Machine Learning & Data Mining
Set 5

Caltech CS/CNS/EE 155
February 2023

Policies

- Due 9 PM PST, February 22nd on Gradescope.
- You are free to collaborate on all of the problems, subject to the collaboration policy stated in the syllabus.
- In this course, we will be using Google Colab for code submissions. You will need a Google account.

Submission Instructions

- Submit your report as a single .pdf file to Gradescope (entry code K3RPGE), under "Set 5 Report".
- In the report, **include any images generated by your code** along with your answers to the questions.
- Submit your code by **sharing a link in your report** to your Google Colab notebook for each problem (see naming instructions below). Make sure to set sharing permissions to at least "Anyone with the link can view". **Links that can not be run by TAs will not be counted as turned in.** Check your links in an incognito window before submitting to be sure.
- For instructions specifically pertaining to the Gradescope submission process, see https://www.gradescope.com/get_started#student-submission.

Google Colab Instructions

For each notebook, you need to save a copy to your drive.

1. Open the github preview of the notebook, and click the icon to open the colab preview.
2. On the colab preview, go to File → Save a copy in Drive.
3. Edit your file name to "lastname_firstname_originaltitle", e.g."yue_yisong_3_notebook_part1.ipynb"

1 SVD and PCA [35 Points]

Problem A [3 points]: Let X be a $N \times N$ matrix. For the singular value decomposition (SVD) $X = U\Sigma V^T$, show that the columns of U are the principal components of X. What relationship exists between the singular values of X and the eigenvalues of XX^T?

Solution A:

$$XX^T = U_{PCA}\Lambda U_{PCA}^T$$

where U_{PCA} is orthogonal and Λ is diagonal.

With $X = U_{SVD}\Sigma V^T$ (where U_{SVD} and V are orthogonal and Σ is diagonal),

$$XX^T = \left(U_{SVD}\Sigma V^T\right)\left(U_{SVD}\Sigma V^T\right)^T = U_{SVD}\Sigma V^T V \Sigma U_{SVD}^T = U_{SVD}\Sigma^2 U_{SVD}^T$$

If we also choose to put the Σ values in descending order (like done in PCA), then the SVD of X is unique, which means that $U_{PCA} = U_{SVD}$ and $\Lambda = \Sigma^2$.

Therefore the columns of U_{SVD} are the principal components of X, and the singular values of X are the square-roots of the eigenvalues of XX^T.

Problem B [4 points]: Provide both an intuitive explanation and a mathematical justification for why the eigenvalues of the PCA of X (or rather XX^T) are non-negative. Such matrices are called positive semi-definite and possess many other useful properties.

Solution B:

Mathematical justification: 1A shows that the eigenvalues are the squares of the singular values of X, so as long as X is real-valued they will be non-negative.

Problem C [5 points]: In calculating the Frobenius and trace matrix norms, we claimed that the trace is invariant under cyclic permutations (i.e., Tr(ABC) = Tr(BCA) = Tr(CAB)). Prove that this holds for any number of square matrices.

Hint: First prove that the identity holds for two matrices and then generalize. Recall that $\text{Tr}(AB) = \sum_{i=1}^{N}(AB)_{ii}$. Can you find a way to expand $(AB)_{ii}$ in terms of another sum?

Solution C:

$$\text{Tr}(AB) = \sum_{i=1}^{N}(AB)_{ii} = \sum_{i=1}^{N}\sum_{j=1}^{N} A_{ij}B_{ji} = \sum_{j=1}^{N}\sum_{i=1}^{N} B_{ji}A_{ij} = \sum_{j=1}^{N}(BA)_{jj} = \text{Tr}(BA)$$

Then for any number K of $N \times N$ matrices $X_1, X_2, \ldots$, we can let $A = X_1 X_2 \ldots X_k$, $B = X_{k+1} X_{k+2} \ldots X_K$, so

$$\mathrm{Tr}(X_1 X_2 \ldots X_K) = \mathrm{Tr}(AB) = \mathrm{Tr}(BA) = \mathrm{Tr}(X_{k+1} X_{k+2} \ldots X_K X_1 X_2 \ldots X_k)$$

which gives encompasses all cyclic permutations.

