Policies

- Due 9 PM, February 16th, via Moodle.
- You are free to collaborate on all of the problems, subject to the collaboration policy stated in the syllabus.
- You should submit all code used in the homework. We ask that you use Python 3 and sklearn version 0.19 for your code, and that you comment your code such that the TAs can follow along and run it without any issues.

Submission Instructions

Please submit your assignment as a .zip archive with filename `LastnameFirstname.zip` (replacing `Lastname` with your last name and `Firstname` with your first name), containing a PDF of your assignment writeup **in the main directory** with filename `LastnameFirstname_Set5.pdf` and your code files **in a directory named `LastnameFirstname`**. Failure to do so will result in a **2 point deduction**. Submit your code as Jupyter notebook .ipynb files or .py files, and **include any images generated by your code along with your answers in the solution .pdf file.**

1 SVD and PCA [35 Points]

Relevant materials: Lectures 10, 11

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: *The PCA of X is*

$$XX^T = U\Sigma V^T \left(U\Sigma V^T\right)^T = U\Sigma^T V^T V \Sigma U^T = U\Sigma^T \Sigma U^T = U\Sigma^2 U^T$$

The columns of U are the eigenvectors of the PCA XX^T, and thus U contains the principal components of X. Furthermore, the singular values of X are the square roots of the eigenvalues of XX^T (or the eigenvalues of XX^T are the squared singular values of X).

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: *The eigenvalues of the PCA of X measure the total variation along their corresponding eigenvectors; this is a non-negative quantity. From Problem A, we know that the eigenvalues of A are the squares of the singular values of X and therefore must 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: *We can write (for square matrices of size $N \times N$)*

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

To generalize to M matrices (say, $A_1, \ldots, A_M$), it suffices to define $B = A_2A_3 \ldots A_M$ and note

$$\mathit{Tr}(A_1A_2 \ldots A_M) = \mathit{Tr}(A_1B) = \mathit{Tr}(BA_1) = \mathit{Tr}(A_2 \ldots A_M A_1)$$

Repeating this procedure yields any permutation.

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: *For Σ, we need only store the k singular values themselves. Each of these values corresponds to a column of size N from U and a row of size N from V^T, for a total of $2Nk$ values. Thus, our truncated SVD requires us to store $(2N+1)k$ values. This is less than N^2 if*

$$k < \frac{N^2}{2N+1} = O(N)$$

Problem E [10 points]: 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 Y 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.

i. [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.i: *Since Σ only has non-zero entries on the elements $(\Sigma)_{ii}$, every row beneath the first N only contains zeroes. These $D-N$ rows cancel out the $D-N$ rightmost columns of U when we multiply, and thus we can neglect them.*

ii. [3 points]: Show that since U' is not square, it cannot be orthogonal according to the defintion given in class.

Solution E.ii: *An orthogonal matrix A has the property that $AA^T = A^TA = I$. But, $U'U'^T$ is a matrix of size $D \times D$, whereas U'^TU' is a matrix of size $N \times N$. These matrices aren't even the same size, and therefore clearly* *cannot be equal.*

iii. [4 points]: Even though U' is not orthogonal, it still has similar properties. Show that $U'^T U' = I_{N \times N}$. Is it also true that $U' U'^T = I_{D \times D}$? Note that the columns of U' are still orthonormal.

Solution E.iii: *The first statement is true because the ij-th element of $(U'^T U')$ is $[U^T U]_{ij} = u_i^T u_j = \mathbb{1}(i = j)$, where u_i is the i-th (orthonormal) column of U' and $\mathbb{1}(\cdot)$ denotes the indicator function.*

The second statement is false. For it to be true, the rows of U' would have to be orthonormal (and therefore linearly independent). But, we have D rows of N-dimensional vectors, which cannot be independent (and therefore cannot be orthonormal).

Problem F [10 points]: 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.

i. [4 points]: Assuming that Σ is invertible, show that the pseudoinverse $X^+ = V \Sigma^+ U^T$ as given in class is equivalent to $V \Sigma^{-1} U^T$.

