Policies

• Due 9 PM PST, February 22nd on Gradescope. (Using 1 late hour)

• 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

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:

From definition of SVD, we know, given X, $X = U\Sigma V^T$

We also know the solution for PCA given X and assuming zero mean is $XX^T = U\Lambda U^T$.

Thus, observe the following...

XX^T [SVD: $X = U\Sigma V^T$]

$(U\Sigma V^T)(U\Sigma V^T)^T$ [Distribution of Transpose]

$U\Sigma V^T(V^T)^T\Sigma^T U^T$ [$(V^T)^T = V$ by Transpose Properties; $\Sigma^T = \Sigma$ bc Σ is Diagonal]

$U\Sigma V^T V\Sigma U^T$ [$V^T V = I$ by Transpose Properties]

$U\Sigma^2 U^T$

Therefore, we have $XX^T = U\Lambda U^T = U\Sigma^2 U^T$. This shows the relationship between the SVD of X and XX^T. It reveals that $\Lambda = \Sigma^2$ as the U matrices of SVD and PCA are equivalent where the eigenvalues of XX^T (located within Λ) equal the square of the singular values of X (located within Σ).

The interpretation of PCA is that the columns of U are the eigenvectors and directions of greatest variance along which we can construct new bases for X, and the values λ along the diagonal of $\Lambda = \Sigma^2$ are the eigenvalues which evince the total variance of the dataset along its associated column/eigenvector basis where the greatest variance is along the first columns indicated by the largest corresponding values for λ. Thus, the principal components of X of which we construct new bases for our dataset are along the columns of the U matrix from PCA, so since we've shown the U matrix from PCA is equal to the U matrix from SVD, we know the columns of the U matrix from SVD are the principal components 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:

From the previous question, we've shown the eigenvalues λ of XX^T (along the diagonal Λ) are the square of the singular values of X. Since all the eigenvalues are the result of squaring, all the eigenvalues of the PCA of X must be nonnegative. Additionally, we know the interpretation of each eigenvalue λ_i is the total variance along its associated eigenvector basis column u_i. Thus, we have an intuitive explanation as the total variation along any column u_i is nonnegative since variance is by definition nonnegative. This also evinces another mathematical justification where for each eigenvalue, $\lambda_i = \sum_{j=1}^{N}(u_i^T x_j)^2$. Since λ_i here is the sum of squared terms, it must be nonnegative.

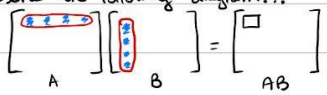
Problem C [5 points]: In calculating the Frobenius and trace matrix norms, we claimed that the trace is invariant under cyclic permutations (i.e., Tr(*ABC*) = Tr(*BCA*) = Tr(*CAB*)). Prove that this holds for any number of square matrices.

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:

Let us first prove $\text{Tr}(AB) = \text{Tr}(BA)$ for any 2 square matrices A, B given that $\text{Tr}(AB) = \sum_{i=1}^{N} (AB)_{ii}$.

Observe the following diagram...



$AB_{11} = A_{row\,1} \cdot B_{col\,1}$

$AB_{11} = A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31} + \ldots + A_{1N}B_{N1}$

$AB_{11} = \sum_{j=1}^{N} A_{1j}B_{j1}$

↓

Generalize for any i: $AB_{ii} = \sum_{j=1}^{N} A_{ij}B_{ji}$

↓

So: $\sum_{i=1}^{N} AB_{ii} = \sum_{i=1}^{N}\sum_{j=1}^{N} A_{ij}B_{ji}$

The key here is realizing that for any $1 \le i \le N$ where N is the dim of A, B, $(AB)_{ii} = \sum_{j=1}^{N} A_{ij}B_{ji}$ such that $\sum_{i=1}^{N}(AB)_{ii} = \sum_{i=1}^{N}\sum_{j=1}^{N} A_{ij}B_{ji}$.

Consider that the element AB_{ii} of any matrix product AB is simply the dot product of the ith row of A and the ith column of B.

Using notation, it becomes clear that $AB_{ii} = \sum_{j=1}^{N} A_{ij}B_{ji}$ for any $1 \le i \le N$.

You can observe my work on the left for justification.

By what we just stated, for $\text{Tr}(AB)$, we have $\text{Tr}(AB) = \sum_{i=1}^{N} (AB)_{ii} = \sum_{i=1}^{N}\sum_{j=1}^{N} A_{ij}B_{ji}$.

