0. Agenda

Wednesday, December 12, 2018 11:31 AM

1. Why Word Embedding (WE)? What is a WE? What are the different types of WE?
2. Pre-cursor to Word2Vec
3. Word2Vec -
    a. Basic Theory (no derivation + mathematical interpretation)
    b. Optimization Methods
    c. Pre-Pseudo Code Pointers
    d. Pseudo Code Implementation
4. Glove
    a. Basic Theory (no derivation + mathematical interpretation)
    b. Pseudo code Implementation
5. Creation and comparison of Word2vec vs Glove word vectors for a toy corpus
6. Pre-trained Word Vectors comparison

1. Why Word Embedding?

Thursday, December 13, 2018 6:19 PM

The most important problem in any NLP task is to how to represent the words

Easy

- Spell Checking
- Keyword Search
- Finding Synonyms

Medium

- Parsing information from websites, documents, etc.

Hard

- Machine Translation (e.g. Translate Chinese text to English)
- Semantic Analysis (What is the meaning of query statement?)
- Coreference (e.g. What does "he" or "it" refer to given a document?)
- Question Answering (e.g. Answering Jeopardy questions).

1.3 How to represent words?

The first and arguably most important common denominator across all NLP tasks is how we represent words as input to any of our models. Much of the earlier NLP work that we will not cover treats words as atomic symbols. To perform well on most NLP tasks we first need to have some notion of similarity and difference between words. With word vectors, we can quite easily encode this ability in the vectors themselves (using distance measures such as Jaccard, Cosine, Euclidean, etc).

1. What is Word Embedding?

Thursday, December 13, 2018 11:14 AM

1. **Simple Word Vector**:

In vector space terms, this is a vector with one 1 and a lot of zeroes

- One-hot encoded word vector representing the occurrence of a word.

Problem with such **discrete representations** - No 'semantic' meaning
Dot product (cosine similarity) between related words cannot be established.

Alternative in that case to find word meaning and relationships:

```
from nltk.corpus import wordnet as wn
```

```
good_synonyms=wn.synsets('good')
good_synonyms[0:10]
```

```
[Synset('good.n.01'),
 Synset('good.n.02'),
 Synset('good.n.03'),
 Synset('commodity.n.01'),
 Synset('good.a.01'),
 Synset('full.s.06'),
 Synset('good.a.03'),
 Synset('estimable.s.02'),
 Synset('beneficial.s.01'),
 Synset('good.s.06')]
```

```
good_synonyms[0].hypernyms()
```

```
[Synset('advantage.n.01')]
```

```
good_synonyms[3].hyponyms()
```

```
[Synset('basic.n.02'),
 Synset('consumer_goods.n.01'),
 Synset('drygoods.n.01'),
 Synset('entrant.n.01'),
 Synset('export.n.01'),
 Synset('fancy_goods.n.01'),
 Synset('fungible.n.01'),
 Synset('future.n.03'),
 Synset('import.n.01'),
 Synset('merchandise.n.01'),
 Synset('middling.n.01'),
 Synset('salvage.n.01'),
 Synset('shopping.n.02'),
 Synset('sporting_goods.n.01'),
 Synset('worldly_possession.n.01')]
```

Limitations:

- Human involvement to keep it up-to-date. And almost impossible to keep it up to date
- Subjective

Alternative: To look for word vectors that can bring out 'distributional similarity'

2. Before going to the alternatives to one-hot encoding type vectors,
   What is a word embedding then?

A word embedding is simply a matrix of stacked word vectors

	Favorite biology book	Favorite physics book
Electron	50	800
Newton	1	500
Energy	200	350
Mitochondria	400	0
Cell	600	30

Single word vector

Word embedding

- V = # of rows = vocabulary size = # of distinct words
- D = embedding dimension = feature vector size

Word Embedding
V x D

3. Different Types of Word Embedding:

Word Embedding (WE) ideas:
- ❑ Frequency/ BoW based WE (TFIDF, Countvec)
- ❑ Distributional Similarity based WE
  - ❑ Word units
    - ❑ Word2Vec
    - ❑ GloVe
  - ❑ Sub-word units
    - ❑ Fasttext
    - ❑ Byte pair encoding
    - ❑ Wordpiece Model
- ❑ Pre-trained vs Corpus-specific

Distributionally Semantic (Count co-occurrence or context) E.g.: LSA, LDA
vs
Distributionally Similar (predict context):

https://www.quora.com/What-does-distributional-semantic-model-D-S-M-mean-in-natural-language-processing
Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors:
http://anthology.aclweb.org/P/P14/P14-1023.pdf

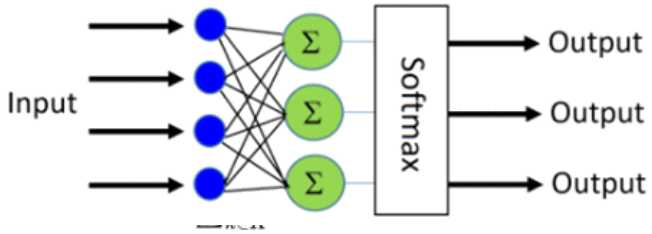
2. Pre-cursors to Word2Vec - LR Bigram Model

Thursday, December 13, 2018 12:00 PM

1. A Simple LR Bigram Model

If x is the current word vector and y is the next word vector,
Prob of y being the next word given the current word is x is given by

$$p(y \mid x) = softmax(W^T x)$$



$$p(y = j|\mathbf{x}) = \frac{e^{(\mathbf{w}_j^T \mathbf{x} + b_j)}}{\sum_{k \in K} e^{(\mathbf{w}_k^T \mathbf{x} + b_k)}}$$

Where j is one of the output words from a vocabulary size of K words

How do we find W?
- By doing **gradient descent on the cost function**
- What is a cost function then?

Cost function/ Log loss of a binary logistic regression is a cross-entropy,

$$\sum_{(x,y)\in D} -y \log(y') - (1-y)\log(1-y')$$

y is the label in a labeled example. Since this is logistic regression, every value of y must either be 0 or 1.

y' is the predicted value (somewhere between 0 and 1), given the set of features in x.

Cost function in our case is: $J = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{V} y_{nk} log(p(y_{nk}|x_n))$

How to compute gradient descent on the cost function?

Initialize W to be a random weights matrix of size V x V, where V is the vocabulary size

Gradient Descent on J = Partial derivative of J with respect to W: $\nabla J = X^T(p(Y|X) - Y)$

Optimize W: $W \leftarrow W - \eta \nabla J, \; where \; \nabla J = X^T(p(Y|X) - Y)$

Where η is the learning rate

2. Pre-cursor to Word2Vec - NN Bigram Model

Thursday, December 13, 2018 12:18 PM

2. NN Bigram Model - Introducing the existence of "TWO" embeddings in a NN based Word vector

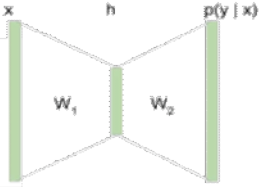
Logistic Regression based model

$$p(y \mid x) = softmax(W^T x)$$

Neural Network based Model

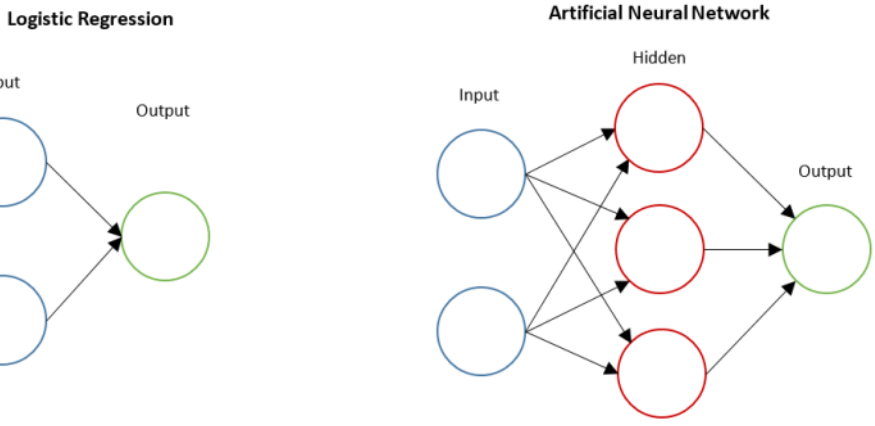
$$h = tanh(W_1{}^T x)$$

$$p(y \mid x) = softmax(W_2{}^T h)$$



In a NN, Using a softmax activation function at the output layer is basically equivalent to a Logistic Regression but additionally has a hidden layer unit

LR vs NN:



Cost function is the same for the NN based model

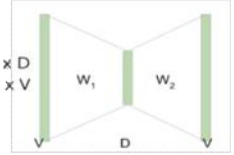
$$J = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{V} y_{n,k} log(p(y_{n,k} \mid x_n))$$

But the gradient descent on the cost function has two weight matrices W1 and W2

$$W_2 \leftarrow W_2 - \eta \nabla_{W_2} J \quad \nabla_{W_2} J = H^T (p(Y \mid X) - Y)$$

$$W_1 \leftarrow W_1 - \eta \nabla_{W_1} J \quad \nabla_{W_1} J = X^T \left[(p(Y \mid X) - Y) W_2{}^T \odot (1 - H^2) \right]$$

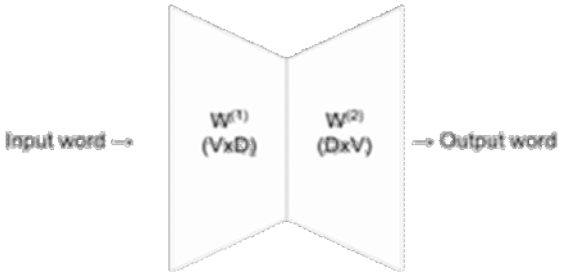
shape(W_1) == V x D
shape(W_2) == D x V



A typical Bigram NN model:

Typical Bigram model using softmax and any activation function f

$$p(x_{t+1} \mid x_t) = softmax(W^{(2)T} f(W^{(1)T} x_t))$$



3a. Word2Vec - Theory Basics

Thursday, December 13, 2018 12:00 PM

Main Idea of Word2Vec:

Predict between every word and its context words!

