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: *Notice that given the singular value decomposition of $X = U\Sigma V^T$, that we can write $XX^T = (U\Sigma V^T)(V\Sigma U^T)$. Given V is an orthogonal matrix $V^TV = I$ and we simplify to $XX^T = U\Sigma^2U^T$. Given U, U^T also orthogonal matrices we see that $XX^T = U\Sigma^2U^T$ is essentially the principal component analysis solution with $\Sigma^2 = \Lambda$, and we prove that U is therefore the matrix in the PCA solution containing the principal components of X. We also notice that the $\Sigma^2 = \Lambda$ implies that the singular values of X squared are equal to 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: *Intuitively the eigenvalues of the PCA of X corresponds to the variance of the data captured by the corresponding principal component to that eigenvalue. Variance by definition ($E(X^2) - E(X)^2$) must be positive so thus each eigenvalue must be positive. Mathematically we proved in part (A) that the singular values of X squared are equal to the eigenvalues of XX^T so thus since the eigenvalues are squares of some other real value all eigenvalues 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:

We can rewrite the Trace as a sum of all elements computed in the Trace:

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

Notice, by symmetry this sums over the same products as:

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

Thus we've proven $Tr(AB) = Tr(BA)$. Using this argument since A and B are square we can extend this to 3 square matrices $Tr(M_1M_2) = Tr(M_2M_1)$, $N \times N$ matrices where we can then write $M_2 = M_3M_4$. In general we can justify that any number product of square matrices can be simplified down to the product of two square matrices for which the rearranging of the product the trace is taken of holds, and thus we can rearrange any number of square matrices and maintain the same trace Tr(ABC) = Tr(BCA) = Tr(CAB) for any extension of ¿3 square matrices as well.

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:

If we were to store the truncated SVD we would store K components from U, V^T of size N each and only need to store the K diagonal components of Σ. This results in a total storage of $2NK + K$ values. To determine if it is more efficient to store the truncation rather than X itself we just need to find K such that $2NK + K < N^2 \rightarrow K < \frac{N^2}{2N+1}$, so we conclude that it is more efficient to store the truncated SVD with K values 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 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: *With $D > N$ and X with rank N we know that $\Sigma_{i,i}$ will only be non_zero for columns and rows of $i \leq N$ leaving $D - N$ rows empty. Thus if we had the original $U\Sigma$ the resulting $D \times N$ matrix would have the last $D - N$ rows of Σ and the last $D - N$ columns of D contribute nothing to the resulting matrix since the last $D - N$ rows of Σ are all 0. As a result we can simply take them out and still achieve the same result which would give us U' $D \times N$ and Σ $N \times N$ producing the same resulting $D \times N$ matrix as it simply cuts out products in the multiplication that were already zero preserving the original values and we conclude that $U\Sigma = U'\Sigma'$*

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

Solution F: *Given U' is not square and has dimension $D \times N$ we can see that $U'U'^T$ is a $D \times D$ matrix and U'^TU' is a $N \times N$ matrix and as a result it is impossible for $U'U'^T = U'^TU'$, and thus the definition of orthogonality of U' as $U'U'^T = U'^TU' = I$ cannot be true and we conclude U' is not orthogonal according to the definition given in class.*

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:

We know that U'^T will have each row orthonormal given that U' is orthonormal along its column vectors. Thus $U'^TU' = M$ will result in the identity matrix since any non-corresponding column by row multiplication of U'^T and U' will be a dot product of two orthonormal vectors and result in $M_{i,j} \neq 0$ for $i \neq j$. FOr $i = j$ the dot product of the same vector orthonormal vector is taken resulting in $M_{i,i} = 1$ giving $U'^TU' = M = I_{N\times N}$. However given $D > N$ notice that $U'U'^T = M_{D\times D}$ but the row vectors of U' and column vectors of U'^T cannot necessarily all be orthogonal to eachother since $D > N$ implies that these vectors cannot form a basis since the amount of vectors D is greater than the dimension of the vectors. This also implies that the vectors cannot all be orthogonal since they are not linearly independent and thus proves that $U'U'^T \neq I_{D\times D}$.