Problem D [3 points]: Outside of learning, the SVD is commonly used for data compression. Instead of storing a full $N \times N$ matrix X with SVD $X = U\Sigma V^T$, we store a truncated SVD consisting of the k largest singular values of Σ and the corresponding columns of U and V. One can prove that the SVD is the best rank-k approximation of X, though we will not do so here. Thus, this approximation can often re-create the matrix well even for low k. Compared to the N^2 values needed to store X, how many values do we need to store a truncated SVD with k singular values? For what values of k is storing the truncated SVD more efficient than storing the whole matrix?

Hint: For the diagonal matrix Σ, do we have to store every entry?

Solution D: *The truncated U and V are both $N \times k$, and the truncated Σ is k non-zero values. In total we have $k(2N+1)$ values to store.*

The truncated SVD is more efficient when $k(2N+1) < N^2$, so when $k < N^2/(2N+1) \approx N/2$.

Dimensions & Orthogonality

In class, we claimed that a matrix X of size $D \times N$ can be decomposed into $U\Sigma V^T$, where U and V are orthogonal and Σ is a diagonal matrix. This is a slight simplification of the truth. In fact, the singular value decomposition gives an orthogonal matrix U of size $D \times D$, an orthogonal matrix V of size $N \times N$, and a rectangular diagonal matrix Σ of size $D \times N$, where Σ only has non-zero values on entries $(\Sigma)_{ii}$, $i \in \{1, \ldots, K\}$, where K is the rank of the matrix X.

Problem E [3 points]: Assume that $D > N$ and that X has rank N. Show that $U\Sigma = U'\Sigma'$, where Σ' is the $N \times N$ matrix consisting of the first N rows of Σ, and U' is the $D \times N$ matrix consisting of the first N columns of U. The representation $U'\Sigma'V^T$ is called the "thin" SVD of X.

Solution E: *Σ only has non-zero entries on row/columns from 1 to N. We can split U and Σ at N rows, so we have*

$$U\Sigma = \begin{pmatrix} U_{1\ldots N} & U_{N+1\ldots D} \end{pmatrix} \begin{pmatrix} \Sigma_{1\ldots N} \\ \Sigma_{N+1\ldots D} \end{pmatrix} = \begin{pmatrix} U' & U_{N+1\ldots D} \end{pmatrix} \begin{pmatrix} \Sigma' \\ 0 \end{pmatrix} = U'\Sigma'$$

Problem F [3 points]: Show that since U' is not square, it cannot be orthogonal according to the definition given in class. Recall that a matrix A is orthogonal if $AA^T = A^TA = I$.

Solution F: *U' has shape $D \times N$, so $U'U'^T$ has shape $D \times D$ and U'^TU' has shape $N \times N$. Therefore $U'U'^T \neq U'^TU'$, so U' cannot be orthogonal.*

Problem G [4 points]: Even though U' is not orthogonal, it still has similar properties. Show that $U'^TU' = I_{N\times N}$. Is it also true that $U'U'^T = I_{D\times D}$? Why or why not? Note that the columns of U' are still orthonormal. Also note that orthonormality implies linear independence.

Solution G:

$$I_{D\times D} = U^TU = \begin{pmatrix} U'^T \\ U^T_{N+1\ldots D} \end{pmatrix} \begin{pmatrix} U' & U_{N+1\ldots D} \end{pmatrix} = \begin{pmatrix} U'^TU' & U'TU_{N+1\ldots D} \\ U_{N+1\ldots D}U' & U^T_{N+1\ldots D}U_{N+1\ldots D} \end{pmatrix}$$

Therefore $U'^TU' = I_{N\times N}$.

$U'U'^T \neq I_{D\times D}$. Since U' has shape $D \times N$, it can only have N linearly independent rows. If $D > N$, there must then be some distinct rows u'_i, u'_j such that $u'_iu'^T_j \neq 0$, since they are not orthonormal. Therefore $U'U'^T$ must at least one non-zero element off the diagonal.

Pseudoinverses

Let X be a matrix of size $D \times N$, where $D > N$, with "thin" SVD $X = U\Sigma V^T$. Assume that X has rank N.

Problem H [4 points]: Assuming that Σ is invertible, show that the pseudoinverse $X^+ = V\Sigma^+ U^T$ as given in class is equivalent to $V\Sigma^{-1}U^T$. Refer to lecture 10 (slide 53) for the definition of pseudoinverse.