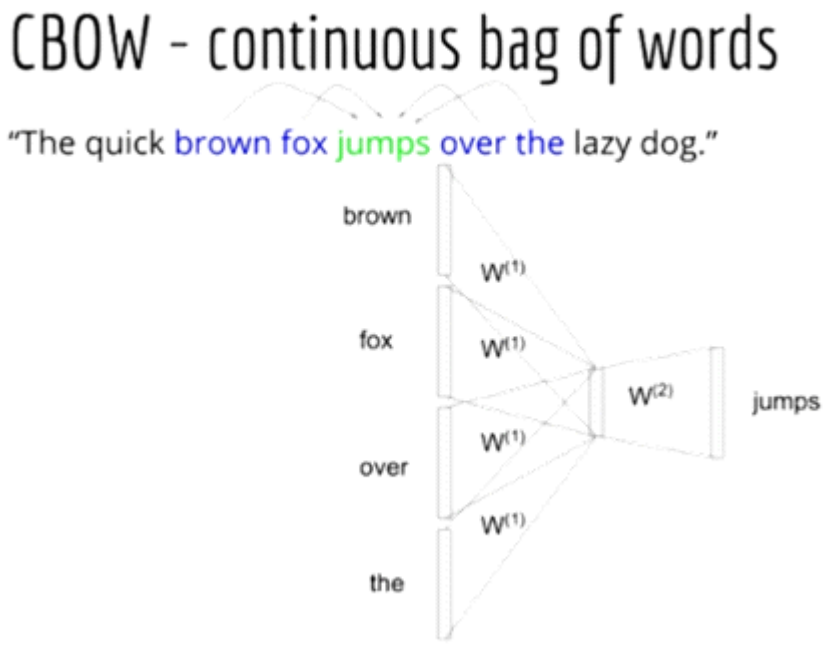
Two algorithms

1. **Skip-grams (SG)**
   Predict context words given target (position independent)
2. Continuous Bag of Words (CBOW)
   Predict target word from bag-of-words context

I enjoyed eating some pizza at the hotel

Target/Focus word

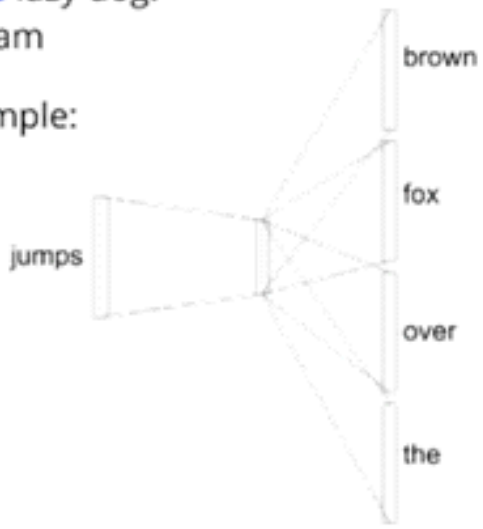
Neighboring words/ Context words



Given a window size of 2 words around a focus word - jumps, **the CBOW model predicts the current word *'jumps'*, given the neighboring words in the window**

Skipgram

- "The quick brown fox jumps over the lazy dog."
- Helpful to think of it in terms of bigram
- Bigram model gives us 1 training sample:
  jumps → over
- Skipgram gives us 3 additional training samples:
  jumps → brown
  jumps → fox
  jumps → the

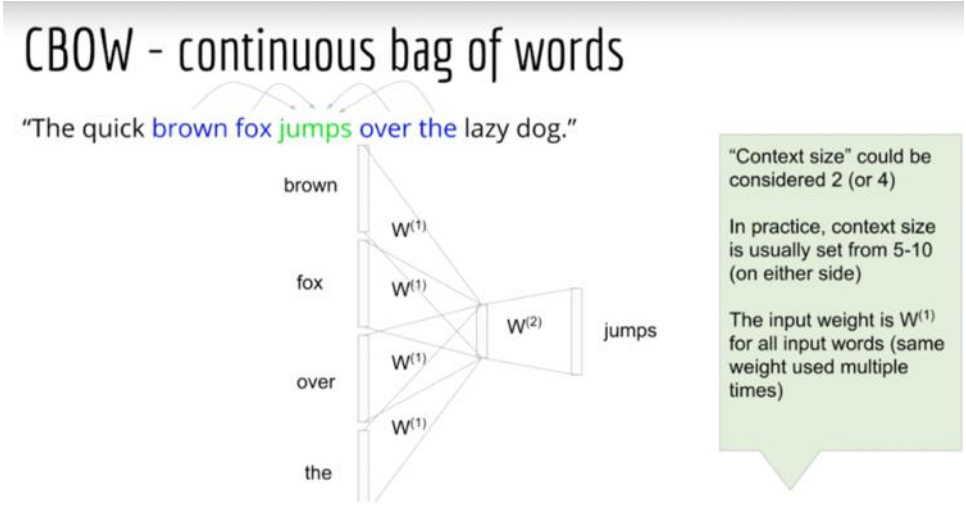


Given a window size of 2 words around a focus word, the **skip-gram model predicts the neighboring/context words given the current/focus word** (here – jumps).

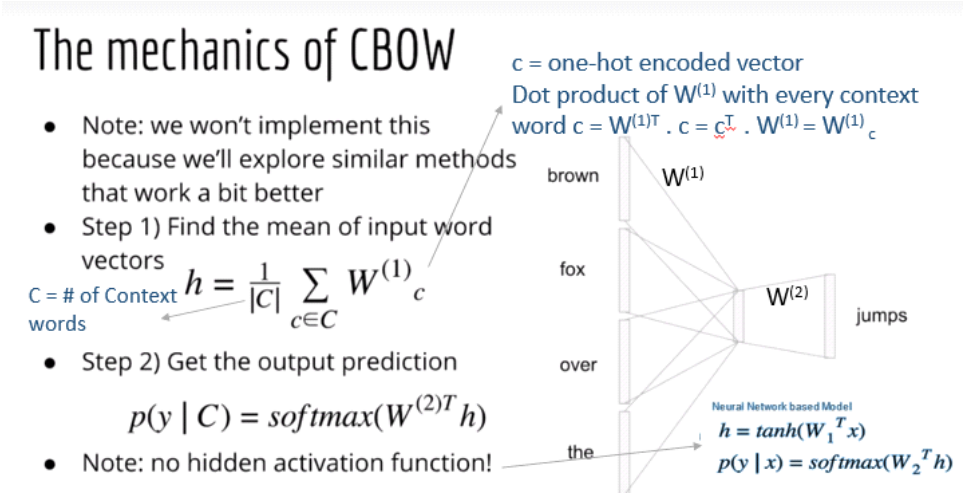
****

How CBOW works?

1. Context Size is one of the hyper parameters
   - Usual size - 5-10. We can widen the context words by dropping more frequent words like 'over' , 'the' (p-keep -- more on it later)

