

Policies

- Due 9 PM PST, February 16th 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 → 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: By the definition of SVD and PCA, $X = U\Sigma V^T$ and $XX^T = U\Lambda U^T$. Substituting the SVD definition of X to find the principal components:

$$XX^T = (U\Sigma V^T)^T(U\Sigma V^T) \quad (1)$$

$$= U\Sigma V^T V \Sigma U^T \quad (2)$$

$$= U\Sigma^2 U^T \quad (3)$$

Note that we since U and V are defined to be orthogonal matrices, we use the property $V^T V = V V^T = I$ in step 2. Furthermore, in the final step, we see that we have the form of the definition of PCA, with $\Lambda = \Sigma^2$. As such, the singular values of X have a squared relationship with the eigenvalues of XX^T : $\lambda_i = \sigma_i^2$ (or $\sigma_i = \sqrt{\lambda_i}$).

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: Intuitively, we know that the eigenvectors of PCA provides the vectors (i.e. new axes after a rotation) of maximum variance and the eigenvalues provide the total variation along these vectors, with the eigenvectors and eigenvalues ordered from largest total variation to least. As such, the eigenvalues must be the same sign and non-negative, as as a opposite signed variations in the same diagonal matrix do not make intuitive sense (negative variation would just be interpreted as a positive value).

Mathematically, the total variation along an axes defined by the eigenvector u_i is defined as $\lambda_i = \sum_j (u_i^T x_j)^2$. As such, you can see that the squared term requires λ_i to 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., $\text{Tr}(ABC) = \text{Tr}(BCA) = \text{Tr}(CAB)$). 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: First let's define two matrices $A = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix}$ and $B = \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix}$. Taking the matrix product of AB and BA , we find:

$$AB = \begin{bmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} \end{bmatrix}$$

$$BA = \begin{bmatrix} a_{1,1}b_{1,1} + a_{2,1}b_{1,2} & a_{1,2}b_{1,1} + a_{2,2}b_{1,2} \\ a_{1,1}b_{2,1} + a_{2,1}b_{2,2} & a_{1,2}b_{2,1} + a_{2,2}b_{2,2} \end{bmatrix}$$

From this, we can see that $\text{Tr}(AB) = \text{Tr}(BA) = a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + a_{2,1}b_{1,2} + a_{2,2}b_{2,2}$. Given that we know this commutes for any two matrices, we can extend this other matrices by expressing representing a matrix product as a single matrix. That is, for $M^{(1)} = AB$, $\text{Tr}(M^{(1)}C) = \text{Tr}(CM^{(1)}) = \text{Tr}(ABC) = \text{Tr}(CAB)$. Expressed as a sum, we see that $\sum_i (M^{(1)}C)_{ii} = \sum_i (ABC)_{ii}$. We can show the same for the other matrix product $M^{(2)} = BC$: $\text{Tr}(M^{(2)}C) = \text{Tr}(CM^{(2)}) = \text{Tr}(BCA) = \text{Tr}(ABC)$, and we can use the shared equivalence to $\text{Tr}(ABC)$ to show that all three permutations are equal. In demonstrating that a matrix can be represented as a matrix product of a another matrix, the cyclic permutation holds for any number of square matrices.

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: Each of the orthogonal matrices U and V is size $N \times K$. Additionally, for the diagonal matrix Σ we only need to store the non-zero terms along the diagonal (K in all). As such, truncated for SVD we need to store $2NK + K$ values. Comparing this to N^2 to find the value of K that makes SVD more efficient:

$$N^2 > 2NK + K$$

$$\text{Values of } K \text{ which makes SVD more efficient: } K < \frac{N^2}{2N + 1}$$

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 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 $(\Sigma)_{ii}$, $i \in \{1, \dots, K\}$, 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: Given that $D > N$, we can visualize the term $U\Sigma$ to be:

$$\begin{bmatrix} u_{11} & . & . & . & . & u_{1D} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ u_{D1} & . & . & . & . & u_{DD} \end{bmatrix} \begin{bmatrix} \sigma_1 & . & . & . \\ . & . & . & . \\ . & . & . & . \\ . & . & . & \sigma_N \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