Solution H: $\Sigma^+ = \begin{bmatrix} \sigma_1^+ & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_N^+ \end{bmatrix}$ *where* $\sigma^+ = 1/\sigma$ *if* $\sigma > 0$ *as given in class, so*

$$\Sigma\Sigma^+ = \begin{bmatrix} \sigma_1 & \dots & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_N \end{bmatrix} \begin{bmatrix} \sigma_1^+ & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_N^+ \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & 1 \end{bmatrix} = I$$

Therefore $\Sigma^+ = \Sigma^{-1}$, *so* $X^+ = V\Sigma^{-1}U^T$.

Problem I [4 points]: Another expression for the pseudoinverse is the least squares solution $X^{+'} = (X^TX)^{-1}X^T$. Show that (again assuming Σ invertible) this is equivalent to $V\Sigma^{-1}U^T$.

Solution I: $X = U\Sigma V^T$, *so*

$$\begin{aligned} (X^TX)^{-1}X^T &= \left((U\Sigma V^T)^T U\Sigma V^T\right)^{-1}(U\Sigma V^T)^T \\ &= \left(V\Sigma U^T U\Sigma V^T\right)^{-1}(U\Sigma V^T)^T \\ &= \left(V\Sigma^2 V^T\right)^{-1} V\Sigma U^T \\ &= (V^T)^{-1}\Sigma^{-2}V^{-1}V\Sigma U^T \\ &= V\Sigma^{-2}\Sigma U^T \\ &= V\Sigma^{-1}U^T \end{aligned}$$

Problem J [2 points]: One of the two expressions in problems H and I for calculating the pseudoinverse is highly prone to numerical errors. Which one is it, and why? Justify your answer using condition numbers.

Solution J: *I is probably more prone to numerical errors. I requires us to invert* X^TX *whereas H requires inverting* Σ. *Since* Σ^2 *gives the singular values of* X^TX *(1A), the condition number for* X^TX *will be the square of the condition number for* Σ.

2 Matrix Factorization [30 Points]

In the setting of collaborative filtering, we derive the coefficients of the matrices $U \in \mathbb{R}^{M \times K}$ and $V \in \mathbb{R}^{N \times K}$ by minimizing the regularized square error:

$$\arg\min_{U,V} \frac{\lambda}{2}\left(\|U\|_F^2 + \|V\|_F^2\right) + \frac{1}{2}\sum_{i,j}\left(y_{ij} - u_i^T v_j\right)^2$$

where u_i^T and v_j^T are the i^{th} and j^{th} rows of U and V, respectively, and $\|\cdot\|_F$ represents the Frobenius norm. Then $Y \in \mathbb{R}^{M \times N} \approx UV^T$, and the *ij*-th element of Y is $y_{ij} \approx u_i^T v_j$.

Problem A [5 points]: Derive the gradients of the above regularized squared error with respect to u_i and v_j, denoted ∂_{u_i} and ∂_{v_j} respectively. We can use these to compute U and V by stochastic gradient descent using the usual update rule:

$$u_i = u_i - \eta \partial_{u_i}$$
$$v_j = v_j - \eta \partial_{v_j}$$

where η is the learning rate.

Solution A: *I use x_{i*} to indicate the i-th row of X, and x_{*i} to indicate the i-th column of X.*

$$\begin{aligned}
\partial_{u_i} &= \frac{\lambda}{2}\frac{\partial}{\partial u_i}\|U\|_F^2 + \frac{1}{2}\frac{\partial}{\partial u_i}\|Y - UV^T\|_F^2 \\
&= \frac{\lambda}{2}\frac{\partial}{\partial u_i} u_{i*}u_{i*}^T + \frac{1}{2}\frac{\partial}{\partial u_i}(y_{i*} - u_{i*}V^T)(y_{i*} - u_{i*}V^T)^T \\
&= \frac{\lambda}{2}\frac{\partial}{\partial u_i} u_{i*}u_{i*}^T + \frac{1}{2}\frac{\partial}{\partial u_i}(y_{i*}y_{i*}^T - y_{i*}Vu_{i*}^T - u_{i*}V^T y_{i*}^T + u_{i*}V^T V u_{i*}^T) \\
&= \frac{\lambda}{2}\frac{\partial}{\partial u_i} u_{i*}u_{i*}^T + \frac{1}{2}\frac{\partial}{\partial u_i}(-2y_{i*}Vu_{i*}^T + u_{i*}V^T V u_{i*}^T) \\
&= \lambda u_{i*} - y_{i*}V + u_{i*}V^T V
\end{aligned}$$