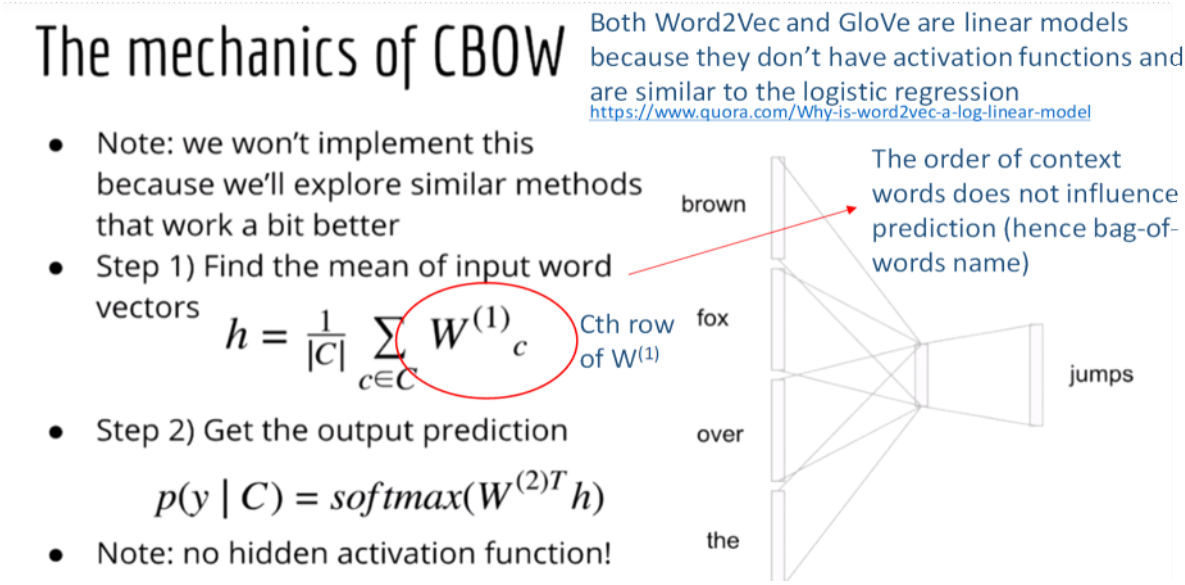


2. How CBOW is different from a typical NN bigram model? - No activation function



3. Compared to NN, Word2vec is a log-linear model

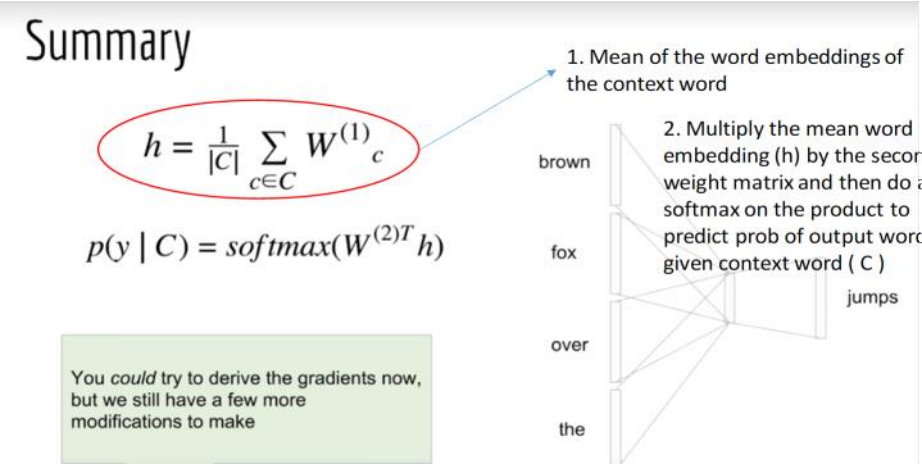
Neural Networks are non-linear because of activation function.



Why W2V log-linear?
"linear in log space"
To make training faster:

- Deep neural networks with **multiple stages of non-linear steps** are **slower to train** because during the backpropagation step we **need to propagate the gradient through each of the intermediate non-linear activation functions**.
- Word2vec being log-linear means we calculate the gradient at the output and then directly propagate this back into the embedding parameters (the main computational burden during training). This means faster trainer over bigger datasets yielding more accurate embedding vectors.

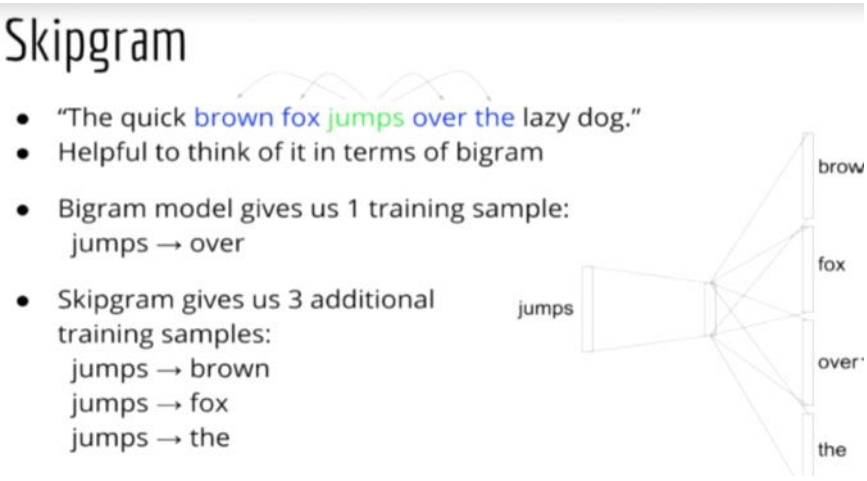
4. Summary of CBoW



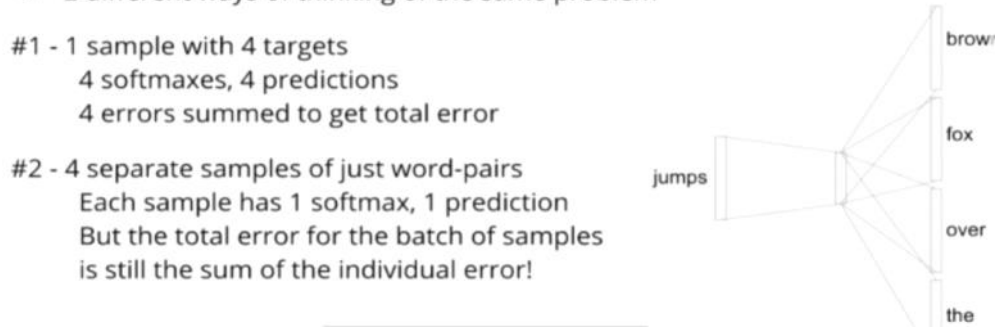
****

How does skip-gram work?

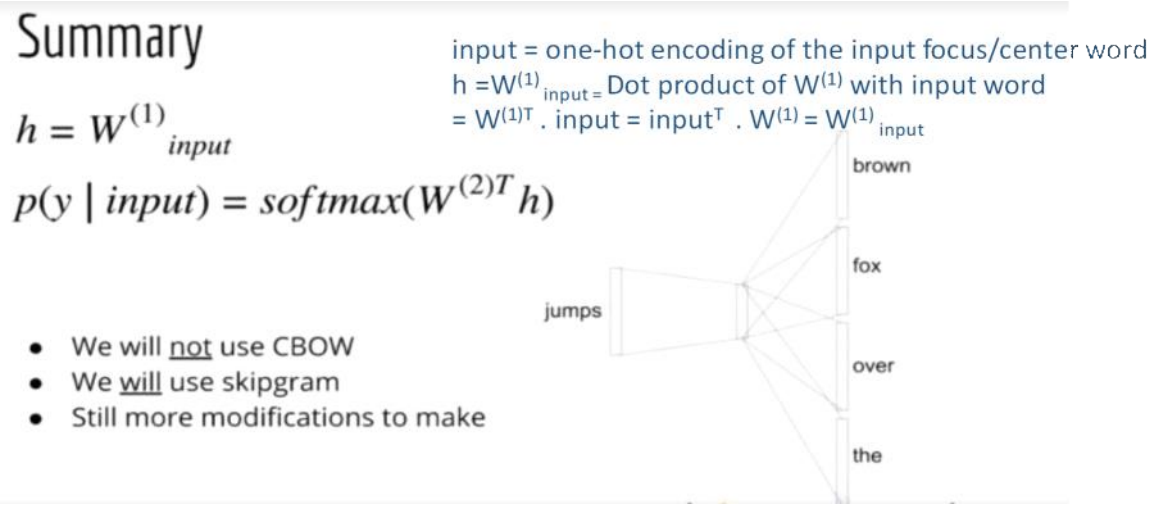
1. Skip-gram is like a bigram except that we "skip" a few words



2. 2 different ways of looking into the skip-gram problem



3.



****

Conclusion on the basic theory:
Skip-gram: works well with small amount of the training data, represents well even rare words or phrases.
CBOW: several times faster to train than the skip-gram, slightly better accuracy for the frequent words.

From <https://towardsdatascience.com/emnlp-what-is-glove-part-i-3b6ce6a7f970>

3b. Word2Vec - Optimization Methods - solving the cost function

Thursday, December 13, 2018 6:58 PM

Predict between every word and its context words!

Two algorithms

1. **Skip-grams (SG)**
   Predict context words given target (position independent)
2. Continuous Bag of Words (CBOW)
   Predict target word from bag-of-words context

Two (moderately efficient) training methods

1. Hierarchical softmax
2. Negative sampling

We'll see briefly about **Hierarchical softmax** and in-depth about **Negative Sampling.**
Reason: Negative Sampling method much more effective than Hierarchical softmax

****

Why ordinary softmax could be problematic?
V = vocab size = no. of output classes

Suppose a large dataset has 100K unique tokens as V

- Large number of output classes - affect accuracy
- P(output_word|center_word) = 1/ 100K = 99.99999% chance of failure
- Order of complexity of softmax calculation is O(V)

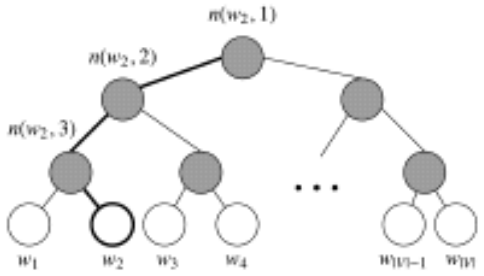
$$p(y = j \mid x) = \frac{exp(w_j{}^T x)}{\sum_{k=1}^{V} exp(w_k{}^T x)}$$