To reiterate, U is a $D \times D$ matrix and Σ is a $D \times N$ matrix with all non-zero values along the diagonal (because it is defined as rank N), and 0 everywhere else. Notably, is all zeros in all rows greater than N . As such, the matrix product between columns of U greater than N and rows of Σ greater than N will also be all zeros. As such, we can truncate both the columns of U to be size N and the rows of Σ to be size N . This leaves us with U' with size $D \times N$ and Σ' with size $N \times N$ (i.e. $U\Sigma = U'\Sigma'$).

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 $AA^T = A^T A = I$.

Solution F: Since U' is a matrix of size $D \times N$, the matrix product $U'U'^T$ is $D \times D$. Conversely, the matrix product of $U'^T U'$ is $N \times N$. As such $U'U'^T \neq U'^T U'$ because they are not the same size, meaning U' (or any non-square matrix) cannot be orthogonal.

Problem G [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 $U'U'^T = I_{D \times D}$? Why or why not? Note that the columns of U' are still orthonormal. Also note that orthonormality implies linear independence.

Solution G: By definition, because the columns of U' are orthonormal, this means that for the dot product of a given column of U' with itself is equal to 1 and 0 with any other column:

$$u_i'^T u_j' = \begin{cases} 1 & \text{for } i = j \\ 0 & \text{for } i \neq j \end{cases}$$

Expanding this to all of U' the matrix product $U'^T U'$ becomes the identity matrix (i.e. the diagonal is all 1's due to the case where column $i = j$ and 0 everywhere else) of size $N \times N$. Assuming that the rows of U' are no longer orthonormal using the "thin" SVD, this means that $U' U'^T$ is not the identity matrix, as the matrix product is not acting on the columns of U' , which are 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 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 11 for the definition of pseudoinverse.

Solution H: Σ is defined as:

$$\Sigma = \begin{bmatrix} \sigma_1 & . & . & 0 \\ . & \sigma_2 & . & 0 \\ . & . & . & 0 \\ 0 & . & . & \sigma_N \end{bmatrix}$$

And similarly Σ^+ is defined as:

$$\Sigma^+ = \begin{bmatrix} \frac{1}{\sigma_1} & . & . & 0 \\ . & \frac{1}{\sigma_2} & . & 0 \\ . & . & . & 0 \\ 0 & . & . & \frac{1}{\sigma_N} \end{bmatrix}$$

Given the definition of an invertible matrix, we know that $\Sigma \Sigma^{-1} = \Sigma^{-1} \Sigma = I$. We then can make the claim that

$$\Sigma^{-1} = \begin{bmatrix} \frac{1}{\sigma_1} & . & . & 0 \\ . & \frac{1}{\sigma_2} & . & 0 \\ . & . & . & 0 \\ 0 & . & . & \frac{1}{\sigma_N} \end{bmatrix}$$

and we see that:

$$\Sigma \Sigma^{-1} = \Sigma^{-1} \Sigma = \begin{bmatrix} \frac{\sigma_1}{\sigma_1} & \cdot & \cdot & 0 \\ \cdot & \frac{\sigma_2}{\sigma_2} & \cdot & 0 \\ \cdot & \cdot & \cdot & 0 \\ 0 & \cdot & \cdot & \frac{\sigma_N}{\sigma_N} \end{bmatrix} = I$$

As such, $\Sigma^+ = \Sigma^{-1}$ and $X^+ = V \Sigma^+ U^T = V \Sigma^{-1} U^T$.

Problem I [4 points]: Another expression for the pseudoinverse 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 I: Given that $X = U \Sigma V^T$ (assuming this is the thin SVD, we can plug this into the least squares solution $X^{+'} = (X^T X)^{-1} X^T$:

$$X^{+'} = [(U \Sigma V^T)^T U \Sigma V^T]^{-1} (U \Sigma V^T)^T \quad (4)$$

$$= (V \Sigma U^T U \Sigma V^T)^{-1} V \Sigma U^T \quad (5)$$

$$= (V \Sigma^2 V^T)^{-1} V \Sigma U^T \quad (6)$$

Note that between lines (5) and (6), we used the property that $U^T U = I$ for the thin SVD that we proved in solution G. For this inverse term, we will again use the definition of an invertible matrix to determine the inverse term in line (6).