$$\begin{aligned}
\partial_{v_j} &= \frac{\lambda}{2}\frac{\partial}{\partial v_j}\|V\|_F^2 + \frac{1}{2}\frac{\partial}{\partial v_j}\|Y - UV^T\|_F^2 \\
&= \frac{\lambda}{2}\frac{\partial}{\partial v_j} v_{j*}v_{j*}^T + \frac{1}{2}\frac{\partial}{\partial v_j}(y_{*j} - Uv_{j*}^T)^T(y_{*j} - Uv_{j*}^T) \\
&= \frac{\lambda}{2}\frac{\partial}{\partial v_j} v_{j*}v_{j*}^T + \frac{1}{2}\frac{\partial}{\partial v_j}(y_{*j}^T y_{*j} - y_{*j}^T U v_{j*}^T - v_{j*}U^T y_{*j} + v_{j*}U^T U v_{j*}^T) \\
&= \frac{\lambda}{2}\frac{\partial}{\partial v_j} v_{j*}v_{j*}^T + \frac{1}{2}\frac{\partial}{\partial v_j}(-2y_{*j}^T U v_{j*}^T + v_{j*}U^T U v_{j*}^T) \\
&= \lambda v_{j*} - y_{*j}^T U + v_{j*}U^T U
\end{aligned}$$

Problem B [5 points]: Another method to minimize the regularized squared error is alternating least squares (ALS). ALS solves the problem by first fixing U and solving for the optimal V, then fixing this new V and solving for the optimal U. This process is repeated until convergence.

Derive closed form expressions for the optimal u_i and v_j. That is, give an expression for the u_i that minimizes the above regularized square error given fixed V, and an expression for the v_j that minimizes it given fixed U.

Solution B: *We set the gradient to 0 to find the optimal u_i or v_j.*

$$\begin{aligned}
\lambda u_{i*} - y_{i*}V + u_{i*}V^TV &= 0 \\
(\lambda I + V^TV)u_{i*} &= y_{i*}V \\
u_{i*} &= (\lambda I + V^TV)^{-1}y_{i*}V
\end{aligned}$$

$$\begin{aligned}
\lambda v_{j*} - y_{*j}^TU + v_{j*}U^TU &= 0 \\
(\lambda I + V^TV)v_{j*} &= y_{*j}^TU \\
v_{j*} &= (\lambda I + U^TU)^{-1}y_{*j}^TU
\end{aligned}$$

Problem C [10 points]: Download the provided MovieLens dataset (train.txt and test.txt). The format of the data is (*user, movie, rating*), where each triple encodes the rating that a particular user gave to a particular movie. Make sure you check if the user and movie ids are 0 or 1-indexed, as you should with any real-world dataset.

Implement matrix factorization with stochastic gradient descent for the MovieLens dataset, using your answer from part A. Assume your input data is in the form of three vectors: a vector of is, js, and y_{ij}s. Set $\lambda = 0$ (in other words, do not regularize), and structure your code so that you can vary the number of latent factors (k). You may use the Python code template in 2_notebook.ipynb; to complete this problem, your task is to fill in the four functions in 2_notebook.ipynb marked with TODOs.

In your implementation, you should:

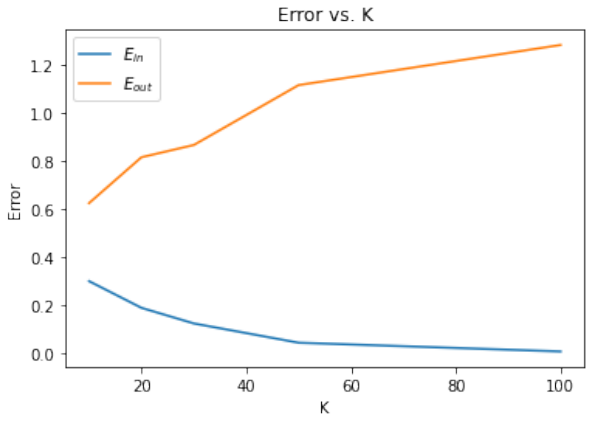
- Initialize the entries of U and V to be small random numbers; set them to uniform random variables in the interval $[-0.5, 0.5]$.
- Use a learning rate of 0.03.
- Randomly shuffle the training data indices before each SGD epoch.