Solution F.i: *Since Σ is a diagonal matrix, it is invertible if and only if all of its diagonal elements are non-zero (to see why, note that setting any of the diagonal elements to zero lowers the rank). The inverse of Σ is then the diagonal matrix with $(\Sigma^{-1})_{ii} = 1/\sigma_i$. As defined in class $(\Sigma^+)_{ii}$ contains either $1/\sigma_i$ (if $\sigma_i \neq 0$) or 0 (if $\sigma_i = 0$). Since all singular values are non-zero, $(\Sigma^+)_{ii} = 1/\sigma_i$ for all i, and therefore $\Sigma^+ = \Sigma^{-1}$.*

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

Solution F.ii: *Using $X = U \Sigma V^T$ and the fact that Σ is invertible, we find*

$$X^{+'} = (X^T X)^{-1} X^T = ((U\Sigma V^T)^T U \Sigma V^T)^{-1} (U \Sigma V^T)^T = (V \Sigma U^T U \Sigma V^T)^{-1} V \Sigma U^T$$

$$= (V \Sigma^2 V^T)^{-1} V \Sigma U^T = (V \Sigma^{-2} V^T) V \Sigma U^T = V \Sigma^{-1} U^T$$

Thus, $X^+ = X^{+'}$. Note that we have used $U^T U = I$ from the previous problem.

iii. [2 points]: One of the two expressions in parts i and ii for calculating the psuedoinverse is highly prone to numerical errors; which one is it, and why?

Hint: While not required, a helpful way to quantify the numerical instability is to compare the condition numbers of Σ and $X^T X$.

Solution F: *The least squares solution is prone to numerical errors as a result of the $(X^TX)^{-1}$ inverse, which can be difficult to compute for certain matrices. The condition number is a measure of this difficulty, and can be defined $\kappa(X) = \sigma_1/\sigma_D$. Since $X^TX = V\Sigma^2V^T$, its singular values are the squares of the singular values of X and therefore $\kappa(X^TX) = (\kappa(X))^2$. In constrast, calculating $\Sigma^+ = \Sigma^{-1}$ has condition number $\kappa(\Sigma) = \kappa(X)$, which can be much lower. Note that Σ is the only matrix we have to invert, since we can just transpose U and V^T.*

2 Matrix Factorization [30 Points]

Relevant materials: Lecture 11

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. 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: $\partial_{u_i} = \lambda u_i - \sum_{j=1}^{N}(y_{ij} - u_i^T v_j)v_j$
$\partial_{v_j} = \lambda v_j - \sum_{i=1}^{N}(y_{ij} - u_i^T v_j)u_i$

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: *The following are obtained by setting the partial derivative expressions ∂_{u_i} and ∂_{v_j} to zero and solving for u_i and v_j respectively:*

$u_i = \left(\lambda I_K + \sum_j v_j v_j^T\right)^{-1}\left(\sum_j y_{ij} v_j\right)$
$v_j = \left(\lambda I_K + \sum_i u_i u_i^T\right)^{-1}\left(\sum_i y_{ij} u_i\right)$

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.

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 the files 2D.py and prob2utils_skeleton.py; to complete this problem, your task is to fill in the four functions in prob2utils_skeleton.py.

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: *See 2D.py and prob2utils.py for the solution code.*

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: *See 2D.py and prob2utils.py for the solution code used to generate the plot below.*