Claim: $(V \Sigma^2 V^T)^{-1} = V \Sigma^{-2} V^T$. Note that for an orthogonal matrix, $V^T = V^{-1}$ by definition and that we are assuming Σ is invertible. Using this expression of the inverse to check if it satisfies the definition of an inverse matrix:

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

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

We can see that our claim is correct, and we can continue from line (6):

$$X^{+'} = V \Sigma^{-2} V^T V \Sigma U^T \quad (7)$$

$$= V \Sigma^{-2} \Sigma U^T \quad (8)$$

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

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 Σ 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: *The solution in problem I is likely to be prone to numerical errors because there is a squared term when trying to solve for the pseudoinverse (the $X^T X$ term), whereas the solution in problem H simply takes the reciprocal of the diagonal terms. Since we know that the values of Σ are non-negative, let's assume that $\sigma_{\max}(\Sigma) = 100$ and $\sigma_{\min}(\Sigma) = 0.01$.*

$$\begin{aligned} \text{for } \Sigma^+: \kappa(\Sigma^+) &= \frac{\frac{1}{0.01}}{\frac{1}{100}} = 10,000 \\ \text{for } \Sigma^2: \kappa(\Sigma^2) &= \frac{100^2}{0.01^2} = 100,000,000 \end{aligned}$$

As such, we are much more prone to numerical errors when solving the pseudoinverse using the least squares solution.

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:

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

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:

$$\begin{aligned} u_i &= u_i - \eta \partial_{u_i} \\ v_j &= v_j - \eta \partial_{v_j} \end{aligned}$$

where η is the learning rate.

Solution A:

$$\begin{aligned} Y &= \frac{\lambda}{2} (\|U\|_F^2 + \|V\|_F^2) + \frac{1}{2} \sum_{i,j} (y_{ij} - u_i^T v_j)^2 \\ \partial_{u_i} &= \frac{1}{\partial u_i} \left(\frac{\lambda}{2} \left(\sum_i u_i^T u_i + \sum_j v_j^T v_j \right) + \frac{1}{2} \sum_{i,j} (y_{ij} - u_i^T v_j)^2 \right) \\ \partial_{u_i} &= \lambda u_i - \sum_j v_j (y_{ij} - u_i^T v_j) \\ \partial_{v_j} &= \frac{1}{\partial v_j} \left(\frac{\lambda}{2} \left(\sum_i u_i^T u_i + \sum_j v_j^T v_j \right) + \frac{1}{2} \sum_{i,j} (y_{ij} - u_i^T v_j)^2 \right) \\ \partial_{v_j} &= \lambda v_j - \sum_i u_i (y_{ij} - u_i^T v_j) \end{aligned}$$

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:

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

$$\lambda u_i - \sum_j v_j y_{ij} + u_i^T \sum_j v_j^T v_j = 0$$

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

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

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

$$\lambda v_j - \sum_i u_i y_{ij} + v_j^T \sum_i u_i^T u_i = 0$$

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

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

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.