Similarly, for $\text{Tr}(BA)$, we have $\text{Tr}(BA) = \sum_{i=1}^{N} (BA)_{ii} = \sum_{i=1}^{N}\sum_{j=1}^{N} B_{ij}A_{ji}$. From here, we can change around the variable names for i and j since they have equivalent bounds and switch around the order of multiplication since multiplication is commutative such that we have $\sum_{i=1}^{N}\sum_{j=1}^{N} B_{ij}A_{ji} = \sum_{j=1}^{N}\sum_{i=1}^{N} B_{ji}A_{ij} = \sum_{i=1}^{N}\sum_{j=1}^{N} B_{ji}A_{ij} = \sum_{i=1}^{N}\sum_{j=1}^{N} A_{ij}B_{ji}$. And so, we get $\text{Tr}(BA) = \sum_{i=1}^{N}\sum_{j=1}^{N} A_{ij}B_{ji} = \text{Tr}(AB)$.

We can then easily generalize this for any number of square matrices $X_1, X_2, \ldots, X_M$ of dim N.

We can show $\text{Tr}(X_1 \ldots X_M) = \text{Tr}(X_{w+1} \ldots X_M X_1 \ldots X_w)$ for any $1 \le w < M$ by grouping matrices together from our 2 case formula.

$\text{Tr}(X_1 \ldots X_M) = \text{Tr}(X_1 \ldots X_w X_{w+1} \ldots X_M) = \text{Tr}((X_1 \ldots X_w)(X_{w+1} \ldots X_M)) \overset{*\text{Using } \text{Tr}(AB) = \text{Tr}(BA)}{=} \text{Tr}((X_{w+1} \ldots X_M)(X_1 \ldots X_w)) = \text{Tr}(X_{w+1} \ldots X_M X_1 \ldots X_w)$.

Therefore, we've generalized this. As an example, its clear having $M=3$ with $w = 1, 2$ proves the 3 matrix case of $\text{Tr}(ABC) = \text{Tr}(BCA) = \text{Tr}(CAB)$.

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: We need to store $2Nk + k$ or $(2N + 1)k$ values for the truncated SVD with k singular values. For matrices U and V, we have N rows for each point and k columns for each new basis axis, yielding $2Nk$ values. Then, for the diagonal matrix Σ, we only have singular values along the diagonal, and we only have to store the first k diagonal entries, yielding another k values. Therefore, we get our $(2N + 1)k$ values. Thus, storing the truncated SVD is more efficient than the whole matrix X as long as $k < N^2 / (2N + 1)$ as that will yield $(2N + 1)k$ values $< N^2$ values.

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:

Similar to $(AB)_{ii} = \sum_{j=1}^{N} A_{ij}B_{ji}$ from 1.C, we can extend this such that $(AB)_{ij} = \sum_{\ell=1}^{M} A_{i\ell}B_{\ell j}$ for the element of the product of any two matrices A with dim X by M and B with dim M by Y such that M is the dimension A and B share.

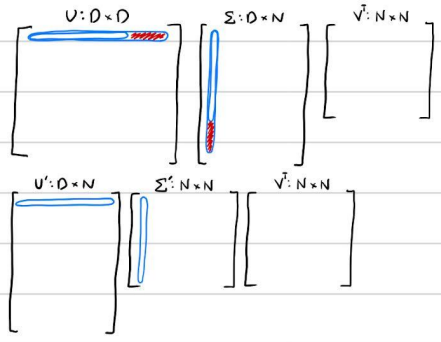
Consider that we can apply this formula to $U\Sigma$ such that U with dim D by D is A and Σ with dim D by N is B. Thus, observe the following...

$(U\Sigma)_{ij} = \sum_{\ell=1}^{D} U_{i\ell}\Sigma_{\ell j}$ [By properties of summation]

$(U\Sigma)_{ij} = \sum_{\ell=1}^{N} U_{i\ell}\Sigma_{\ell j} + \sum_{\ell=N+1}^{D} U_{i\ell}\Sigma_{\ell j}$ [Σ only has nonzero on Σ_{ii} with $1 \le i \le N$ (Given Rank of $X = N$), so $\Sigma_{\ell j} = 0 \ \forall \ell > N$. Thus, $\sum_{\ell=N+1}^{D} U_{i\ell}\Sigma_{\ell j} = 0$ as $\ell > N$.]

