Machine Learning & Data Mining
Set 5

Caltech CS/CNS/EE 155
February 2022

---

## Policies

- Due 9 PM PST, February $16^{\text{th}}$ on Gradescope.
- You are free to collaborate on all of the problems, subject to the collaboration policy stated in the syllabus.
- If you have trouble with this homework, it may be an indication that you should drop the class.
- 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 7426YK), 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 $\rightarrow$ 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]

*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:**

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

*where $\Sigma^2 = \Lambda$, and using the fact that $V^T V = I$ because they are orthogonal. Thus, each column of $U$ is an eigenvector of $XX^T$, which are defined as the principal components of X. At the same time, the singular values of $X$ are 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:** *Intuitive explanation: the eigenvalues of the PCA quantify the the variance captured by projecting onto a given eigenvector, and variance should be positive. Mathematical explanation: the eigenvalues are the squares of the singular values, so they must be positive.*

**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 Tr$(AB) = \sum_{i=1}^{N} (AB)_{ii}$. Can you find a way to expand $(AB)_{ii}$ in terms of another sum?

**Solution C:**

$$Tr(AB) = \sum_{i=1}^{N} (AB)_{ii} = \sum_{i=1}^{N} \sum_{j=1}^{N} A_{ij} B_{ji} \tag{2}$$

$$Tr(BA) = \sum_{i=1}^{N} (BA)_{ii} = \sum_{i=1}^{N} \sum_{j=1}^{N} B_{ij} A_{ji} = \sum_{i=1}^{N} \sum_{j=1}^{N} A_{ji} B_{ij} = \sum_{j=1}^{N} \sum_{i=1}^{N} A_{ji} B_{ij} \tag{3}$$

*The expressions at the end of these two equations are the same, with indices $i$ and $j$ flipped. Thus Tr(AB) = Tr(BA).*
*This does indeed generalize to cyclic permutations if we group two of the matrices at a time. $Tr(A[BC]) = Tr([BC]A)$ and similarly, $Tr(B[CA]) = Tr([CA]B)$. By the transitive property, all cycle permutations are*

*equivalent.*

**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 $\Sigma$ 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 $\Sigma$, do we have to store every entry?

**Solution D:** *We need to store k columns of length N from U and V, which is 2Nk values. Furthermore, we need to store k diagonal elements from $\Sigma$ for a total of k(2N+1) values. This is less than the $N^2$ values needed to store the whole matrix when $k < \frac{N^2}{2N+1}$.*

## 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 $\Sigma$ 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 $\Sigma$ of size $D \times N$, where $\Sigma$ 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 $\Sigma'$ is the $N \times N$ matrix consisting of the first $N$ rows of $\Sigma$, 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:** *Each element of matrix multiplication can be expressed as:*

$$(U\Sigma)_{ij} = \sum_{k=1}^{D} U_{ik}\Sigma_{kj} = \sum_{k=1}^{N} U_{ik}\Sigma_{kj} + \sum_{k=N+1}^{D} U_{ik}\Sigma_{kj} \tag{4}$$

*Notice that the second term, $\sum_{k=N+1}^{D} U_{ik}\Sigma_{kj} = 0$, because $\Sigma$ is only non-zero at $(\Sigma)_{ii}$ where $i \in \{1, \ldots, N\}$ (in this case N is the rank of the matrix X). Thus,*

$$(U\Sigma)_{ij} = \sum_{k=1}^{N} U_{ik}\Sigma_{kj} = (U'\Sigma')_{ij} \tag{5}$$

*where $U'$ and $\Sigma'$ are defined in the problem statement.*

**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:** *$U'$ is D x N, which suggests that $U'U'^T$ is D x D while $U'^TU'$ is N x N. These matrices cannot be equal to each other.*

**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:** *The columns of $U'$ are still orthonormal, which means that the dot product of a column with itself =1, but the dot product of two different columsn = 0. Thus, in the (i,j) entry of matrix product, $U'^TU'$, only the entries where i=j will be 1, whereas the remaining entries will be zero. This correspond to the $I_{N\times N}$ identity matrix.*

*On the other hand, $U'U'^T \neq I_{D\times D}$. Because $U'$ has more rows than columns, it is not possible for the rows of $U'$ to be linearly independent, thus they cannot be orthonormal.*

## 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 $\Sigma$ 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 11 for the definition of pseudoinverse.