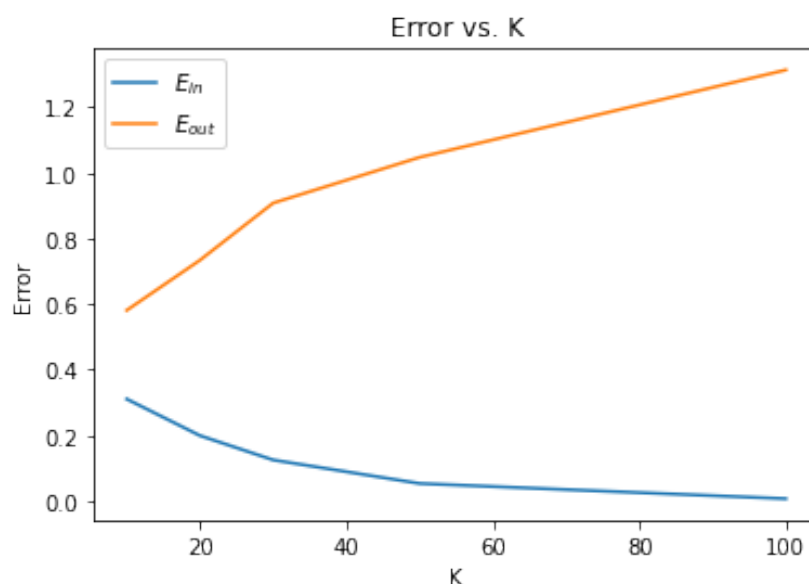
In your implementation, you should:

- 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 $\Delta_{t,i-1}$ is defined analogously, then stop after epoch t if $\Delta_{t-1,t}/\Delta_{0,1} \leq \epsilon$.

Solution C: [Link to Google Colab notebook for Problem 2.](#)

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 $\frac{1}{2} \sum_{i,j} (y_{ij} - u_i^T v_j)^2$. What trends do you notice in the plot? Can you explain them?

Solution D:

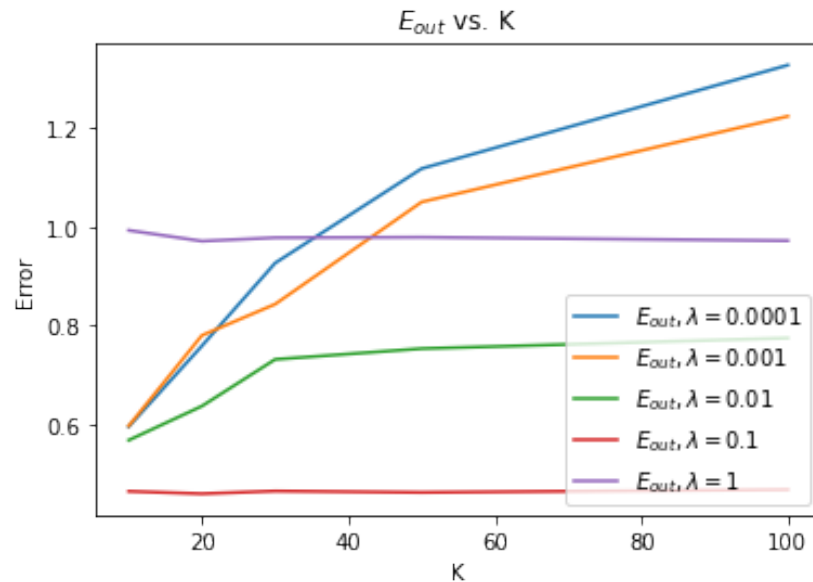


We notice that as we increase the latent factors k , we decrease the training error (E_{in}) but increase the test error (E_{out}). This makes sense since with higher values of k , we are effectively overfitting our training set, which is

giving us lower training error but higher test error.

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:



For higher values of regularization (i.e. $\lambda = 1$ and $\lambda = 0.1$), we see that the test error is more or less constant for various values of K . This makes sense given our conceptual understanding of what regularization does: we are penalizing our loss function to prevent overfitting, so larger coefficients λ are able to prevent overfitting that we see without using regularization at large values of K . This is further supported by the trends we see for the lower regularization coefficients, particularly $\lambda = 0.001$ and $\lambda = 0.0001$, where we see the same trend as in part 2D, where the test error increases with increasing K . Two interesting points to note from the graph: using $\lambda = 0.01$ results in slight increase in test error, but this tapers off and becomes relatively constant after $K > 30$. Another interesting point to note, despite having a large regularization coefficient, $\lambda = 1$ still has a relatively high test error. This could be explained by being initialized in or near a difficult location, which because the regularization coefficient is so large, makes it difficult to traverse out of.