$(U\Sigma)_{ij} = \sum_{\ell=1}^{N} U_{i\ell}\Sigma_{\ell j}$ [This means only first N columns of U and N rows of Σ are present in the computation of $(U\Sigma)_{ij}$ which is the equivalent to the product of $(U'\Sigma')_{ij}$.]

$(U\Sigma)_{ij} = (U'\Sigma')_{ij}$



Therefore, $U\Sigma$ and $U'\Sigma'$ share all the same elements where Σ_i's having zero for all elements Σ_{ii} after $i > N$ had the effect of removing the N+1 to D columns of U and the N+1 to D rows of Σ from the matrix multiplication product.

And, since $U\Sigma$ and $U'\Sigma'$ share all elements for resulting dimensions D by N, $U\Sigma = U'\Sigma'$.

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: U' has the dimensions D by N, so U'^{T} has dimensions N by D. By matrix multiplication, the product U'U'^{T} has dimensions D by D and the product U'^{T}U' has dimensions N by N. Since U'U'^{T} and U'^{T}U' do not have the same dimensions, they cannot be equal i.e U'U'^{T} ≠ U'^{T}U'. Thus, U' cannot be orthogonal as it fails to meet the requirement U'U'^{T} = U'^{T}U' = I.

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: For any two linearly independent vectors, the dot product is zero. However, any orthonormal vector dotted with itself equals 1 since the vectors are parallel in the same direction. Any other situation yields a dot product that's not 0 or 1. Consider that for any matrix product AB, the (i, j) element of AB is equal to the ith row of A dotted with the jth column of B. Thus, the elements of U'^{T}U' are the results of computing the dot product on the columns of U' for every combination of columns. Along the diagonal of U'^{T}U', the values would be each column dotted with itself, so the result is 1 as stated above. For any other elements of U'^{T}U', the values would be each column dotted with some other column, so the result is 0 since the columns are linearly independent. Thus, U'^{T}U' has 1's along the diagonal and 0 everywhere else, yielding the identity matrix $I_{N \times N}$. U'U'^{T}, on the other hand, does not yield an identity matrix. The elements of U'U'^{T} are the results of computing the dot product of the rows of U' for every combination of rows. There are more rows than columns in U', and the rows can be considered a set of D vectors each with N elements. Since D > N, there must be a linear overlap between some of the rows such that all the rows of D cannot be linearly independent. This means there will be some values (off the diagonal) of U'U'^{T} where two different rows that aren't orthonormal dotted together yields a nonzero value. Thus, U'U'^{T} doesn't yield $I_{D \times D}$ because some values off the diagonal are guaranteed to be nonzero.

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: The definition of Σ^+ as given in the lecture slides is the reciprocal of every corresponding singular value σ for every value along the diagonal of Σ^+ if σ is > 0, and 0 if otherwise. Assuming that all the singular values σ are chosen as the positive square roots of their corresponding eigenvalues, then Σ^+ is simply Σ with every singular value replaced with its reciprocal. Now, consider that the inverse of a diagonal matrix is the matrix where every value along the diagonal is replaced with its reciprocal. Thus, assuming that all $\sigma > 0$, Σ^+ and Σ^{-1} are equal such that $X^+ = V\,\Sigma^+ U^T = V\,\Sigma^{-1}U^T$.

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:

We can show $X^{+\prime} = (X^TX)^{-1}X^T = V\Sigma^{-1}U^T$ directly through matrix manipulations.

Note that by definition both U and V are orthogonal. For any orthogonal matrix A, $A^{-1} = A^T$ s.t. $A^TA = AA^T = I$.

Also note that for any diagonal matrix D, $D^T = D$.