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:

To show $X^+ = V\Sigma^+U^T = V\Sigma^{-1}U^T$ we need to show $\Sigma^+ = \Sigma^{-1}$. Expanding we have that this is equivalent to $(\Sigma^T\Sigma)^{-1}\Sigma^T = \Sigma^-1$. Notice that since Σ is a diagonal matrix since it is the 'thin' SVD case, that $\Sigma^T\Sigma = \Sigma^2$ also diagonal. Taking the inverse of a diagonal matrix can be done by simply inverting all non-negative entries which since Σ^2 is squared must all be non-negative. Thus we know $(\Sigma^T\Sigma)^{-1} = M$ has every $M_{i,i} = \frac{1}{\Sigma_{i,i}^2}$. Right multiplying this then by Σ^T will result in $(\Sigma^T\Sigma)^{-1}\Sigma^T = M'$ where now every diagonal entry $M'_{i,i} = \frac{1}{\Sigma_{i,i}}$. Notice this result of M' implies that $M'_{N\times N} = \Sigma^{-1}$ since Σ by definition in 'thin' SVD is a diagonal matrix and M' definition satisfies that it is the inverse of Σ so we can conclude that $M' = (\Sigma^T\Sigma)^{-1}\Sigma^T = \Sigma^-1$ completing the proof.

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

Solution I:

We can prove this by substituting the 'thin' SVD solution of $X = U\Sigma V^T$:

$$
\begin{aligned}
(X^TX)^{-1}X^T &= ((U\Sigma V^T)^TU\Sigma V^T)^{-1}(U\Sigma V^T)^T \\
&= (V\Sigma U^TU\Sigma V^T)^{-1}(V\Sigma U^T) \\
&\textit{Given } U \textit{ is a square orthogonal matrix: } U^TU = I\textit{:} \\
&= (V\Sigma^2V^T)^{-1}(V\Sigma U^T) \\
&\textit{By properties of matrix inversion:} \\
&= V\Sigma^{-2}V^TV\Sigma U^T \\
&\textit{Since this is a 'thin' SVD case } V \textit{ is also square and orthogonal so } V^TV = I\textit{:} \\
&= V\Sigma^{-2}\Sigma U^T \\
&= V\Sigma^{-1}U^T
\end{aligned}
$$

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

Hint: Note that the transpose of a matrix is easy to compute. Compare the condition numbers of Σ and X^TX. 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:

From (I) we have that $X^TX = V\Sigma^2V^T$ meaning that the the singular value diagonal matrix of X^TX is Σ^2. Thus the condition number of this matrix is $\kappa(X^TX) = \frac{max\Sigma^2}{min\Sigma^2} = (\frac{max\Sigma}{min\Sigma})^2$. We see that this is clearly greater than the condition number on Σ as Σ's diagonal values are its singular values resulting in the comparison

$\kappa(X^TX) = (\frac{max\Sigma}{min\Sigma})^2 > (\frac{max\Sigma}{min\Sigma}) = \kappa(\Sigma)$ so we conclude that $\kappa(X^TX) > \kappa(\Sigma)$ and thus the expression in problem I is more highly prone to numerical errors compared to problem H.

2 Matrix Factorization [30 Points]

Relevant materials: Lecture 11

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

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

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

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

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

where η is the learning rate.

Solution A: *Given*

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

Notice that

$$\partial\|U\|_F^2 = 2\|U\|_F^2 = 2\sum_i u_i^T u_i = u_i$$

Using this, we can evaluate the partials ∂_{u_i} and ∂_{v_j} with the previous conclusion and chain rule resulting in:

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

The partials for both variables follow the same process so we also get:

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

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:

With V fixed we can calculate closed-form solution to u_i using the gradient from part (A) set to 0.

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

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

Now given $y_i jand u_i^T v_j$ are real values not matrices, we can distribute vector v_j. Additionally, given u_i, v_j are vectors and $u_i^T v_j$ essentially their dot product, we can rearrange the equivalent $u_i^T v_j = v_j^T u_i$:

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

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

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

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

The solution process for v_j is analogous to u_i so by the same steps we conclude for v_j:

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

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 $[-0.5, 0.5]$.