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:

$$p(w_O|w_I) = \frac{\exp(v'_{w_O} v_{w_I})}{\sum_{w=1}^W \exp(v'_w v_{w_I})} \quad (2)$$

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

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

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

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:

$$\begin{aligned} \nabla_{w_I} \log p(w_O|w_I) &= \nabla_{w_I} \log \left(\frac{\exp(v'_{w_O} v_{w_I})}{\sum_{w=1}^W \exp(v'_w v_{w_I})} \right) \\ &= \nabla_{w_I} \left((v'_{w_O} v_{w_I}) - \log \left(\sum_{w=1}^W \exp(v'_w v_{w_I}) \right) \right) \\ &= v'_{w_O} - \frac{\sum_{w=1}^W v'_w \exp(v'_w v_{w_I})}{\sum_{w=1}^W \exp(v'_w v_{w_I})} \end{aligned}$$

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

Because v is a vector of length D , the $v_w'^T v_{w_I}$ terms have complexity $\mathcal{O}(D)$. Additionally, each summation has W terms, which has complexity $\mathcal{O}(W)$. So the total complexity of calculating a gradient for a single pair is $\mathcal{O}(W) + \mathcal{O}(D)$, which considering W can encode 10^5 words or more, a single gradient calculation is very computationally expensive (and still needs to scale for every word 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 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.

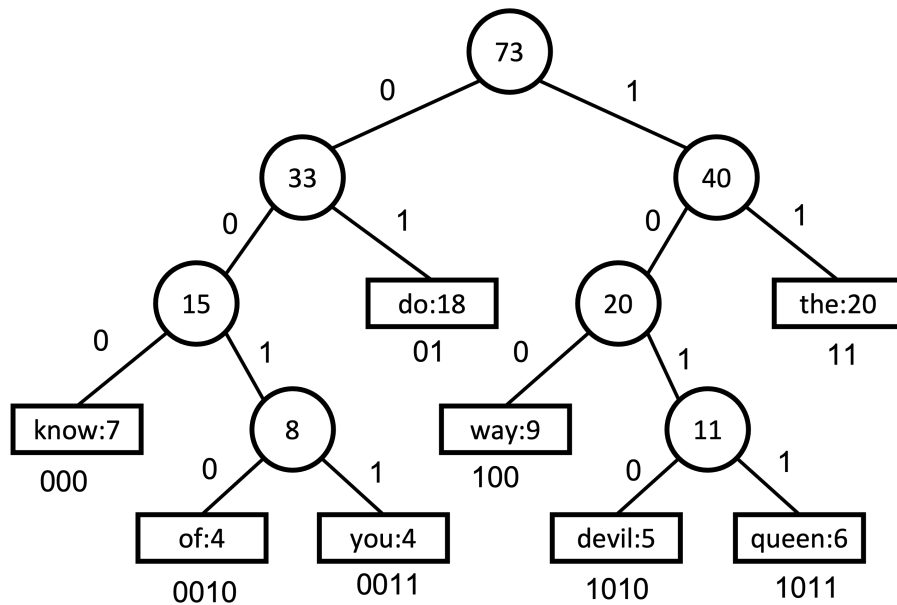
When the words have all the same frequency, then any balanced binary tree will minimize the average representation length and maximize computational efficiency of the hierarchical softmax. But in practice, words occur with very different frequencies — words like “a”, “the”, and “in” will occur many more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

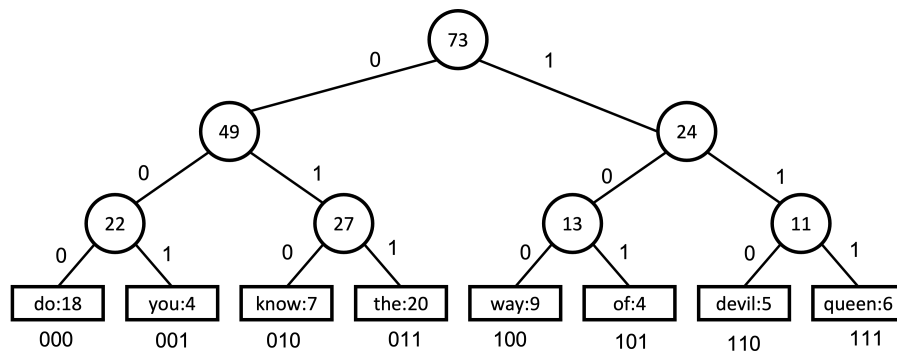
The representation length of a word is then the length of the path (the number of edges) from the root to the leaf node corresponding to the word. For each tree you constructed, compute the expected representation length (averaged over the actual frequencies of the words).