****

Hierarchical Softmax

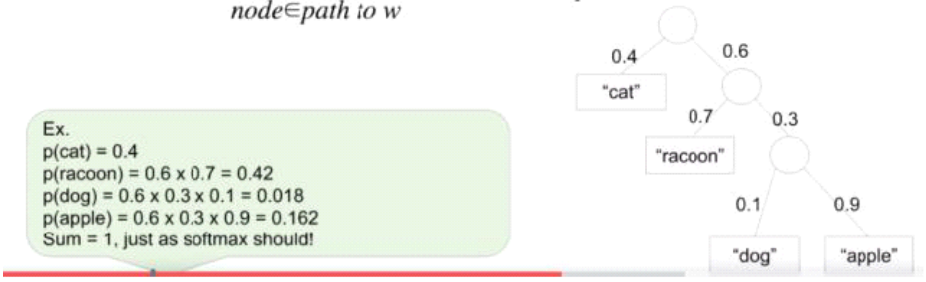
- An approximation to the normal softmax.
- Reduces the number of classes in the denominator
- Reduces the complexity from O(V) to $O(\log_2 V)$

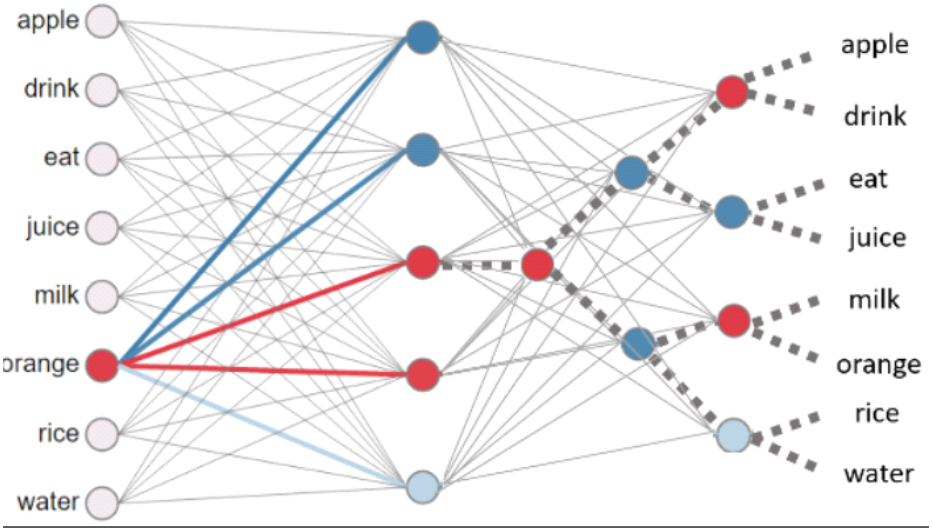
Hierarchical Softmax uses a binary tree where leaves are the words. The probability of a word being the output word is defined as the probability of a random walk from the root to that word's leaf. Computational cost becomes $O(\log(|V|))$ instead of O(|V|).



The sum of probabilities of all leaf nodes is 1

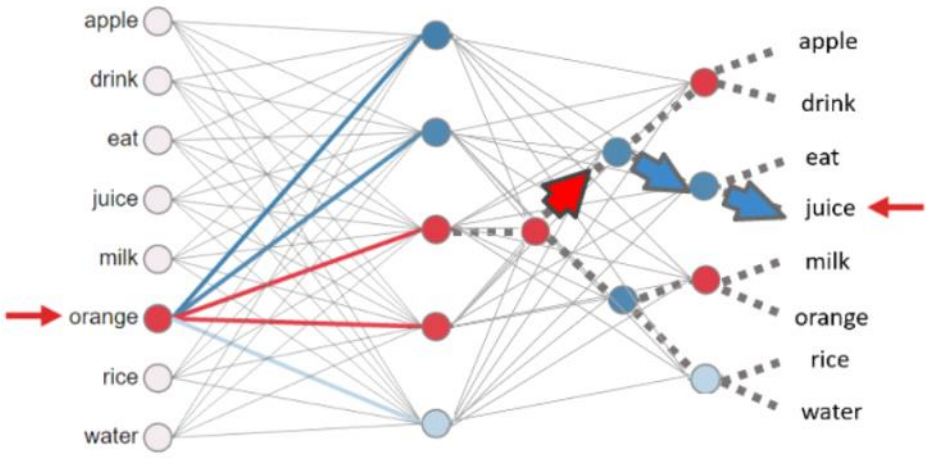
$$p(w \mid input) = \prod_{node \in path\ to\ w} \sigma(v_{node}{}^{T} h_{input})$$





instead of mapping each output vector to its corresponding word, we consider the output vector as a form of binary tree

the output vector is not making a prediction about how probable the word is, but it is making a prediction about which way you want to go in the binary tree. So, either you want to visit this branch or you want to visit the other branch



The error is propagated backwards to only those nodes that are activated at the time of prediction

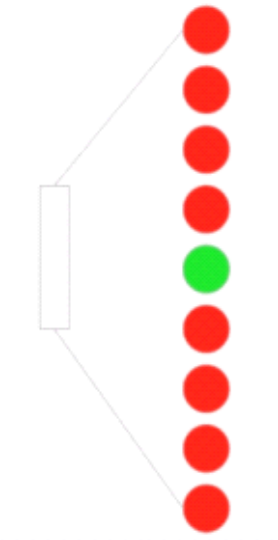
- To train the model, our goal is still to minimize the negative log Likelihood (since this also a softmax).
- But instead of updating all output vectors per word, we update the vectors of the nodes in the binary tree that are in the path from root to leaf node.
- "Huffman coding" is used to construct this binary tree.
    - Frequent words are closer to the top
    - Infrequent words are closer to the bottom

****

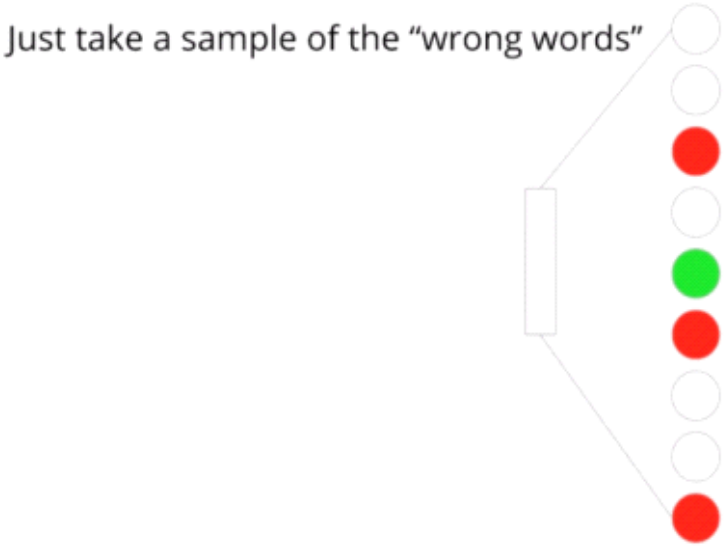
Negative Sampling:

- Negative sampling is also a kind of optimization method.
- To update the vector of the output word, but we are NOT going to update all the vectors of other words.
- Take the sample from the words other than the output vector.
- We are selecting a sample from the negative sample set of words, hence the name of this technique is negative sampling.

Usual Softmax:



Negative Sampling Softmax



A quick recap - Multiclass cross entropy vs Binary cross-entropy

Multiclass

$$p(y_{out} \mid x_{in}) = \frac{exp({W^{(2)}}_{out}{}^{T} W^{(1)}{}_{in})}{\sum_{j=1}^{V} exp({W^{(2)}}_{j}{}^{T} W^{(1)}{}_{in})}, \; J = t_{out} log p(y_{out} \mid x_{in})$$

Binary

$$p(y_{out} = 1 \mid x_{in}) = \sigma({W^{(2)}}_{out}{}^{T} W^{(1)}{}_{in})$$

$$J = t_{out} log p(y_{out} = 1 \mid x_{in}) + (1 - t_{out}) log(1 - p(y_{out} = 1 \mid x_{in}))$$

t_{out} is the actual output word in that position

Negative sign is missing in this pic for cost function J

For each training sample, the classifier is fed **a true pair (a center word and another word that appears in its context)** and **a number of k randomly corrupted pairs** (consisting of the center word and a randomly chosen word from the vocabulary). By learning to distinguish the true pairs from corrupted ones, the classifier will ultimately learn the word vectors.

This is important: instead of predicting the next word (the "standard" training technique), **the optimized classifier simply predicts whether a pair of words is good or bad**.

For Negative Sampling, we are taking word pairs with positive and negative class, hence taking the binary cross-entropy

Example

"The quick brown fox jumps over the lazy dog."

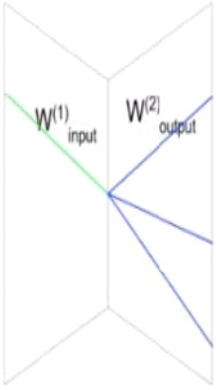
Input word: jumps
Target words: brown, fox, over, the
Negative samples: apple, orange, boat, tokyo

$$p(brown \mid jumps) = \sigma({W^{(2)}}_{brown}{}^{T} {W^{(1)}}_{jumps})$$
$$p(fox \mid jumps) = \sigma({W^{(2)}}_{fox}{}^{T} {W^{(1)}}_{jumps})$$
$$p(over \mid jumps) = \sigma({W^{(2)}}_{over}{}^{T} {W^{(1)}}_{jumps})$$
$$p(the \mid jumps) = \sigma({W^{(2)}}_{the}{}^{T} {W^{(1)}}_{jumps})$$
$$p(apple \mid jumps) = \sigma({W^{(2)}}_{apple}{}^{T} {W^{(1)}}_{jumps})$$
$$p(orange \mid jumps) = \sigma({W^{(2)}}_{orange}{}^{T} {W^{(1)}}_{jumps})$$
$$p(boat \mid jumps) = \sigma({W^{(2)}}_{boat}{}^{T} {W^{(1)}}_{jumps})$$
$$p(tokyo \mid jumps) = \sigma({W^{(2)}}_{tokyo}{}^{T} {W^{(1)}}_{jumps})$$

Example cont.