**Solution H:** *$\Sigma$ is defined as a diagonal matrix with entries $\sigma_i$, with the assumption that $\sigma_i$ are nonnegative. Thus, $\Sigma^{-1}$ is a diaognal matrix with entries $1/\sigma_i$. Similarly, $\Sigma^+$ is defined as being a diaognal matrix with entries $1/\sigma_i$ if $\sigma_i > 0$ and 0 otherwise, which means that $\Sigma^+ = \Sigma^{-1}$ and thus the expressions are equivalent.*

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

**Solution I:**

$$X^{+'} = (X^T X)^{-1} X^T \tag{6}$$

*substituting* $X = U\Sigma V^T$,

$$X^{+'} = ((U\Sigma V^T)^T (U\Sigma V^T))^{-1} (U\Sigma V^T)^T \tag{7}$$

$$X^{+'} = (V\Sigma U^T U\Sigma V^T)^{-1} (V\Sigma U^T) \tag{8}$$

$$X^{+'} = (V\Sigma^2 V^T)^{-1} V\Sigma U^T \tag{9}$$

*Using the fact that* $\Sigma$ *is diagonal,*

$$X^{+'} = (V^T)^{-1} \Sigma^{-2} V^{-1} V\Sigma U^T \tag{10}$$

*Using the fact that* $V^T = V^{-1}$ *because V is orthogonal:*

$$X^{+'} = V\Sigma^{-2}\Sigma U^T \tag{11}$$

*Finally, using the fact that* $\Sigma$ *is diagonal,*

$$X^{+'} = V\Sigma^{-1} U^T \tag{12}$$

**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.

*Hint*: Note that the transpose of a matrix is easy to compute. Compare the condition numbers of $\Sigma$ and $X^T X$. The condition number of a matrix $A$ is given by $\kappa(A) = \frac{\sigma_{max}(A)}{\sigma_{min}(A)}$, where $\sigma_{max}(A)$ and $\sigma_{min}(A)$ are the maximum and minimum singular values of $A$, respectively.

**Solution J:**

$$\kappa(\Sigma) = \frac{\sigma_{max}(\Sigma)}{\sigma_{min}(\Sigma)} \tag{13}$$

*Recall that*

$$X^T X = (U\Sigma V^T)^T U\Sigma V^T = V\Sigma U^T U\Sigma V^T = V\Sigma^2 V^T \tag{14}$$

*Thus, the singular values of* $X^T X$ *are the singular values (diagonal entries) of* $\Sigma^2$*:*

$$\kappa(X^T X) = \frac{\sigma_{max}(\Sigma^2)}{\sigma_{min}(\Sigma^2)} \tag{15}$$

*Since the singular values are positive,* $\sigma_{max} > \sigma_{min}$ *and* $\kappa(X^T X) > \kappa(\Sigma)$*. In conclusion, computing the inverse of* $X^T X$ *has a higher condition number than using* $\Sigma$*, which means that the former method is also more numerically unstable.*

## 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 $i^{\text{th}}$ and $j^{\text{th}}$ 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 $\partial_{u_i}$ and $\partial_{v_j}$ 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 $\eta$ is the learning rate.

**Solution A:**

$$||U||_F^2 = \sum_i u_i^T u_i \tag{16}$$

$$\partial_{u_i} = \frac{\lambda}{2}(2u_i) + \frac{1}{2}\sum_j (-v_j)(2(y_{ij} - u_i^T v_j)) = \lambda u_i - \sum_j v_j (y_{ij} - u_i^T v_j)^T \tag{17}$$