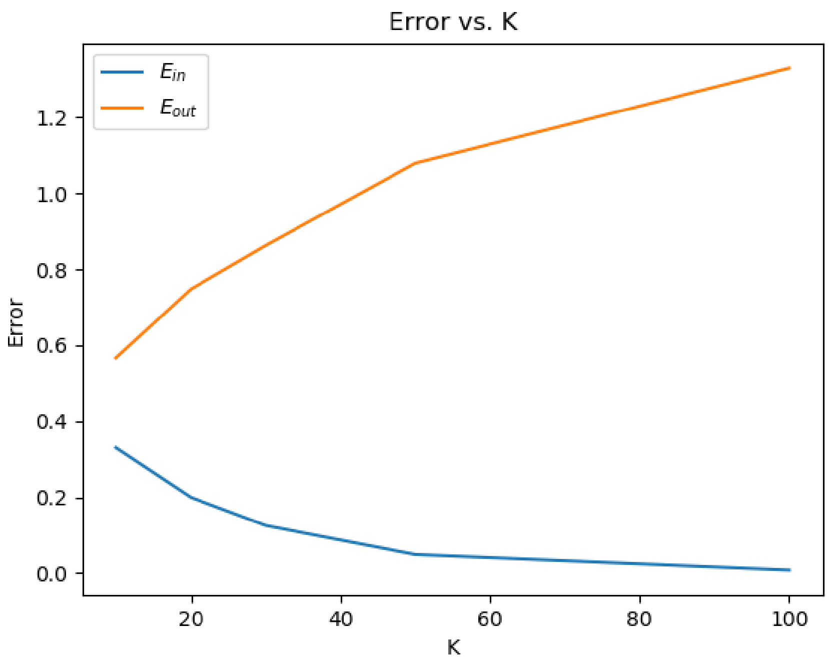

Figure 1: Unregularized factorization

We notice that E_{in} decreases monotonically as k increases. This makes sense, as we are able to capture more information with more latent factors. E_{out} increases monotonically with k, which indicates that we are overfitting to 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 may use your code you wrote for part C, as well as the file 2E.py.

Solution E: *See 2E.py and prob2utils.py for the solution code used to generate the plots below.*

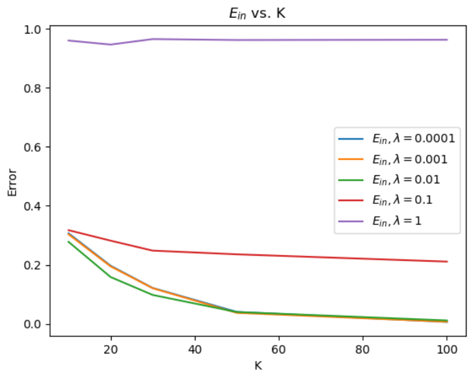

Figure 2: E_{in} vs k for different λ

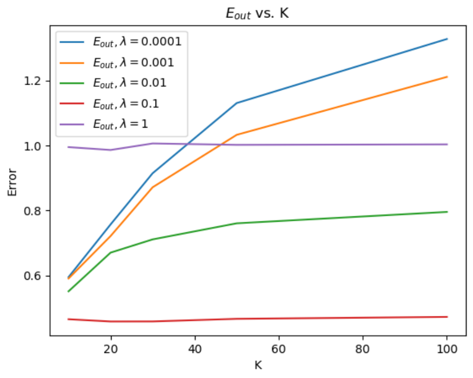

Figure 3: E_{out} vs k for different λ

E_{in} continues to decrease with k across all values of λ, while E_{out} increases with k for most values of λ. $\lambda = 0.1$ results in an E_{out} below 0.5, which is better than the best E_{out} achieved without regularization (0.57, achieved when $k = 10$).

3 Word2Vec Principles [35 Points]

Relevant materials: Lecture 12

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?

Solution A: $\log p(w_O|w_I) = {v'}_{w_O}^{T} v_{w_I} - \log \sum_{w=1}^{W} \exp\left({v'_w}^{T} v_{w_I}\right)$

Thus we have $\frac{\partial \log p(w_O|w_I)}{\partial v'_{w_O}} = v_{w_I} - \frac{v_{w_I} \exp({v'_{w_O}}^{T} v_{w_I})}{\sum_{w=1}^{W} \exp\left({v'_w}^{T} v_{w_I}\right)}$

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

Each dot product between two d-dimensional feature vectors takes time $O(D)$, so the time complexity of computing $\nabla \log p(w_O|w_I)$ is $O(WD)$.

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

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

reference, the standard fastText pre-trained word vectors encode approximately $W \approx 218000$ words in $D = 100$ 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 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: *For the balanced binary tree, the representation length for all the words is simply the depth of the tree: 3. Hence,* $\langle L \rangle_{binary} = 3$.

For the Huffman tree, one follows the algorithm and obtains a tree with the following representation lengths: $[2, 4, 3, 2, 3, 4, 4, 4]$ *for the words [do, you, know, the, way, of, devil, queen].*

Weighting each of the representation lengths with the corresponding frequency of occurrence given in the table results in $\langle L \rangle_{Huffman} \approx 2.739$, *an improvement of 8% over the balanced binary tree representation length. Larger improvements may be seen over greater number of words and real-life distributions of word frequencies.*

Note that this improvement is based on the assumed distribution of words as seen in the training set — if the test set words are distributed sufficiently differently, then the Huffman-based hierarchical softmax may perform worse than a naive balanced binary tree.