Now, observe the following manipulations...

$$X^{+\prime} = (X^TX)^{-1}X^T$$

$$X^{+\prime} = X^{-1}(X^T)^{-1}X^T$$

$$X^{+\prime} = (U\Sigma V^T)^{-1}\left((U\Sigma V^T)^T\right)^{-1}(U\Sigma V^T)^T$$

$$X^{+\prime} = (V^T)^{-1}\Sigma^{-1}U^{-1}\left(V^T\right)^T\Sigma^{-1}U^T)^{-1}(V^T)^T\Sigma U^T$$

$$X^{+\prime} = (V^{-1})^{-1}\Sigma^{-1}U^{-1}(U^T)^{-1}\Sigma^{-1}V^{-1}V\Sigma U^T$$

$$X^{+\prime} = V\Sigma^{-1}(U^TU)^{-1}\Sigma^{-1}(V^{-1}V)\Sigma U^T$$

$$X^{+\prime} = V\Sigma^{-1}(\Sigma^{-1}\Sigma)U^T$$

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

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 *See GitHub.*

Solution J: Condition numbers are used to measure how sensitive a computation is to changes or errors, so we can use the condition numbers of Σ and X^TX to see which expression is more error prone. The condition number of a matrix is its max singular value divided by its min singular value. Since we chose the positive square root for our singular values, we know all singular values are positive, so the max singular value divided by the min singular value will always yield a value > 1 as long as we have more than one singular value (which we can assume). We can first easily compute the condition number of Σ. Since Σ is a diagonal matrix, we know its singular values already, as they are all along its diagonal. So, defining the max and min singular values of Σ respectively as σmax and σmin, the condition number for Σ = (σmax / σmin). Now, for the condition number of X^TX, we need to analyze X^TX first. Let's substitute the SVD of X, yielding $X^TX = (U\Sigma V^T)^T U\Sigma V^T$. From there, we have $X^TX = (U\Sigma V^T)^T U\Sigma V^T = V\Sigma U^T U\Sigma V^T = V\Sigma\Sigma V^T = V\Sigma^2 V^T$. Thus, the singular values of X^TX are defined by the diagonal matrix Σ^2 such that its max and min singular values are σmax^2 and σmin^2 respectively, so the condition number of X^TX is $(\sigma max / \sigma min)^2$. Thus, as we've already established $(\sigma max / \sigma min) > 1$, $(\sigma max / \sigma min)^2 > (\sigma max / \sigma min)$. Since the condition number is larger for X^TX than it is for Σ, the expression involving X^TX is highly prone to numerical errors.

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: *See GitHub*

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

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

where η is the learning rate.

Solution A:

The regularized square error is given by $\frac{\lambda}{2}(\|U\|_F^2 + \|V\|_F^2) + \frac{1}{2}\sum_{i,j}(y_{ij} - u_i^T v_j)^2$. So observe the following...

By definition of Frobenius Norm $\|X\|_F^2 = \sum_{i,j} x_{ij}^2$ where x_{ij} is the elem at (i,j) $= \sum_i x_i^2$ where x_i is the ith column, we can rewrite

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

Now, the gradient $\frac{\partial}{\partial u_i}$, we isolate the summation for a single (ith) column, and compute the derivative from there.

$$\frac{\partial}{\partial u_i}\left(\frac{\lambda}{2}\left(\sum_i u_i^2 + \sum_j v_j^2\right) + \frac{1}{2}\sum_{i,j}(y_{ij} - u_i^T v_j)^2\right) = \frac{\partial}{\partial u_i}\left(\frac{\lambda}{2}u_i^2 + \frac{1}{2}\sum_j (y_{ij} - u_i^T v_j)^2\right) = \frac{\lambda}{2}(2u_i) + \frac{1}{2}\sum_j\left(-2v_j(y_{ij} - u_i^T v_j)^T\right) = \lambda u_i - \sum_j v_j (y_{ij} - u_i^T v_j)^T.$$

Similarly, we do the same isolation and computing with respect to v_j for $\frac{\partial}{\partial v_j}$

$$\frac{\partial}{\partial v_j}\left(\frac{\lambda}{2}\left(\sum_i u_i^2 + \sum_j v_j^2\right) + \frac{1}{2}\sum_{i,j}(y_{ij} - u_i^T v_j)^2\right) = \frac{\partial}{\partial v_j}\left(\frac{\lambda}{2}v_j^2 + \frac{1}{2}\sum_i (y_{ij} - u_i^T v_j)^2\right) = \frac{\lambda}{2}(2v_j) + \frac{1}{2}\sum_i\left(-2u_i(y_{ij} - u_i^T v_j)^T\right) = \lambda v_j - \sum_i u_i (y_{ij} - u_i^T v_j)^T.$$

Therefore, we computed the gradients.