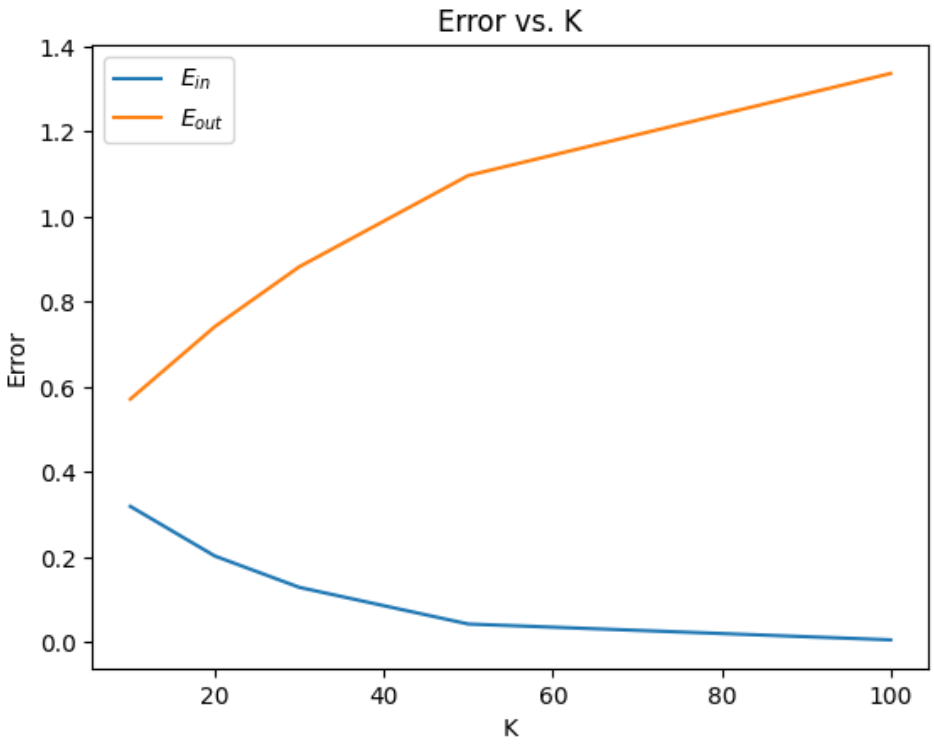
- Use a learning rate of 0.03.
- Randomly shuffle the training data indices before each SGD epoch.
- Set the maximum number of epochs to 300, and terminate the SGD process early via the following early stopping condition:
  - Keep track of the loss reduction on the training set from epoch to epoch, and 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 $\Delta_{i,i-1}$ is defined analogously, then stop after epoch t if $\Delta_{t-1,t}/\Delta_{0,1} \leq \epsilon$.

Solution C:

Colab link to solution

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?

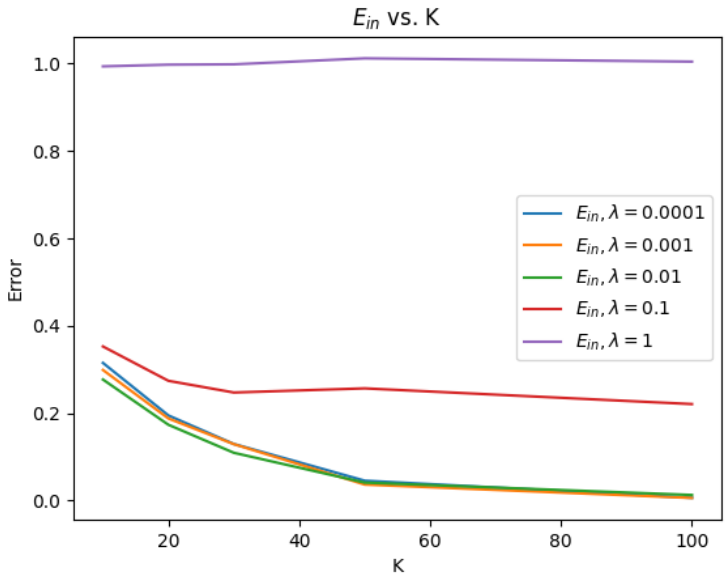
Solution D:

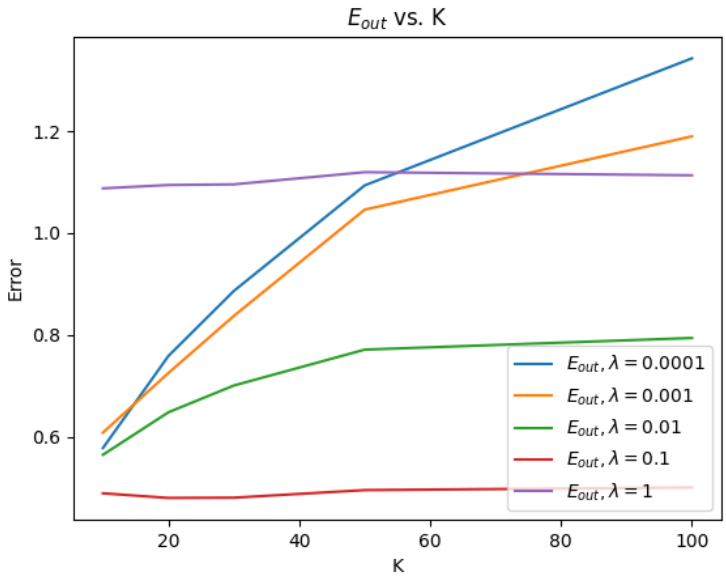


Notice as K increases that the E_{in} converges towards 0 but the E_{out} diverges, suggesting that as K increases and we use more latent factors that the model begins to overfit which makes sense as we may be storing/ trying to find information too specific to the training set and not generalizable when we have a larger K value.