- Set the maximum number of epochs to 300, and terminate the SGD process early via the following early stopping condition:
  - Keep track of the loss reduction on the training set from epoch to epoch, and stop when the relative loss reduction compared to the first epoch is less than $\epsilon = 0.0001$. That is, if $\Delta_{0,1}$ denotes the loss reduction from the initial model to end of the first epoch, and $\Delta_{i,i-1}$ is defined analogously, then stop after epoch t if $\Delta_{t-1,t}/\Delta_{0,1} \leq \epsilon$.

Solution C: *Colab notebook*

Problem D [5 points]: Use your code from the previous problem to train your model using $k = 10, 20, 30, 50, 100$, and plot your E_{in}, E_{out} against k. Note that E_{in} and E_{out} are calculated via the squared loss, i.e. via $\frac{1}{2}\sum_{i,j}\left(y_{ij} - u_i^T v_j\right)^2$. What trends do you notice in the plot? Can you explain them?

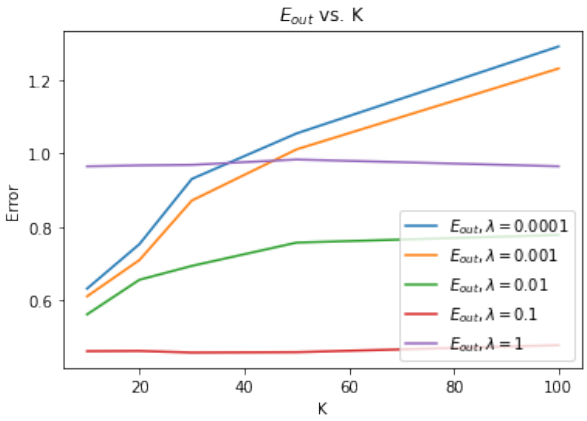
Solution D:



As K increases, in sample error decreases while out of sample error increases. This is expected because increasing K increases the capacity of the model, so it overfits on the training data.

Problem E [5 points]: Now, repeat problem D, but this time with the regularization term. Use the following regularization values: $\lambda \in \{1e-4, 1e-3, 0.01, 0.1, 1\}$. For each regularization value, use the same range of values for k as you did in the previous part. What trends do you notice in the graph? Can you explain them in the context of your plots for the previous part? You should use your code you wrote for part C in 2_notebook.ipynb.

Solution E:



For low λ, we see the same behavior as before: out of sample error increases with a larger model due to overfitting. As λ increases, the overfitting decreases to a minimum around $\lambda = 0.1$. Then at $\lambda = 1$, error increases again because the regularization is too high, so we are underfitting the data.

3 Word2Vec Principles [35 Points]

The Skip–gram model is part of a family of techniques that try to understand language by looking at what words tend to appear near what other words. The idea is that semantically similar words occur in similar contexts. This is called "distributional semantics", or "you shall know a word by the company it keeps".

The Skip–gram model does this by defining a conditional probability distribution $p(w_O|w_I)$ that gives the probability that, given that we are looking at some word w_I in a line of text, we will see the word w_O nearby. To encode p, the Skip-gram model represents each word in our vocabulary as two vectors in $\mathbb{R}^D$: one vector for when the word is playing the role of w_I ("input"), and one for when it is playing the role of w_O ("output"). (The reason for the 2 vectors is to help training — in the end, mostly we'll only care about the w_I vectors.) Given these vector representations, p is then computed via the familiar softmax function:

$$p(w_O|w_I) = \frac{\exp\left({v'}_{w_O}^T v_{w_I}\right)}{\sum_{w=1}^{W} \exp\left({v'}_w^T v_{w_I}\right)} \tag{2}$$

where v_w and v'_w are the "input" and "output" vector representations of word a $w \in \{1, ..., W\}$. (We assume all words are encoded as positive integers.)

Given a sequence of training words $w_1, w_2, \ldots, w_T$, the training objective of the Skip-gram model is to maximize the average log probability

$$\frac{1}{T}\sum_{t=1}^{T} \sum_{-s\leq j\leq s, j\neq 0} \log p(w_{t+j}|w_t) \tag{1}$$

where s is the size of the "training context" or "window" around each word. Larger s results in more training examples and higher accuracy, at the expense of training time.

Problem A [5 points]: If we wanted to train this model with naive gradient descent, we'd need to compute all the gradients $\nabla \log p(w_O|w_I)$ for each w_O, w_I pair. How does computing these gradients scale with W, the number of words in the vocabulary, and D, the dimension of the embedding space? To be specific, what is the time complexity of calculating $\nabla \log p(w_O|w_I)$ for a single w_O, w_I pair?