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 can find the closed form for each by setting the gradients to 0, as that will be the optimal u_i and v_j.

$$\frac{\partial}{\partial u_i}\left(\frac{\lambda}{2}\left(\sum_i u_i^2 + \sum_j v_j^2\right) + \frac{1}{2}\sum_{i,j}(y_{ij} - u_i^T v_j)^2\right) = 0$$

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

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

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

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

Similarly, we can calculate the optimal v_j in the same way.

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

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. 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:
https://colab.research.google.com/drive/1hewIKW6uv6hLvZ5DY_WLXU0g-Dd8ec-w?usp=sharing

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, *See GitHub*. What trends do you notice in the plot? Can you explain them?

Solution D: *Graph Below*
The E_{out} is significantly higher than the E_{in} for all K, and the difference is more pronounced with higher K. This means all of our models are overfitting, and this makes sense because we have set regularization to 0. Since we aren't regularizing, we aren't generalizing well for points we don't know and are basically just memorizing the points we've seen. So, our models know very little about new data they haven't seen. This is why we need regularization. The overfitting increases with higher K because K controls model complexity. With higher K, the models can capture more of the intricacies of the input, which makes them more complex; thus we overfit more. Higher complexity equals more overfitting if regularization isn't taking place, so this also makes sense.

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: *Graphs Below*
For most of the lambdas, the trend is the same. In general, testing error is much higher than training error as clear evidence of overfitting. And, as K increases, we can represent more features so the model gets more complex resulting in more overfitting. We can see these trends most pronounced with lambda values of $1e-4$, $1e-3$, and 0.01; as these values share patterns most similar to no regularization, this shows they are not regularizing enough. The models with these lambdas are still too complex and prioritizing memorization over generalizing, leading to increased overfitting as K increases. With lambda values of 0.1 and 1, however, things start to change. For lambda = 1, we are regularizing too much. The regularization term is given too much weight such that the model really only cares about minimizing the norms of U and V, yielding relatively the same predictions for all inputs and poor performance for both training and testing error (both very close to 1.0 error). Because of this pure focus on minimizing U and V, increasing K doesn't have much effect on the model, and it performs poorly for all K. Here, we have too much overfitting. Finally, for lambda = 0.1, we reach the best balance. The testing error is still higher than the training error, but they are very close together for all K (about 0.1 difference). As K increases, training error decreases by a little, but testing error stays about the same. Here, we have reached a good balance of regularization. Overfitting is mitigated (training error and testing error are close to each other), and the model doesn't perform worse with higher K. This is because with effective regularization, we can leverage and balance model complexity towards our learning objective. By having regularization at this point, higher model complexity (higher K) yields lower training error, but not higher testing error because we are still able to generalize well for new data due to good regularization. Out of these lambda, 0.1 has optimal regularization and performs best.

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 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:

See GitHub

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:

To find gradients $\nabla \log p(w_o|w_I)$, first we'll manipulate $\log p(w_o|w_I)$.

Observe the following, assuming log is base e:

$$\log p(w_o|w_I) = \log\left(\frac{e^{v'^{T}_{w_o} v_{w_I}}}{\sum_{w=1}^{W} e^{v'^{T}_{w} v_{w_I}}}\right) = \log\left(e^{v'^{T}_{w_o} v_{w_I}}\right) - \log\left(\sum_{w=1}^{W} e^{v'^{T}_{w} v_{w_I}}\right) = v'^{T}_{w_o} v_{w_I} - \log\left(\sum_{w=1}^{W} e^{v'^{T}_{w} v_{w_I}}\right)$$

Now, we can consider $\nabla \log p(w_o|w_I)$.

$$\nabla_{v_{w_I}}\left(v'^{T}_{w_o} v_{w_I} - \log\left(\sum_{w=1}^{W} e^{v'^{T}_{w} v_{w_I}}\right)\right) = v'_{w_o} - \frac{1}{\sum_{w=1}^{W} e^{v'^{T}_{w} v_{w_I}}} \sum_{w=1}^{W} v'_{w} e^{v'^{T}_{w} v_{w_I}}$$

$$\nabla_{v'_{w_o}}\left(v'^{T}_{w_o} v_{w_I} - \log\left(\sum_{w=1}^{W} e^{v'^{T}_{w} v_{w_I}}\right)\right) = v_{w_I} - \frac{1}{\sum_{w=1}^{W} e^{v'^{T}_{w} v_{w_I}}} v_{w_I} e^{v'^{T}_{w_o} v_{w_I}}$$

From here, we can isolate any summation term i.e $\sum_{w=1}^{W} v'_{w} e^{v'^{T}_{w} v_{w_I}}$ or $\sum_{w=1}^{W} e^{v'^{T}_{w} v_{w_I}}$.