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:





Similar to the previous part we notice a general trend that as K increases, the E_{in} converges to zero unless λ regularization value is high (e.g. 1 or .1) which in those cases the frobenius norm minimization is given priority to the point that the squared loss term does not matter and overall squared loss is not minimized well in training. For E_{out} we again notice that for lower regularization values as K increases the respective E_{out} increases likely for similar reasons as the previous part that the increased amount of latent factors results in overfitting to the

training data. Generally for high $lambda$ regularization terms the E_{out} remains relatively stagnant as the training tends to prioritize the frobenius norm of matrices U, V rather than the squared loss but we do notice that $\lambda = .1$ results in the lowest E_out. In general the trends from the previous part of K increasing also resulting in E_{out} increasing to still exist but be less present as regularization levels go up.

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: *The complexity of the gradient is dependent on* $p(w_0|w_I) = \frac{\exp\left({v'}_{w_O}^T v_{w_I}\right)}{\sum_{w=1}^{W} \exp\left({v'_w}^T v_{w_I}\right)}$ *which has complexity dependent on the denominator (clearly larger complexity than numerator by scale* W*). Summing from every item of length* W *and taking inner product of 2* D*-length items in each iteration results in* $O(WD)$ *computations, for every single pair. Given* W *items there are on the scale of* W^2 *pairs resulting in complexity of* $O(W^2) * O(WD) = O(W^3D)$ *for calculating* $\nabla \log p(w_O|w_I)$ *for a single* w_O, w_I *pair* W^2 *times for each 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.

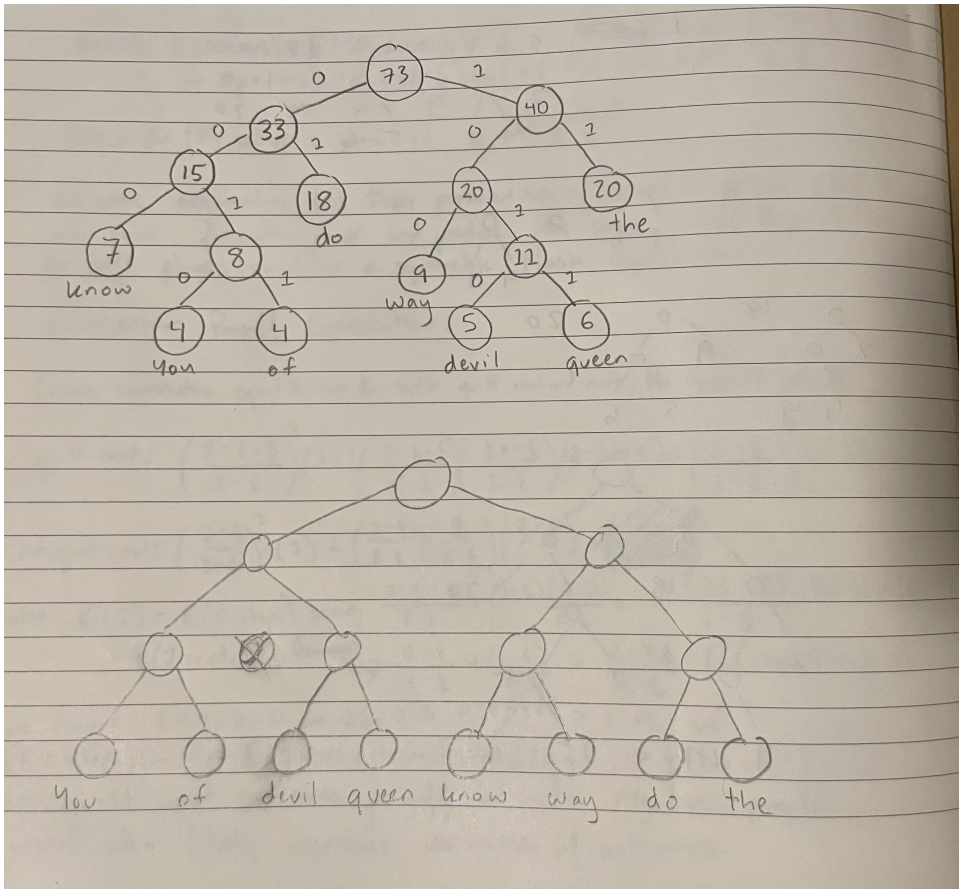
When the words have all the same frequency, then any balanced binary tree will minimize the average representation length and maximize computational efficiency of the hierarchical softmax. But in practice, words occur with very different frequencies — words like "a", "the", and "in" will occur many more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

For the balanced binary tree to store the words of depth 3, each word leaf is stored at depth 3 meaning the averaged representation length is also 3. For the Huffman tree construction we can multiply the representation length by the cumulative frequency at each level and divide that by the sum of all frequencies = 73. We get: $(4 * 19 + 3 * 16 + 2 * 38)/73 = 2.712$.