Input word: jumps
Target words: brown, fox, over, the
Negative samples: apple, orange, boat, tokyo

$$J = logp(brown \mid jumps) + logp(fox \mid jumps) + logp(over \mid jumps) + logp(the \mid jumps) + log[1 - p(apple \mid jumps)] + log[1 - p(orange \mid jumps)] +$$



Negative Sampling

- The objective in general:
  - C = set of context words
  - N = set of negative samples (these screens only)

$$J = \sum_{c \in C} log\sigma({W^{(2)}}_{c}{}^{T} {W^{(1)}}_{in}) + \sum_{n \in N} log[1 - \sigma({W^{(2)}}_{n}{}^{T} {W^{(1)}}_{in})]$$

Negative Sampling

- Another way of writing it

$$J = \sum_{c \in C} log\sigma({W^{(2)}}_c^T {W^{(1)}}_{in}) + \sum_{n \in N} log[1 - \sigma({W^{(2)}}_n^T {W^{(1)}}_{in})]$$

$$p(y = 1 \mid x) = \sigma(logit),\ p(y = 0 \mid x) = \sigma(-logit)$$
$$\sigma(logit) + \sigma(-logit) = 1$$

$$J = \sum_{c \in C} log\sigma({W^{(2)}}_c^T {W^{(1)}}_{in}) + \sum_{n \in N} log\sigma(-{W^{(2)}}_n^T {W^{(1)}}_{in})$$

Negative Sampling Gradient

The loss

$$J = -\sum_{n=1}^{N} t_n log p_n + (1 - t_n) log(1 - p_n)$$

Gradient wrt $W^{(2)}$ (In general)

$$\frac{\partial J}{\partial W^{(2)}} = H^T(P - T)$$

H = W(1)input = D, 1 dimension; P = an array of output probabilities. T = true values
Capital lettered characters represent full arrays.

For a single center/input word = jumps

$$\frac{\partial J}{\partial {W^{(2)}}_C} = np.outer({W^{(1)}}_{jumps}, (P_C - T_C)),\ C = \{brown, fox, over\}$$

For the whole input embedding:

$$\frac{\partial J}{\partial {W^{(1)}}_{in}} = \sum_{n=1}^{N} (p_n - t_n) {W^{(2)}}_{out(n)}$$

Hyper parameter for Negative Sampling - Number of Negative samples:

- Modified Unigram distribution

Negative Sampling - Details

- How many negative samples should you choose?
  - 5 or 10, even as high as 25
  - (Remember, with "true" softmax, there would be 2,999,999 negative samples!)

The paper says that selecting 5-20 words works well for smaller datasets, and you can get away with only 2-5 words for large datasets.

Unigram Distribution:

$$p(w) = \frac{count(w)}{\sum_{w'} count(w')}$$

Modified Unigram Distribution:

$$\tilde{p}(w) = \frac{count(w)^{0.75}}{\sum_{w'} count(w')^{0.75}}$$

The decision to raise the frequency to the 3/4 power appears to be empirical; in their paper they say it outperformed other functions.

Negative Sampling - Details

- Consider our running example: "The quick brown fox jumps over the lazy dog."
  - Input: jumps
  - Context: brown, fox, over, the
- What if you sample "the" again in your negative samples? Oops!
  - It's ok, treat it like a negative sample too
  - Too much work to ensure we don't accidentally sample context

Apart from Hierarchical Softmax and Negative Sampling:

Built-in Softmax Alternatives

```
tf.nn.nce_loss

tf.nn.sampled_softmax_loss
```

NCE_Loss:
Noise Contrastive Estimation (NCE)

a simplified variant of Noise Contrastive Estimation (NCE) is what Word2Vec paper calls Negative Sampling

https://datascience.stackexchange.com/questions/13216/intuitive-explanation-of-noise-contrastive-estimation-nce-loss

The basic idea is to convert a multinomial classification problem (as it is the problem of predicting the next word) to a binary classification problem. That is, instead of using softmax to estimate a true probability distribution of the output word, a binary logistic regression (binary classification) is used instead.

For each training sample, the enhanced (optimized) classifier is fed a true pair (a center word and another word that appears in its context) and a number of kk randomly corrupted pairs (consisting of the center word and a randomly chosen word from the vocabulary). By learning to distinguish the true pairs from corrupted ones, the classifier will ultimately learn the word vectors.

This is important: instead of predicting the next word (the "standard" training technique), the optimized classifier simply predicts whether a pair of words is good or bad.

3c. Word2Vec Pre-Pseudo code Pointers

Thursday, December 13, 2018 7:16 PM

Pre-implementation Points to Note:

1. Twist in Negative Sampling

Negative Sampling - Details

How you think it's going to work:

The quick brown fox jumps over the lazy dog.

Positive samples: jumps → brown, jumps → fox, ...

Negative samples: jumps → boat, jumps → tokyo, ...

Negative Sampling - Details

How it's actually going to work:

The quick brown fox jumps over the lazy dog.

The quick brown fox lighthouse over the lazy dog.

+ve samples: jumps → brown, jumps → fox, ...

-ve samples: lighthouse → brown, lighthouse → fox, ...

2. Dropping high frequency words increases context window

If we drop "over" and "the" so that:

"The quick brown fox jumps over the lazy dog" becomes

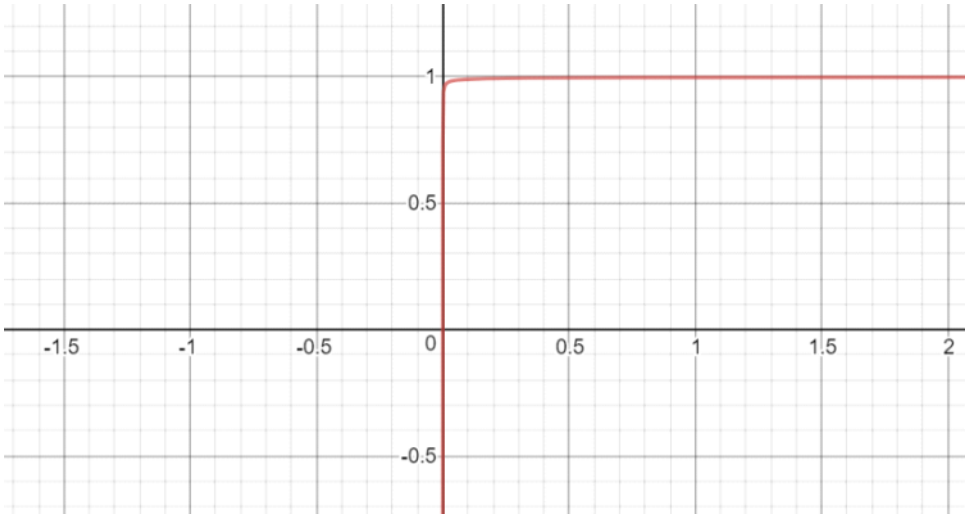
"Quick brown fox jumps lazy dog"

This *effectively* widens the context window

$$p_{keep}(w) = 1 - \sqrt{\frac{threshold}{\tilde{p}(w)}}$$

$\tilde{p}(w)$

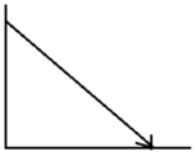
-- p'(w) is the prob of selecting a word (used for picking negative samples of context words -- in theory)



The higher the P_{keep}, the higher is the chance of keeping that word
The greater is the p'(w) (modified probability of a word) from threshold, greater is its chance of being dropped

3. Learning rate scheduling

Linearly decrease learning rate from max → min



3d. W2V - Pseudo Code Implementation

Friday, December 14, 2018 3:02 PM

Skip-gram W2V Implementation:

Main sources:
https://github.com/nathanrooy/word2vec-from-scratch-with-python/blob/master/word2vec.py
https://github.com/lazyprogrammer/machine_learning_examples/blob/master/nlp_class2/word2vec.py

(Skipping the sentence to word_index processing function)

word2idx = indices for every unique word in the vocabulary
sentences = series of sentences with words represented by the word2idx

Hyper parameters:

window_size = 5

learning_rate = 0.025
final_learning_rate = 0.0001
# learning rate decay
learning_rate_delta = (learning_rate - final_learning_rate) / epochs

num_negatives = 5 # number of negative samples to draw per input word
epochs = 20
D = 100 # word embedding size

Initializing W(1) and W(2) embeddings
Defining p(w) - probability of picking up a negative sample and prob of keeping a word

```
# initialize parameters
W(1) = random(V, D)
W(2) = random(D, V)

# calculate negative sampling dist
p(w) = count(w)^0.75 / sum(count(w)^0.75)

# calculate subsampling dist
p_keep (w) = 1 - sqrt(threshold / p(w))
```