Solution A: *For a single w_O, w_I pair, we need to compute one partial derivative for each of the weights (linear with D). For w_O weights, there is only the numerator term so the time complexity is constant. For w_I weights, there is one numerator term and W denominator terms, so the time complexity is linear with W. Therefore the time complexity for fully computing a single pair is $O(WD)$.*

Since we have W^2 pairs of words, the overall time complexity is $O(W^3D)$.

Problem B [10 points]: When the number of words in the vocabulary W is large, computing the regular softmax can be computationally expensive (note the normalization constant on the bottom of Eq. 2). For reference, the standard fastText pre-trained word vectors encode approximately $W \approx 218000$ words in

Table 1: Words and frequencies for Problem B

Word	Occurrences
do	18
you	4
know	7
the	20
way	9
of	4
devil	5
queen	6

$D = 100$ latent dimensions. One trick to get around this is to instead represent the words in a binary tree format and compute the hierarchical softmax.

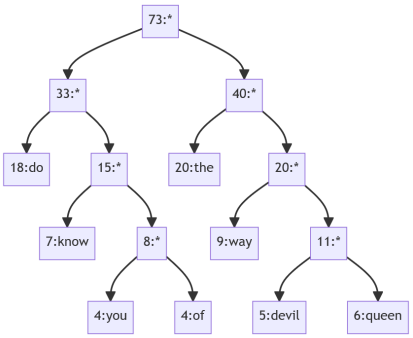
When the words have all the same frequency, then any balanced binary tree will minimize the average representation length and maximize computational efficiency of the hierarchical softmax. But in practice, words occur with very different frequencies — words like "a", "the", and "in" will occur many more times than words like "representation" or "normalization".

The original paper (Mikolov et al. 2013) uses a Huffman tree instead of a balanced binary tree to leverage this fact. For the 8 words and their frequencies listed in the table below, build a Huffman tree using the algorithm found here. Then, build a balanced binary tree of depth 3 to store these words. Make sure that each word is stored as a *leaf node* in the trees.

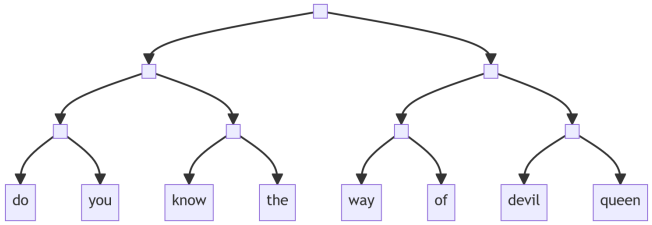
The representation length of a word is then the length of the path (the number of edges) from the root to the leaf node corresponding to the word. For each tree you constructed, compute the expected representation length (averaged over the actual frequencies of the words).

Solution B:

Huffman tree:



Binary tree:



Expected length with Huffman tree: $(2(18+20)+3(7+9)+4(4+4+5+6))/73 \approx 2.7$.

Expected length with binary tree: 3

Problem C [3 points]: In principle, one could use any D for the dimension of the embedding space. What do you expect to happen to the value of the training objective as D increases? Why do you think one might not want to use very large D?

Solution C: *As D increases, the training objective should improve, since the model will be able to learn more about each word. But very large D would allow the model to make lots of word vectors orthogonal to each other, which would improve its training performance but reduce our ability to find word similarity by comparing their vectors, since the dot product of orthogonal vectors is 0.*

Implementing Word2Vec

Word2Vec is an efficient implementation of the Skip–gram model using neural network–inspired training techniques. We'll now implement Word2Vec on text datasets using Keras. This blog post provides an overview of the particular Word2Vec implementation we'll use.