*Similarly,*

$$\partial_{v_i} = \lambda v_i - \sum_i u_i (y_{ij} - u_i^T v_j)^T \tag{18}$$

**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:**

$$\partial_{u_i} = \lambda u_i - \sum_j v_j (y_{ij} - u_i^T v_j)^T = 0 \tag{19}$$

$$\lambda u_i = \sum_j (y_{ij} v_j - v_j v_j^T u_i) \tag{20}$$

$$\lambda u_i = \sum_j y_{ij} v_j - u_i \sum_j v_j v_j^T \tag{21}$$

$$\lambda u_i + u_i \sum_j v_j v_j^T = \sum_j y_{ij} v_j \tag{22}$$

$$(\lambda I + \sum_j v_j v_j^T) u_i = \sum_j y_{ij} v_j \tag{23}$$

$$u_i = (\lambda I + \sum_j v_j v_j^T)^{-1} \sum_j y_{ij} v_j \tag{24}$$

*By symmetry,*

$$v_i = (\lambda I + \sum_i u_i u_i^T)^{-1} \sum_i y_{ij} u_i \tag{25}$$

**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 $i$s, $j$s, 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.

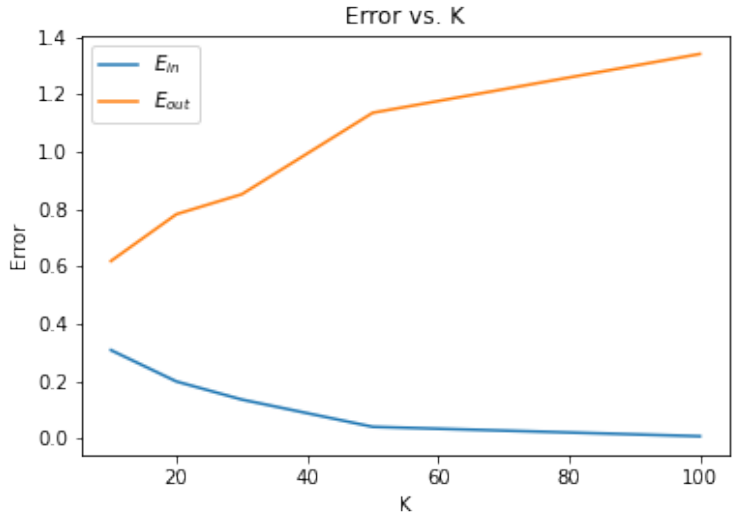
In your implementation, you should:

- 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:**

**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?

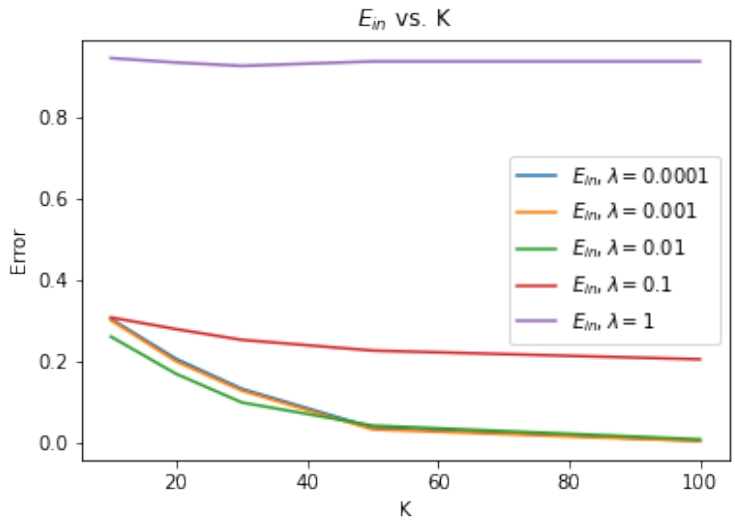
**Solution D:**

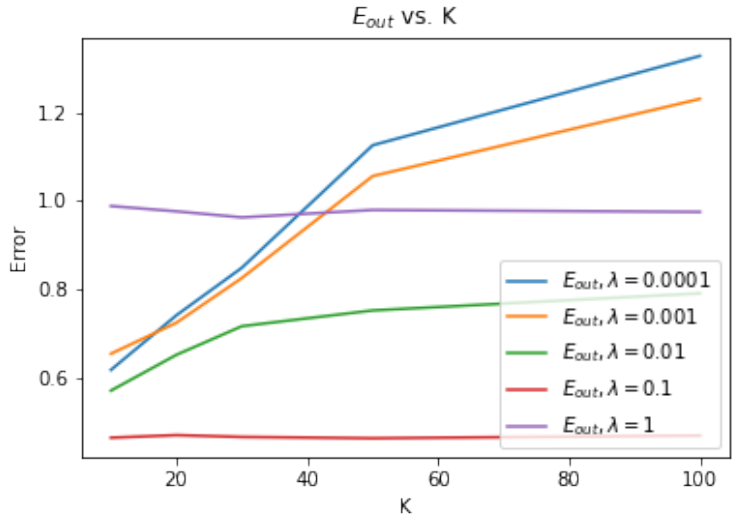



*It appears that there is overfitting as we increase K. As K increases, we use more latent factors, which allows for a higher dimensional and thus better representation of the training data (Ein decreases). However, as K increases, Eout increases.*

**Problem E [5 points]:** Now, repeat problem D, but this time with the regularization term. Use the following regularization values: $\lambda \in \{10^{-4}, 10^{-3}, 0.01, 0.1, 1\}$. 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:**




*In general, the trends are the same as part (d); Ein decreases as K increases, wheras Eout increases as K increases (evidence of overfitting). However, as we increase the regularization penalty, the impact of changing K on both Ein and Eout are dampened (the errors are less affected by K).*

# 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? 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:**

$$\frac{\partial \log p(w_O|w_I)}{\partial w_0} = \frac{v_{w_I} \exp\left({v'}_{w_O}^T v_{w_I}\right)}{\sum_{w=1}^{W} \exp\left({v'_w}^T v_{w_I}\right)} \tag{26}$$

*Similarly, we get a similar numerator for $\frac{\partial \log p(w_O|w_I)}{\partial w_1}$. Thus, the computational cost of calculating the dot product in the numerator scales as $O(WD)$ for each $w_O$, $w_I$ pair.*

**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 |

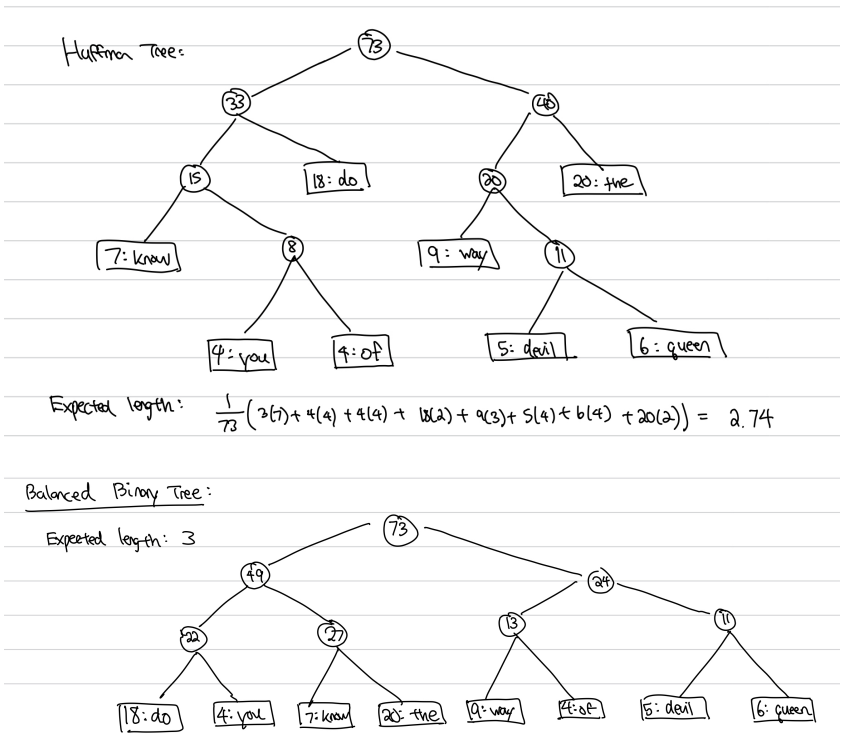
reference, the standard fastText pre-trained word vectors encode approximately $W \approx 218000$ words in $D = 100$ latent dimensions. One trick to get around this is to instead represent the words in a binary tree format and compute the hierarchical softmax.

