. Fprop: visit nodes in topological sort order
   - Compute value of node given predecessors
2. Bprop:
   - initialize output gradient = 1
   - visit nodes in reverse order:
     Compute gradient wrt each node using gradient wrt successors
     \[
     \{y_1, y_2, \ldots, y_n\} = \text{successors of } x
     \]
     \[
     \frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}
     \]

Done correctly, big O() complexity of fprop and bprop is \textbf{the same}

In general our nets have regular layer-structure and so we can use matrices and Jacobians...
Automatic Differentiation

- The gradient computation can be automatically inferred from the symbolic expression of the fprop.
- Each node type needs to know how to compute its output and how to compute the gradient wrt its inputs given the gradient wrt its output.
- Modern DL frameworks (Tensorflow, PyTorch, etc.) do backpropagation for you but mainly leave layer/node writer to hand-calculate the local derivative.
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
Implementation: forward/backward API

\[(x, y, z) \text{ are scalars}\]

```python
class MultiplyGate(object):
    def forward(x, y):
        z = x * y
        self.x = x  # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz  # [dz/dx * dL/dz]
        dy = self.x * dz  # [dz/dy * dL/dz]
        return [dx, dy]
```
Gradient checking: Numeric Gradient

- For small $h$ ($≈ 1e-4$), $f'(x) ≈ \frac{f(x + h) - f(x - h)}{2h}$
- Easy to implement correctly
- But approximate and very slow:
  - Have to recompute $f$ for every parameter of our model
- Useful for checking your implementation
  - In the old days when we hand-wrote everything, it was key to do this everywhere.
  - Now much less needed, when throwing together layers
Summary

• We’ve mastered the core technology of neural nets!!!

• Backpropagation: recursively apply the chain rule along computation graph
  • \([\text{downstream gradient}] = [\text{upstream gradient}] \times [\text{local gradient}]\)

• Forward pass: compute results of operations and save intermediate values

• Backward pass: apply chain rule to compute gradients
Why learn all these details about gradients?

- Modern deep learning frameworks compute gradients for you
- But why take a class on compilers or systems when they are implemented for you?
  - Understanding what is going on under the hood is useful!
- Backpropagation doesn’t always work perfectly.
  - Understanding why is crucial for debugging and improving models
  - See Karpathy article:
    - [https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b](https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b)
  - Example in future lecture: exploding and vanishing gradients
We have models with many params! Regularization!

- Really a full loss function in practice includes regularization over all parameters $\theta$, e.g., L2 regularization:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} -\log \left( \frac{e^{f_{yi}}}{\sum_{c=1}^{C} e^{f_{c}}} \right) + \lambda \sum_{k} \theta_{k}^{2}$$

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

ridge regression
L1 reg: lasso regression
“Vectorization”

- E.g., looping over word vectors versus concatenating them all into one large matrix and then multiplying the softmax weights with that matrix

```python
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 C x N matrix

Always try to use vectors and matrices rather than for loops!

You should speed-test your code a lot too!!
Non-linearities: The starting points

logistic ("sigmoid")

\[ f(z) = \frac{1}{1 + \exp(-z)} \]

\[ f(z) = \text{tanh}(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \]

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

tanh is just a rescaled and shifted sigmoid (2 \times as steep, \([-1,1]):

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

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) hard tanh

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

- 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):
  \[
  \text{Var}(W_i) = \frac{2}{n_{in} + n_{out}}
  \]
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: $lr = lr_0 e^{-kt}$, for epoch $t$
  • There are fancier methods like cyclic learning rates

• 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