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: *The training objective will increase as D increases because the model has higher capacity and can fit the training data better. We don't want too large a D because that will overfit to a finite training set.*

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 $s = 2$:

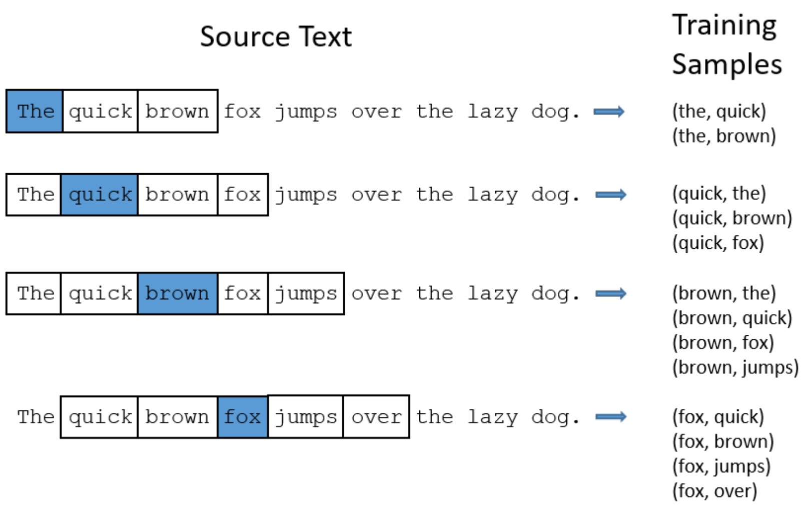

Figure 4: Generating Word2Vec Training Points

(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.

Your Task

Download the helper functions (P3CHelpers.py) and skeleton code (P3CSkeleton.py) from the course website, which implement 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 P3C.py*

Problem E [7 points]: Run your model on dr_seuss.txt.

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

Solution E.i: *We obtain a matrix of shape* $(308, 10)$

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

Solution E.ii: *We obtain a matrix of shape* $(10, 308)$

iii. [3 points]: List the top 30 pairs of most similar words that your model generates. What patterns do you notice across the resulting pairs of words?

Solution E.iii:

One run of the solution code results in the following list:

```
Pair(foot, shoe), Similarity: 0.984582
Pair(shoe, foot), Similarity: 0.984582
Pair(heads, upon), Similarity: 0.969106
Pair(upon, heads), Similarity: 0.969106
Pair(cold, off), Similarity: 0.963113
Pair(off, cold), Similarity: 0.963113
Pair(did, milk), Similarity: 0.962288
Pair(milk, did), Similarity: 0.962288
Pair(dont, girls), Similarity: 0.959069
Pair(girls, dont), Similarity: 0.959069
Pair(top, left), Similarity: 0.953689
Pair(left, top), Similarity: 0.953689
Pair(glad, sad), Similarity: 0.949372
Pair(sad, glad), Similarity: 0.949372
Pair(drink, wink), Similarity: 0.948035
Pair(wink, drink), Similarity: 0.948035
Pair(finger, top), Similarity: 0.947593
Pair(cannot, hear), Similarity: 0.946927
Pair(hear, cannot), Similarity: 0.946927
Pair(fox, goat), Similarity: 0.944915
Pair(goat, fox), Similarity: 0.944915
Pair(thin, sad), Similarity: 0.944636
Pair(wave, ish), Similarity: 0.943365
Pair(ish, wave), Similarity: 0.943365
Pair(where, more), Similarity: 0.942425
Pair(more, where), Similarity: 0.942425
Pair(dark, rain), Similarity: 0.936788
Pair(rain, dark), Similarity: 0.936788
Pair(slow, thin), Similarity: 0.933333
Pair(gox, socks), Similarity: 0.927345
```

Words that occur infrequently but next to each other have high similarities. Also, if word b is the word most similar to word a, it's typically also the case that word a is the word most similar to word b. Words found to be similar often rhyme; this is likely because rhyming words are often located near each other in Dr. Seuss's writing.