At a high level, we'll do the following:

(i) Load in a list L of the words in a text file

(ii) Given a window size s, generate up to $2s$ training points for word L_i. The diagram below shows an example of training point generation for $s = 2$:

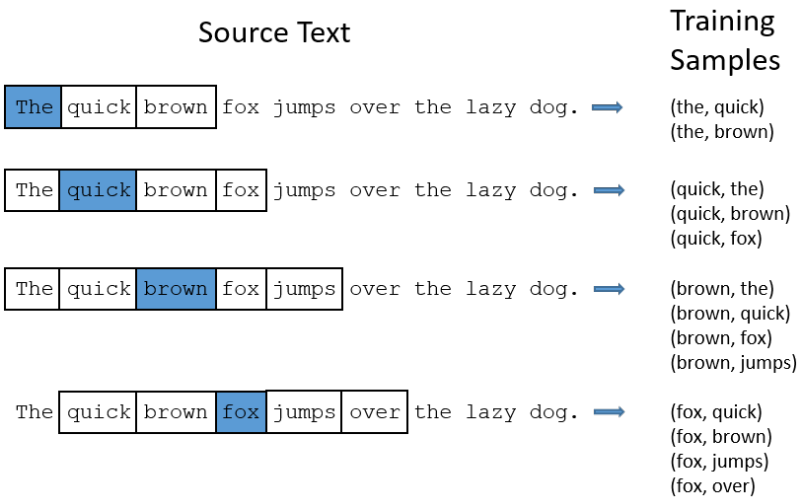


Figure 1: Generating Word2Vec Training Points

(iii) Fit a neural network consisting of a single hidden layer of 10 units on our training data. The hidden layer should have no activation function, the output layer should have a softmax activation, and the loss function should be the cross entropy function.

Notice that this is exactly equivalent to the Skip–gram formulation given above where the embedding dimension is 10: the columns (or rows, depending on your convention) of the input–to–hidden weight matrix in our network are the w_I vectors, and those of the hidden–to–output weight matrix are the w_O vectors.

(iv) Discard our output layer and use the matrix of weights between our input layer and hidden layer as the matrix of feature representations of our words.

(v) Compute the cosine similarity between each pair of distinct words and determine the top 30 pairs of most-similar words.

Implementation

See 3_notebook.ipynb, which implements most of the above.

Problem D [10 points]: Fill out the TODOs in the skeleton code; specifically, add code where indicated to train a neural network as described in (iii) above and extract the weight matrix of its input–to–hidden weight matrix. Also, fill out the generate_traindata() function, which generates our data and label matrices.

Solution D: *See solution code in 3_notebook.ipynb*

Colab notebook

Running the code

Run your model on dr_seuss.txt and answer the following questions:

Problem E [2 points]: What is the dimension of the weight matrix of your hidden layer?

Solution E: 308×10

Problem F [2 points]: What is the dimension of the weight matrix of your output layer?

Solution F: 10×308

Problem G [1 points]: List the top 30 pairs of most similar words that your model generates.

Solution G:

```
Pair(town, found), Similarity: 0.9917908
Pair(found, town), Similarity: 0.9917908
Pair(hand, wave), Similarity: 0.9860244
Pair(wave, hand), Similarity: 0.9860244
Pair(likes, drink), Similarity: 0.9838316
Pair(drink, likes), Similarity: 0.9838316
Pair(comes, put), Similarity: 0.9741637
Pair(put, comes), Similarity: 0.9741637
Pair(goat, boat), Similarity: 0.97260076
Pair(boat, goat), Similarity: 0.97260076
Pair(play, game), Similarity: 0.97163326
Pair(game, play), Similarity: 0.97163326
Pair(long, way), Similarity: 0.9714244
Pair(way, long), Similarity: 0.9714244
Pair(is, shoe), Similarity: 0.96591914
Pair(shoe, is), Similarity: 0.96591914
Pair(cant, but), Similarity: 0.9646215
Pair(but, cant), Similarity: 0.9646215
Pair(zeep, today), Similarity: 0.964261
```

```
Pair(today, zeep), Similarity: 0.964261
Pair(fly, kite), Similarity: 0.96381795
Pair(kite, fly), Similarity: 0.96381795
Pair(hair, heads), Similarity: 0.96340865
Pair(heads, hair), Similarity: 0.96340865
Pair(finger, top), Similarity: 0.96332663
Pair(top, finger), Similarity: 0.96332663
Pair(did, ever), Similarity: 0.9629352
Pair(ever, did), Similarity: 0.9629352
Pair(wire, goodbye), Similarity: 0.9616629
Pair(goodbye, wire), Similarity: 0.9616629
```

Problem H [2 points]: What patterns do you notice across the resulting pairs of words?

Solution H: *Mostly words that intuitively would be near each other: (hand, wave), (likes, drink), (game, play), (long, way), (fly, kite), etc.*

Some rhymes: (town, found), (boat, goat)