The $v'^{T}_{w} v_{w_I}$ has $\Theta(D)$ time complexity as we need to do multiplication for all D dimensions.

The $\sum_{w=1}^{W}$ summation of $e^{v'^{T}_{w} v_{w_I}}$, thus, produces $\Theta(WD)$ time complexity as $v'^{T}_{w} v_{w_I}$ is done W times.

Therefore, for a single w_o, w_I pair, the time complexity for computing gradients is $\Theta(WD)$.

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 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,

Table 1: Words and frequencies for Problem B

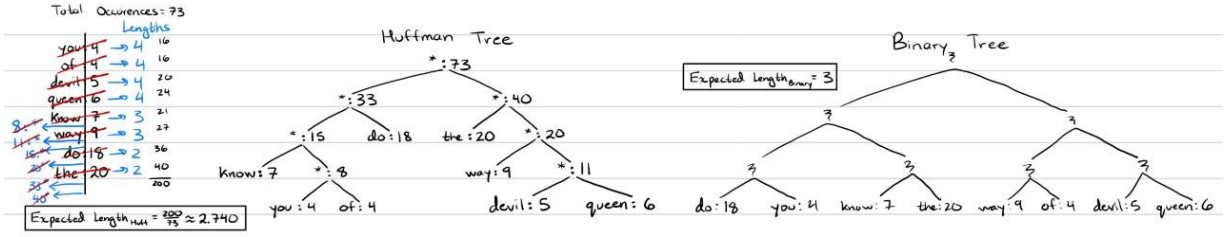
Word	Occurrences
do	18
you	4
know	7
the	20
way	9
of	4
devil	5
queen	6

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:



The expected length of Huffman Tree is approximately 2.740 and the expected length of the Binary Tree is 3, which makes sense why they chose to use a Huffman 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:** Increasing D makes for a better model, but after a certain point once we reach a large value of D, the training objective becomes too complex and may require too much computational effort. For any practical embedding problem like word2vec for example, the training objective is already very complex and gets substantially more complex as we increase D, so after a certain point, the complexity becomes too much and it's unfavorable. The complexity produced from more dimensions also makes our model more likely to overfit our data with large D. Thus, we want to have a substantial D but not too large.

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 2*s* training points for word L_i. The diagram below shows an example of training point generation for *s* = 2:

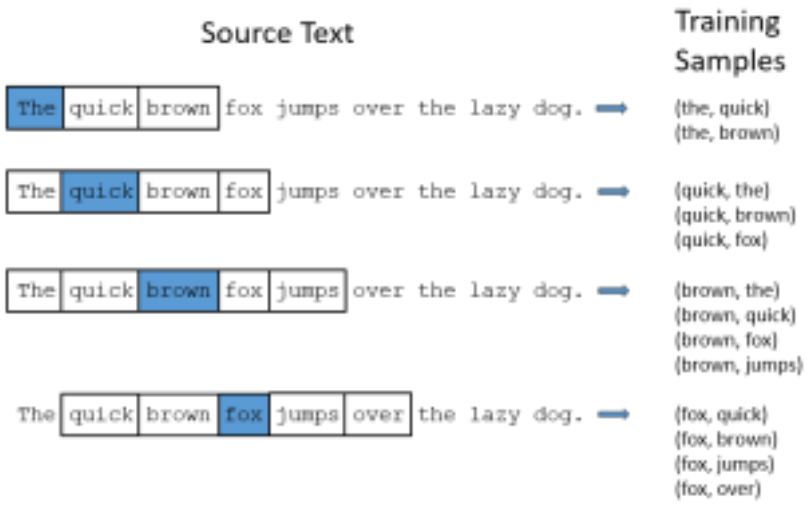


Figure 1: Generating Word2Vec Training Points

(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*
https://colab.research.google.com/drive/1OHy0pvMxefc0l07htVvjBeb5kMV1vaxW?usp=sharing

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: The dimensions are 308 by 10, reflecting the number of words and the embedding dimension.