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 (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:**




**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:** *Using a large D will allow the model to achieve a better training objective, as the representation can capture more information. However, it is not advisable to increase D too much, as the computational cost will increase and the model will overfit. Furthermore, if there is not information bottleneck, the word2vec embedding will not learn anything meaningful about the words.*

## 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 Pytorch. 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 $s = 2$:

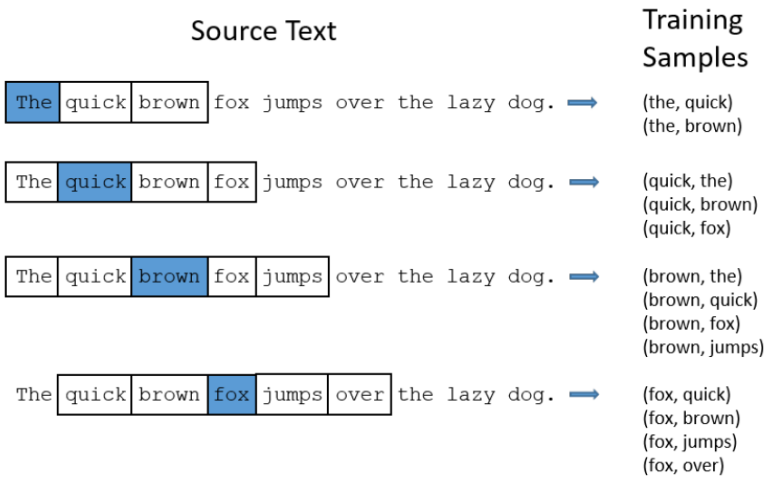



Figure 1: Generating Word2Vec Training Points

(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 set5_prob3.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:**

## 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), as there are 308 unique words and 10 units in the embedding dimension.*

**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(likes, drink), Similarity: 0.98902935*
*Pair(drink, likes), Similarity: 0.98902935*
*Pair(upon, grows), Similarity: 0.9775233*
*Pair(grows, upon), Similarity: 0.9775233*
*Pair(gone, tomorrow), Similarity: 0.97598*
*Pair(tomorrow, gone), Similarity: 0.97598*
*Pair(off, cold), Similarity: 0.9744982*
*Pair(cold, off), Similarity: 0.9744982*
*Pair(wink, pink), Similarity: 0.97420156*
*Pair(pink, wink), Similarity: 0.97420156*
*Pair(down, town), Similarity: 0.9718718*
*Pair(town, down), Similarity: 0.9718718*
*Pair(stick, only), Similarity: 0.971735*
*Pair(only, stick), Similarity: 0.971735*
*Pair(there, here), Similarity: 0.97125745*
*Pair(here, there), Similarity: 0.97125745*
*Pair(took, down), Similarity: 0.96773994*
*Pair(eight, nine), Similarity: 0.96755034*
*Pair(nine, eight), Similarity: 0.96755034*
*Pair(read, walked), Similarity: 0.9647553*
*Pair(walked, read), Similarity: 0.9647553*
*Pair(today, tomorrow), Similarity: 0.96456414*

*Pair(eggs, ham), Similarity: 0.9641697*
*Pair(ham, eggs), Similarity: 0.9641697*
*Pair(heads, upon), Similarity: 0.96246827*
*Pair(foot, off), Similarity: 0.96228147*
*Pair(milk, kind), Similarity: 0.9609818*
*Pair(kind, milk), Similarity: 0.9609818*
*Pair(told, read), Similarity: 0.9546322*
*Pair(samiam, anywhere), Similarity: 0.9541136*

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

**Solution H:** *Many of the word pairs have similar meaning, such as (here, there) and (eight, nine). Furthermore, many of the word pairs rhyme such as (wink, pink) and (down, town). This makes sense in the context of a Dr. Seuss poem. Since similarity is a symmetric metric, inverting the word pair produces the same similarity.*