Looping through every sentence in the corpus to calculate SGD

```
# loop through data
for epoch in epochs:
  for sentence in sentences:
    sentence = subsample(sentence, p_ keep )
    for middle_word in sentence:
      context = sentence[mid - window...mid + window]
      SGD(middle_word, context, label=1)
      neg_word = sample from p(w)
      SGD(neg_word, context, label=0)
```

SGD - stochastic gradient descent:

Vanilla gradient descent:

```
while True:
    theta_grad = evaluate_gradient(J,corpus,theta)
    theta = theta - alpha * theta_grad
```

Stochastic gradient descent:

```
while True:
    window = sample_window(corpus)
    theta_grad = evaluate_gradient(J,window,theta)
    theta = theta - alpha * theta_grad
```

#sigmoid function is like the forward pass
#gW(1) and gW(2) are the back propagation

```
def SGD(word, context, label):
  prob = sigmoid(W(1)[word] . W(2)[:,context])

  gW(2) = outer(W(1)[word], prob - label) # D x N
  gW(1) = W(2)[:,context] . (prob - label) # (D x N)(N) → D

  W(2)[:,context] -= lr * gW(2)
  W(1)[word] -= lr * gW(1)

  return -(label*log(prob) + (1-label)*log(1-prob)).sum()
```

In tensorflow implementation, instead of SGD based cost computation:

```
correct_output = dot(pos_input, emb_output)
pos_loss = tf.nn.sigmoid_cross_entropy_with_logits(
  labels=ones, logits=correct_output)

incorrect_output = dot(neg_input, emb_output)
neg_loss = tf.nn.sigmoid_cross_entropy_with_logits(
  labels=zeros, logits=incorrect_output)

loss = tf.reduce_mean(pos_loss) + tf.reduce_mean(neg_loss)
```

== (minus sign is missing)

$$J = \sum_{c \in C} log\sigma({W^{(2)}}_c{}^T {W^{(1)}}_{in}) + \sum_{n \in N} log[1 - \sigma({W^{(2)}}_n{}^T {W^{(1)}}_{in})]$$

3e. Showcase the W2V in TF

Thursday, January 10, 2019 1:36 PM

Local Location:
http://localhost:8888/notebooks/Desktop/Best%20Text%20Analytics%20Materials/Word%20Embedding%20Materials/Codes%20and%20Data/Word2Vec_tf.ipynb

4a. Matrix Factorization Theory

Thursday, January 17, 2019 11:55 AM

Section Outline

- To understand GloVe, we have to take a detour into Recommender Systems
- Key algorithm: matrix factorization
- Ratings data: users + movies + what users rated those movies
- We can predict what a user might rate a movie they haven't seen yet

We can then predict what those users might rate movies they haven't seen yet.

The User-Movie Ratings Matrix

- N users (N rows)
- M items (M columns)
- shape(R) = N x M

	Movie 1	Movie 2	Movie 3
Ted	4	5	5
Carol		5	5
Bob		5	?

the roads and the movies go along the columns.

Sparsity is good!

- Recommender system: we want to recommend a movie you might like, that you haven't seen yet
- If R isn't sparse, that means you've already seen and rated every movie!
- Would have nothing to recommend

User / Item	Batman	Star Wars	Titanic
Bill	3	3	
Jane		2	4
Tom		5	

Collaborative Filtering

- MF falls into a class of algorithms called collaborative filtering
- It's collaborative because we use other users' data to help make predictions for you

	Batman	X-Men	Star Wars	The Notebook	Bridget Jones's Diary
Alice	5	4.5	5	2	1
Bob	4.5	4		2	2
Carol	2	3	1	5	5

as an example of a class of algorithms known as collaborative filtering.

How do we exploit redundancy?

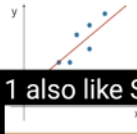
- We want to reduce the dimensionality of the data (dimensionality reduction)
- One such method is called SVD (singular value decomposition)

$$X = USV^T$$

In all these cases we can exploit the property of redundancy to build a model that reduces the dimensionality

Key Property: Redundancy

- In Machine Learning, 2 features are redundant if they are correlated with each other (we can predict one from the other)
- E.g. one feature is height in cm, one feature is height in inches
  - One completely determines the other!
- Documents are also redundant
  - Do I need 1 million documents to tell me "quantum" and "gravity" are related?
- Movie ratings can be redundant
  - If everyone who loves Star Wars Episode 1 also loves Star Wars Episode 2, I don't need to know the rating for both movies



Suppose that everyone who likes Star Wars Episode 1 also like Star Wars Episode 2 then it would be easy

How is it "reduced"?

- U, S, and V together take up more space than X!

$$X_{N\times D} = U_{N\times D} S_{D\times D} V^T{}_{D\times D}$$

So what exactly makes this a dimensionality reduction.

We can "chop off" parts of U, S, V

- Choose K << D
- Equality is now only approximate

$$\hat{X}_{N\times D} = U_{N\times K} S_{K\times K} V^T{}_{K\times D}$$

Quick calculations

- N = 1 million, D = 500,000, K = 10
- size(X) = NxD = 10^6 x 5x10^5 = 5x10^{11}
- size(U) = NxK = 10 million
- size(S) = 10x10 = 100
- size(V) = 500,000x10 = 5 million
- 15 million / 500 billion = 3x10^{-5}

$$\hat{X}_{N\times D} = U_{N\times K} S_{K\times K} V^T{}_{K\times D}$$

Let's put some real numbers on these dimensions so you get some sense of the savings we get let's say

Matrix Factorization

- SVD lets us know it's possible to factorize a matrix
- Examples of factors:
  - 15 = 3 x 5
  - 10 = 5 x 2
- We want to factorize R into just 2 matrices

$$R \approx \hat{R} = W U^T$$

What are the savings?

- N=100,000, M=20,000
- size(R) = 10^5 x $2x10^4$ = 2 billion
- 20 million ratings (e.g. MovieLens dataset)
- 20 million / 2 billion = 0.01
  - 99% empty!
  - In reality it's even more sparse
- K = 10
- size(W) = N x K = 1 million
- size(U) = M x K = 200,000
- 1.2 million parameters / 20 million ratings = 6%

$$R \approx \hat{R} = W U^T$$

How do we use the model?

- Our prediction matrix has no missing values!

$$R \approx \hat{R} = W U^T$$

- Each individual prediction is just a simple dot product between 2 K-length vectors

$$\hat{r}_{ij} = W_i^T U_j$$

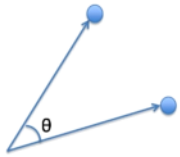
- Note: recommendations would be movies we haven't seen yet, sorted by their rating predictions

Why does it make sense?

$$\hat{r}_{ij} = W_i^T U_j$$

- Dot product is just cosine similarity
- We can pretend each of the K latent dimensions is a meaningful, interpretable feature
- E.g. action-adventure, comedy, romance, etc.

$$sim(A, B) = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$



Example:

Example

- 3 dimensions:
  - Action-adventure
  - Comedy
  - Romance
- User vector: w = [1, 0, -1]
  - Likes action-adventure
  - Doesn't care about comedy
  - Dislikes romance
- Movie vector: u = [1, 1, 0]
  - Has action-adventure
  - Has comedy
  - No romance
- $w^T u = +1$

Example

- User vector: w = [1, 0, -1]
  - Likes action-adventure
  - Doesn't care about comedy
  - Dislikes romance
- Movie vector: u = [-1, 1, 1]
  - Definitely no action-adventure
  - Has comedy
  - Lots of romance
- $w^T u = -2$
- Suggests our ratings should be centered around 0

Evaluating the model

- Regression since we want to predict ratings
- We want the squared error only for ratings that are known

$$J = \sum_{i,j \in \Omega} (r_{ij} - \hat{r}_{ij})^2$$

$$\Omega = set\ of\ all\ pairs\ (i,j)\ where\ user\ i\ rated\ movie\ j$$

- Analogous to SVD cost

$$J = \sum_{i=1}^{N} \sum_{j=1}^{D} (X_{ij} - \hat{X}_{ij})^2$$

More Symbols

- Will be useful later

$$\Omega_j = set\ of\ users\ i\ who\ rated\ movie\ j$$

$$\Psi_i = set\ of\ movies\ j\ which\ user\ i\ rated$$

Summary

- Ratings matrix

$$r_{ij} = what\ user\ i\ rated\ movie\ j$$

- Model

$$\hat{r}_{ij} = W_i^T U_j$$

- Cost

$$J = \sum_{i,j \in \Omega} (r_{ij} - \hat{r}_{ij})^2$$

- Useful sets

$$\Omega_j = set\ of\ users\ i\ who\ rated\ movie\ j$$
$$\Psi_i = set\ of\ movies\ j\ which\ user\ i\ rated$$

What do we do now?