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

Solution F: The dimensions are 10 by 308 for the same reasons as the hidden layer, except it has to be reversed since it works by receiving the hidden layer's output.

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

Solution G:

```
Pair(pet, wave), Similarity: 0.97445124
Pair(wave, pet), Similarity: 0.97445124
Pair(bike, should), Similarity: 0.966437
Pair(should, bike), Similarity: 0.966437
Pair(bump, did), Similarity: 0.96465135
Pair(did, bump), Similarity: 0.96465135
Pair(glad, cook), Similarity: 0.95977306
Pair(cook, glad), Similarity: 0.95977306
Pair(name, wave), Similarity: 0.9531311
Pair(had, zans), Similarity: 0.9505988
Pair(zans, had), Similarity: 0.9505988
Pair(only, bump), Similarity: 0.94843185
Pair(four, three), Similarity: 0.9446459
Pair(three, four), Similarity: 0.9446459
Pair(grow, glad), Similarity: 0.93765974
Pair(am, cold), Similarity: 0.93736213
Pair(cold, am), Similarity: 0.93736213
Pair(bird, fly), Similarity: 0.9322248
Pair(fly, bird), Similarity: 0.9322248
Pair(star, long), Similarity: 0.93215
Pair(long, star), Similarity: 0.93215
Pair(sleep, brush), Similarity: 0.92933524
Pair(brush, sleep), Similarity: 0.92933524
Pair(thin, cant), Similarity: 0.9292662
Pair(cant, thin), Similarity: 0.9292662
Pair(hold, let), Similarity: 0.9288228
Pair(let, hold), Similarity: 0.9288228
Pair(out, sheep), Similarity: 0.92821527
Pair(sheep, out), Similarity: 0.92821527
Pair(off, top), Similarity: 0.9280913
```

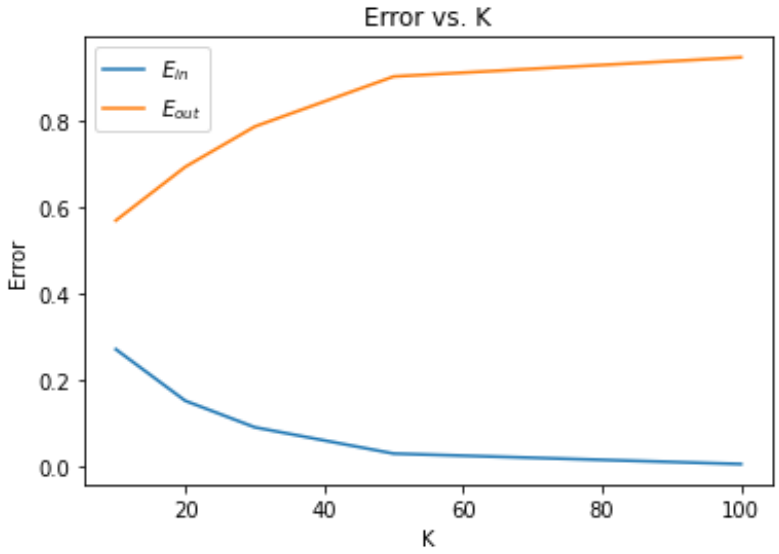
Problem H [2 points]: What patterns do you notice across the resulting pairs of words?

Solution H: Most pairs show up twice in the form (a, b) then (b, a) for any pair of words a, b. This is because the model associates words by their neighbors in both directions, so if words are close to each other, they probably show up in both directions and thus are related in both directions.

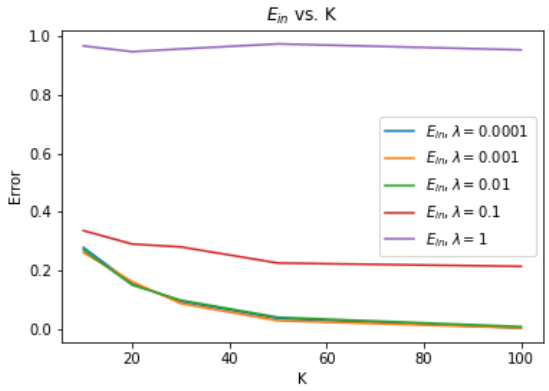
Generated Images

Part 2

Graph for 2D



Graph for 2E: E_{in}



Graph for 2E: E_{out}