Solution B:

Huffman tree: Expected representation length: 2.740



Balanced binary tree: Expected representation length: 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 we increase D , we will effectively be over fitting our training data set. As such, when we deploy our model to a real world or live application, we wouldn't expect our model to be generalizable and accurate. As such, it is probably not wise to use a very large D as we want a model that can accurately predict a wide range of data.

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 $s = 2$:

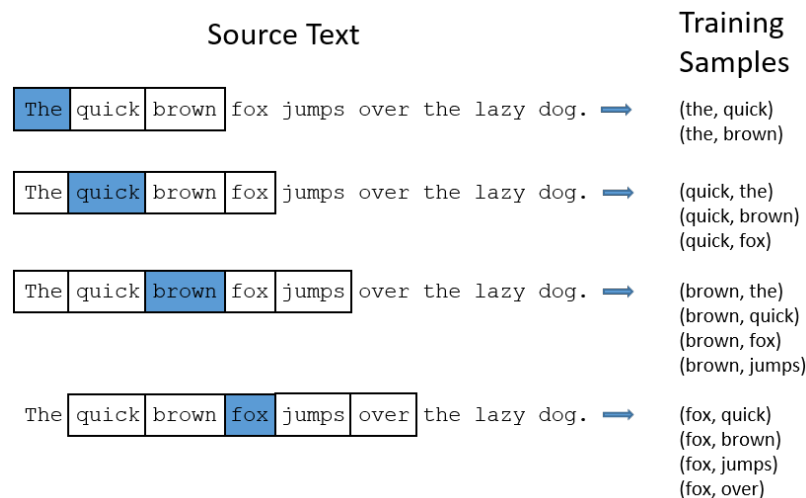


Figure 1: Generating Word2Vec Training Points

- (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: [Link to Google Colab notebook for Problem 3.](#)

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: 10×308

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

Solution F: 308×10

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

Solution G:

1. *(shoe, off)*
2. *(off, shoe)*
3. *(foot, shoe)*
4. *(long, way)*

5. (*way, long*)
6. (*cold, foot*)
7. (*today, gone*)
8. (*gone, today*)
9. (*sad, glad*)
10. (*glad, sad*)
11. (*green, eggs*)
12. (*eggs, green*)
13. (*brush, comb*)
14. (*comb, brush*)
15. (*finger, top*)
16. (*top, finger*)
17. (*yet, met*)
18. (*met, yet*)
19. (*likes, drink*)
20. (*drink, likes*)
21. (*eight, nine*)
22. (*nine, eight*)
23. (*did, ever*)
24. (*ever, did*)
25. (*star, walked*)
26. (*walked, star*)
27. (*gox, socks*)
28. (*socks, gox*)
29. (*ying, sings*)
30. (*sings, ying*)

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

Solution H: Interestingly, it seems that word pairs and the reciprocals (i.e. (shoe, off) and (off, shoe)) seem to cluster together due to having similar similarity scores. Intuitively this makes sense since words in a given window will tend to cluster together and be associated with each other. One interesting outlier is (foot, shoe), as (shoe, foot) is not present in the top 30. Additionally, there does seem to be some type of associated meaning encoded with word pairings (e.g. (shoe, off), (long, way), (cold, foot), (comb, brush), (likes, drink), (eight, nine)). There is also a decent number of word pairings that rhyme, which makes sense given the data set (e.g. (sad, glad), (yet, met), (gox, socks), (ying, sings)). And then of course, we see that (green, eggs) made its way into the top 30 as well.