- The same thing we always do in machine learning!
- We have a cost, we minimize it wrt parameters
- Bias terms and regularization
- After, we'll see how MF is applied to NLP/GloVe

Solving for W

- Careful about which sets are being summed over
- For J, we want to sum over all ratings
- For a particular user vector w_i, we only care about movies that user rated (because only those ratings involve w_i)

$$J = \sum_{i,j \in \Omega} (r_{ij} - \hat{r}_{ij})^2 = \sum_{i,j \in \Omega} (r_{ij} - w_i^T u_j)^2$$

$$\frac{\partial J}{\partial w_i} = 2 \sum_{j \in \Psi_i} (r_{ij} - w_i^T u_j)(-u_j) = 0$$

Solving for W

- Try to isolate w_i
- It's stuck inside a dot product!

$$\frac{\partial J}{\partial w_i} = 2 \sum_{j \in \Psi_i} (r_{ij} - w_i^T u_j)(-u_j) = 0$$

$$\sum_{j \in \Psi_i} (w_i^T u_j) u_j = \sum_{j \in \Psi_i} r_{ij} u_j$$

Solving for W

- Dot product is commutative

$$\sum_{j \in \Psi_i} (w_i^T u_j) u_j = \sum_{j \in \Psi_i} r_{ij} u_j$$

$$\sum_{j \in \Psi_i} (u_j^T w_i) u_j = \sum_{j \in \Psi_i} r_{ij} u_j$$

Solving for W

$$\sum_{j \in \Psi_i} (u_j^T w_i) u_j = \sum_{j \in \Psi_i} r_{ij} u_j$$

$$scalar \times vector = vector \times scalar$$

$$\sum_{j \in \Psi_i} u_j (u_j^T w_i) = \sum_{j \in \Psi_i} r_{ij} u_j$$

Solving for W

- Drop the brackets

$$\sum_{j \in \Psi_i} u_j (u_j^T w_i) = \sum_{j \in \Psi_i} r_{ij} u_j$$

$$\sum_{j \in \Psi_i} u_j u_j^T w_i = \sum_{j \in \Psi_i} r_{ij} u_j$$

Solving for W

- Summation doesn't actually depend on i
- Add more brackets

$$\sum_{j \in \Psi_i} u_j u_j^T w_i = \sum_{j \in \Psi_i} r_{ij} u_j$$

$$\left(\sum_{j \in \Psi_i} u_j u_j^T \right) w_i = \sum_{j \in \Psi_i} r_{ij} u_j$$

Solving for W

- Now it's just Ax = b, which we know how to solve
- `x = np.linalg.solve(A, b)`

$$\left(\sum_{j \in \Psi_i} u_j u_j^T \right) w_i = \sum_{j \in \Psi_i} r_{ij} u_j$$

$$w_i = \left(\sum_{j \in \Psi_i} u_j u_j^T \right)^{-1} \sum_{j \in \Psi_i} r_{ij} u_j$$

Solving for U

- Try to isolate u_j

$$\frac{\partial J}{\partial u_j} = 2 \sum_{i \in \Omega_j} (r_{ij} - w_i^T u_j)(-w_i) = 0$$

$$\sum_{i \in \Omega_j} (w_i^T u_j) w_i = \sum_{i \in \Omega_j} r_{ij} w_i$$

Solving for U

$$\left(\sum_{i \in \Omega_j} w_i w_i^T\right) u_j = \sum_{i \in \Omega_j} r_{ij} w_i$$

$$u_j = \left(\sum_{i \in \Omega_j} w_i w_i^T\right)^{-1} \sum_{i \in \Omega_j} r_{ij} w_i$$

2-way dependency

- Solution for W depends on U
- Solution for U depends on W

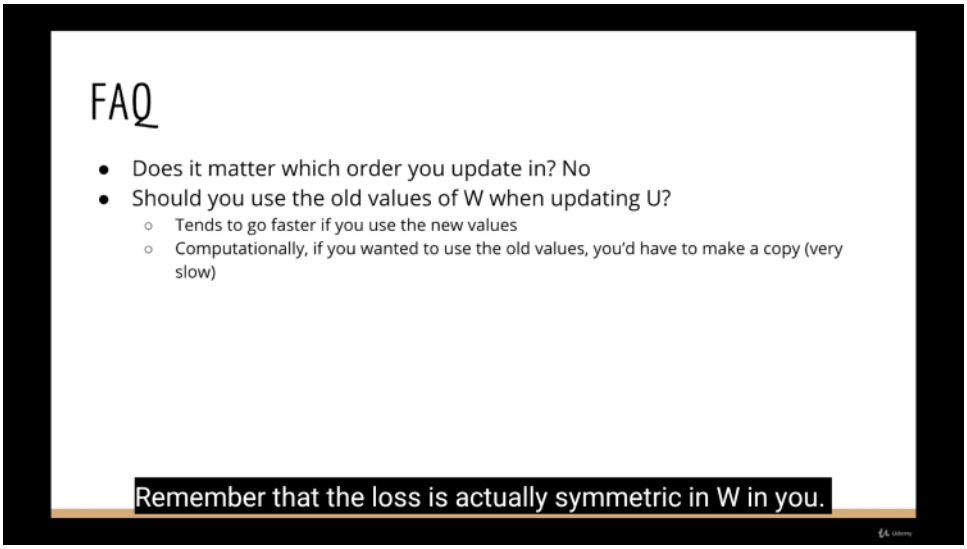
Training algorithm

- Simply apply the equations as is, iteratively
- Is called "alternating least squares"

```
W = randn(N, K); U = randn(M, K);
for t in range(T):
```

$$w_i = \left(\sum_{j \in \Psi_i} u_j u_j^T\right)^{-1} \sum_{j \in \Psi_i} r_{ij} u_j$$

$$u_j = \left(\sum_{i \in \Omega_j} w_i w_i^T\right)^{-1} \sum_{i \in \Omega_j} r_{ij} w_i$$



Hurray, let's see GLOVE !!!

4b. Glove - Theory Basics

Thursday, December 13, 2018 12:34 PM

The GloVe model learns word vectors by examining word co-occurrences within a text corpus.

- Before we train the actual model, we need to construct a co-occurrence matrix X, where a cell Xij is a “strength” which represents how often the word i appears in the context of the word j.

A simple window based co-occurrence matrix:

Example corpus:

- I like deep learning.
- I like NLP.
- I enjoy flying.

counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

We could also construct context distance with a windows size (dynamic context distance)

For example, for a window size of 3:

Context distance
“I love dogs and cats”
X(I, love) += 1
X(I, dogs) += ½
X(I, and) += ⅓

- We run through our corpus just once to build the matrix X, and from then on use this co-occurrence data in place of the actual corpus. We will construct our model based only on the values collected in X.

Long-tail distribution → non-zero values will be very large
So we will take the log
log X(i, j) will be the target
Add 1 before taking the log so we don't have NaNs

Range of X(i,j) will be from zero (no occurrence) to many occurrences.
Taking log X(i,j) will normalize this range.

We will produce embedding for each word wi and wj in the occurrence matrix such that it follows this equation

$$\vec{w}_i^T \vec{w}_j + b_i + b_j = \log X_{ij}$$

- Dot product of wi (1XK) and wj (1XK) = w_i^T * w_j = w_i * w_i^T = (1X1) dimension
- bi and bj are scalar bias terms associated with words i and j, respectively

"**GloVe is essentially a log-bilinear model with a weighted least-squares objective**"

"global log-bilinear regression model which combines the benefits of both **global matrix factorization (Decompose large matrices into low-rank approximations)** and **local context window methods (Learn word representations using adjacent words.)**"

- A bilinear function (see https://en.wikipedia.org/wiki/Bilinear_map) is a function which is linear in each variable when all other variables are fixed. For instance, f(x,y) = x * y.
- **p(y | x) is log-linear if f(x,y) is linear in x and y. It is log-bilinear if f(x,y) is bilinear in x and y**.

  See slide 9 of http://www.cs.utoronto.ca/~hinton/csc2535/notes/hlbl.pdf and slides 17, 23, and 24 of https://piotrmirowski.files.wordpress.com/2014/06/piotrmirowski_2014_wordembeddings.pdf as reference.

From <https://stats.stackexchange.com/questions/157136/log-linear-vs-log-bilinear>

Objective function for the above regression equation:

$$J = \sum_{i=1}^{V} \sum_{j=1}^{V} f(X_{ij}) \left(\vec{w}_i^T \vec{w}_j + b_i + b_j - \log X_{ij} \right)^2$$

-- sum of squared errors weighted with a function f

Function f(Xij) is to prevent **common word pairs** (i.e., those with large Xij values) from skewing our objective too much

$$f(X_{ij}) = \begin{cases} \left(\frac{X_{ij}}{x_{\max}}\right)^{\alpha} & \text{if } X_{ij} < x_{\max} \\ 1 & \text{otherwise.} \end{cases}$$

Two methods of optimizing this cost function:

1. Gradient Descent
2. Alternating Least Squares

After attempting gradient descent on J, the following are the gradients (partial derivatives):

$$\nabla_{\vec{w}_i} J = f(X_{ij})\, \vec{w}_j \odot \left(\vec{w}_i^T \vec{w}_j + b_i + b_j - \log X_{ij}\right)$$

$$\nabla_{\vec{w}_j} J = f(X_{ij})\, \vec{w}_i \odot \left(\vec{w}_i^T \vec{w}_j + b_i + b_j - \log X_{ij}\right)$$

$$\frac{\partial J}{\partial b_i} = f(X_{ij}) \left(\vec{w}_i^T \vec{w}_j + b_i + b_j - \log X_{ij}\right)$$

$$\frac{\partial J}{\partial b_j} = f(X_{ij}) \left(\vec{w}_i^T \vec{w}_j + b_i + b_j - \log X_{ij}\right)$$