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

Solution C:

As D increases the computational complexity of the training objective given by formula (1) increases, this should lead to a more complex gradient and thus likely a lower minimization and lower training error. Conceptually, if we have more D latent dimensions we'll have more information to fit on and minimize training error. However this may not be recommended as it can lead to overfitting giving reason to not wanting to use a very large D.

Implementing Word2Vec

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

overview of the particular Word2Vec implementation we'll use.

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

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

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

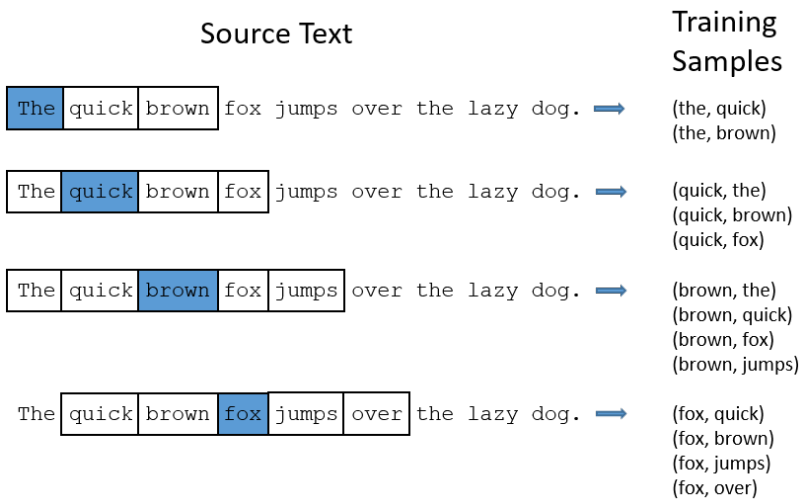


Figure 1: Generating Word2Vec Training Points

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

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

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

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

Implementation

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

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

Solution D:

Colab link to solution

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 dimension of the weight matrix of the hidden layer (size of matrix of weights used for similarity) is* 10×308

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

Solution F: *The dimension of the weight matrix of the output layer is* 308×10

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

Solution G:

```
Pair(took, zans), Similarity: 0.998761
Pair(zans, took), Similarity: 0.998761
Pair(ear, fear), Similarity: 0.99773675
Pair(fear, ear), Similarity: 0.99773675
Pair(cat, never), Similarity: 0.99751425
Pair(never, cat), Similarity: 0.99751425
Pair(cow, hand), Similarity: 0.9974514
Pair(hand, cow), Similarity: 0.9974514
Pair(seven, now), Similarity: 0.99739563
Pair(now, seven), Similarity: 0.99739563
Pair(well, yop), Similarity: 0.99676454
Pair(yop, well), Similarity: 0.99676454
Pair(sticks, play), Similarity: 0.9967127
Pair(play, sticks), Similarity: 0.9967127
Pair(gox, joe), Similarity: 0.99670386
Pair(joe, gox), Similarity: 0.99670386
Pair(yes, cats), Similarity: 0.9963636
Pair(cats, yes), Similarity: 0.9963636
Pair(haircut, yop), Similarity: 0.99627954
Pair(gump, time), Similarity: 0.9961715
Pair(time, gump), Similarity: 0.9961715
Pair(nine, bird), Similarity: 0.9961398
Pair(bird, nine), Similarity: 0.9961398
Pair(five, eight), Similarity: 0.99600697
Pair(eight, five), Similarity: 0.99600697
Pair(wish, let), Similarity: 0.9959919
Pair(let, wish), Similarity: 0.9959919
Pair(hills, yink), Similarity: 0.9959753
Pair(yink, hills), Similarity: 0.9959753
Pair(little, six), Similarity: 0.9959713
```

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

Solution H: *To some degree similar pairs have similar vowel pronunciation (e.g. ear, fear or little, six or yink, hills), additionally numbers appear paired together suggesting frequent counting. Also, the alternate pairs of both words have the same similarity score so they are both ranked, which makes sense as the absolute cosine distance when comparing two vectors does not matter on the order of the vectors. As a result of this pattern, we really only get the top 15 pairs.*