Then update the weights of wi, wj and bi and bj

4c. Glove - Pseudocode Implementation

Thursday, January 17, 2019 11:45 AM

0. Preprocess text and Extract Vocabulary from text data
1. Build Co-occurrence matrix "X_{ij}"
2. Compute $f(X_{ij})$,
3. Tune Hyper-parameters: learning_rate=1e-4, reg=0.1, xmax=100, alpha=0.75, epochs=100
   (from the paper:
   The performance of the model depends weakly on the cutoff, which we fix to xmax = 100 for all our experiments. We found that $\alpha = 3/4$ gives a modest improvement over a linear version with $\alpha = 1$.
   Source: https://nlp.stanford.edu/pubs/glove.pdf)
4. Initialize Wi, Wj, bi, bj

Repeat this 'epochs' number of times

5. Compute Cost Function
6. Do backward propagation (computing gradients of the cost function with respect to Wi, Wj, bi, bj)
7. Update Wi, Wj, bi, bj
8. Perform analogies with existing vocabulary to visually observe the performance over many different iterations

5. A comparison of the Word Embedding Implementations

Thursday, December 13, 2018 12:34 PM

Comparison of Word2Vec vs Glove vs FastText

Before that, what is FastText:

- This model is exactly the same as word2vec except instead of training on words, it trains on character n-grams that compose words
- FastText is an extension to Word2Vec proposed by Facebook in 2016.
- Instead of feeding individual words into the Neural Network, FastText breaks words into several n-grams (sub-words). For instance, the tri-grams for the word *apple* is *app, ppl*, and *ple* (ignoring the starting and ending of boundaries of words). The word embedding vector for *apple* will be the sum of all these n-grams.
- After training the Neural Network, we will have word embeddings for all the n-grams given the training dataset.
- Rare words can now be properly represented since it is highly likely that some of their n-grams also appears in other words.

Logic:
It breaks down words into ngrams (e.g. the word "behave" becomes, for ngram size being 2-4 <b, <be,<beh,be, beh, beha, eh, eha, ehav, ha, hav, have, av, ave, ave> - "<" and ">" denote end of words. So for example the ngram "av" is represented as a vector and it learns its context from surrounding ngrams when it appears in behave and in "have", "pavlov" etc.).

Model	Pros	Cons
Word2vec	• The first scalable model that generated word embeddings for large corpus (millions of unique words). • Feed the model raw text and it outputs word vectors.	Word sense is not captured separately. - For example, a word like "cell" that could mean "prison, "biological cell", "phone" etc are all represented in one vector
GloVe	• Training time is lesser than word2vec • There are some benchmarks that show it does better in **semantic relatedness tasks** compared to word2vec	• It has a larger memory footprint need - so maybe an issue for large corpus. (co-occurrence matrix, in addition to Wi and Wj weights matrices) • Also sense separation is absent just as in word2vec
FastText	• Handles **out of vocabulary cases** (which none of the other word embedding models described here can) since it can generate a representation of a vector that is close to the original despite some	• Same as word2vec - multiple sense embeddings not captured.

	ngrams beings wrong (e.g. behaave). This makes it useful in **word sentence similarity** tasks on corpus with misspellings like tweets • Constructing word vectors from summing character ngram vectors, is that word embeddings for **even words that occur rarely in corpus are of better quality** (in word2vec case, since rarely occurring words are not "tugged" often enough, their neighborhoods tend to be of lower quality)	

Source:

- https://www.quora.com/Could-you-please-explain-the-choice-constraints-of-the-pros-cons-while-choosing-Word2Vec-GloVe-or-any-other-thought-vectors-you-have-used
- https://towardsdatascience.com/word-embedding-with-word2vec-and-fasttext-a209c1d3e12c
- https://rare-technologies.com/wordrank-embedding-crowned-is-most-similar-to-king-not-word2vecs-canute/
- https://github.com/parulsethi/gensim/blob/wordrank_wrapper/docs/notebooks/Wordrank_comparisons.ipynb (use brown corpus from nltk - easy to download and compare)
- Dataset: https://github.com/parulsethi/gensim/tree/wordrank_wrapper/docs/notebooks/datasets

Algorithm	Corpus size (No. of tokens)	Train time (sec)	No. of passes for corpus	No. of cores (8GB RAM)	Semantic accuracy	Syntactic accuracy	Total accuracy	Word similarity (SimLex-999)	Word similarity (WS-353)
Word2Vec	1 Million	18	6	4	4.69	2.77	3.0	**Pearson: 0.17 Spearman: 0.15**	0.37 0.37
FastText	1 Million	50	6	4	6.57	**36.95**	**32.9**	Pearson: 0.13 Spearman: 0.11	0.36 0.36

6. (Self-reading) Pre-Trained Vectors Comparison

Thursday, January 03, 2019 1:06 PM

Using pre-trained Word2Vec (gensim) and Glove (spacy) models:
https://www.shanelynn.ie/word-embeddings-in-python-with-spacy-and-gensim/

END. Appendix - Good Links

Wednesday, December 12, 2018 11:30 AM

Glove:

- http://www.foldl.me/2014/glove-python/
- http://nbviewer.jupyter.org/github/hans/glove.py/blob/master/demo/glove.py%20exploration.ipynb
- https://gist.github.com/shagunsodhani/efea5a42d17e0fcf18374df8e3e4b3e8
- Good glove numpy implementation (implemented using adaptive gradient descent method):
  https://github.com/hans/glove.py
  https://github.com/hans/glove.py/blob/master/glove.py
  http://nbviewer.jupyter.org/github/hans/glove.py/blob/master/demo/glove.py%20exploration.ipynb
- Glove implementation using Alternating Least Squares method:
  https://github.com/lazyprogrammer/machine_learning_examples/blob/master/nlp_class2/glove.py
- https://towardsdatascience.com/emnlp-what-is-glove-part-i-3b6ce6a7f970
- https://towardsdatascience.com/emnlp-what-is-glove-part-ii-9e5ad227ee0

Word2Vec:

- https://towardsdatascience.com/an-implementation-guide-to-word2vec-using-numpy-and-google-sheets-13445eebd281 (part 1)
- http://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-negative-sampling/ (part 2)
- https://machinelearningmastery.com/develop-word-embeddings-python-gensim/ (practical)

W2V implementations in Python

- https://nathanrooy.github.io/posts/2018-03-22/word2vec-from-scratch-with-python-and-numpy/
- https://github.com/lazyprogrammer/machine_learning_examples/blob/master/nlp_class2/
- https://github.com/nathanrooy/word2vec-from-scratch-with-python/blob/master/word2vec.py
- https://www.tensorflow.org/tutorials/representation/word2vec
  https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/word2vec/word2vec_basic.py
  https://github.com/tensorflow/models/blob/master/tutorials/embedding/word2vec.py

Comparison:

- https://towardsdatascience.com/word-embedding-with-word2vec-and-fasttext-a209c1d3e12c
- https://rare-technologies.com/wordrank-embedding-crowned-is-most-similar-to-king-not-word2vecs-canute/
- https://github.com/parulsethi/gensim/blob/wordrank_wrapper/docs/notebooks/Wordrank_comparisons.ipynb (use brown corpus from nltk - easy to download and compare)

END. Appendix - Glove efficient numpy Implementation

Thursday, December 13, 2018 6:54 PM

Most important portions of Glove code:
https://github.com/hans/glove.py/blob/master/glove.py

1. build_cooccur accepts a corpus and yields a list of co-occurrence blobs (the XijXij values). It calculates co-occurrences by moving a *sliding n-gram window* over each sentence in the corpus.

   this matrix Xij is built symmetrically: This means that we treat word co-occurrences where the context word is to the left of the main word exactly the same as co-occurrences where the context word is to the right of the main word.

2. train_glove, which prepares the parameters of the model and manages training at a high level

   """
   Train GloVe vectors on the given generator `cooccurrences`, where
   each element is of the form
   (word_i_id, word_j_id, x_ij)
   where `x_ij` is a cooccurrence value X_{ij} as presented in the
   matrix defined by `build_cooccur` and the Pennington et al. (2014)
   paper itself.
   If `iter_callback` is not `None`, the provided function will be
   called after each iteration with the learned `W` matrix so far.
   Keyword arguments are passed on to the iteration step function
   `run_iter`.
   Returns the computed word vector matrix `W`.
   """

3. run_iter, which runs a single parameter update step.
   """
   Run a single iteration of GloVe training using the given
   cooccurrence data and the previously computed weight vectors /
   biases and accompanying gradient histories.
   `data` is a pre-fetched data / weights list where each element is of
   the form

```
    (v_main, v_context,
     b_main, b_context,
     gradsq_W_main, gradsq_W_context,
     gradsq_b_main, gradsq_b_context,
     cooccurrence)
as produced by the `train_glove` function. Each element in this
tuple is an `ndarray` view into the data structure which contains
it.
See the `train_glove` function for information on the shapes of `W`,
`biases`, `gradient_squared`, `gradient_squared_biases` and how they
should be initialized.
The parameters `x_max`, `alpha` define our weighting function when
computing the cost for two word pairs; see the GloVe paper for more
details.
Returns the cost associated with the given weight assignments and
updates the weights by online AdaGrad in place.
"""
```