Machine Learning & Data Mining
Set 5

Caltech CS/CNS/EE 155
February 2022

Dallas Taylor

Google Colab Links

- Problem 2: https://colab.research.google.com/drive/1I9AOCtfeBNNbcqvfMhSbecVmRAqAe?usp=sharing

- Problem 3: https://colab.research.google.com/drive/1Vafu3OosFr656bGCgcZnMsJ5ZYnf6tlL?usp=sharing

Dallas Taylor

---

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: *We can demonstrate that the columns of U are the principal components of X by the following:*

Let us first examine the result of XX^T using SVD:

$$\begin{aligned} X &= U\Sigma V^T \\ \therefore XX^T &= U\Sigma V^T (U\Sigma V^T)^T \\ XX^T &= U\Sigma V_T (V\Sigma U^T) \\ XX^T &= U\Sigma I\Sigma U^T \\ XX^T &= U\Sigma^2 U^T \end{aligned}$$

We additionally know (Lecture 10) that the solution to our PCA is $XX^T = U\Lambda U^T$. Thus, we can see that when we have $\Sigma^2 = \Lambda$, our U matrix is the same. Therefore, we can determine that each column of U is an eigenvector of XX^T, which by definition represents the principal component of X.

Thus, we can also determine that the eigenvalues of XX^T are equivalent to the square of singular values of X (from Σ^2).

Problem B [4 points]: Provide both an intuitive explanation and a mathematical justification for why the eigenvalues of the PCA of X (or rather XX^T) are non-negative. Such matrices are called positive semi-definite and possess many other useful properties.

Solution B: *The PCA eigenvalues are non-negative for multiple reasons. The first is since these eigenvalues represent variance (which is intuitively a positive value). We can also see this by considering the relationship between the eigenvalues of XX^T and the singular values of X that was described in Problem A. Since we know that the PCA eigenvalues are equivalent to the squares of our singular values (from Σ^2), resulting in a positive number.*

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, by the hint, first prove that trace is invariant under cyclic permutations for two base case matrices:*

We can clearly see that this holds for $n = 1$, as there are no cyclic permutations.

Base Case of $n = 2$:

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

Case for $n > 2$ (note that we show for letter up to Z for ease of understanding, where this Z really represents the n-th matrix and thus X, Y represent the $n-2$-th and $n-1$-th matrix respectively):

$$\begin{aligned} Tr(ABC...Z) &= \sum_{i=1}^{N}(ABC...Z)_{ii} \\ Tr(ABC...Z) &= \sum_{i=1}^{N}\sum_{j=1}^{N}(ABC...X)_{ij}(YZ)_{ji} \end{aligned}$$

From the above, we can clearly see that we are able to break down any series of matrix multiplication steps into a group of $n = 1$ or $n = 2$ matrices, which we have proven above to hold for our desired identity. Thus, since every possible permutation results in the same equivalent sum, we can determine that trace is invariant under cyclic permutations 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: *We need $N * k * 2 + k$ values (corresponding to N values in each of k columns of U and V^T as well as the k values from Σ) to store a truncated SVD with k singular values. Thus, we find that storing the truncated SVD is more effifcient than storing the whole matrix when:*

$$\begin{aligned} k * 2 * N + k &< N^2 \\ k(2 * N + 1) &< N^2 \\ k &< \frac{N^2}{2N+1} \end{aligned}$$

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: *Let us first show $U\Sigma$, where U is size $D \times D$ and Σ is size $D \times N$, and thus $U\Sigma$ is of size $D \times N$:*

$$U\Sigma = \begin{bmatrix} \sum_{i=1}^{D} U_{1i}\Sigma_{i1} & \ldots & \sum_{i=1}^{D} U_{1i}\Sigma_{iN} \\ & \ldots & \\ \sum_{i=1}^{D} U_{Di}\Sigma_{i1} & \ldots & \sum_{i=1}^{D} U_{Di}\Sigma_{iN} \end{bmatrix}$$

We now know that Σ only has non-zero values on entries $(\Sigma)_{ii}, i \in \{1, ..., N\}$. Thus, we can conclude that $(\Sigma)_{ji} = 0 \forall j > N$. Thus, we can rewrite our matrix $U\Sigma$ as the following (since $N < D$):

$$U\Sigma = \begin{bmatrix} \sum_{i=1}^{N} U_{1i}\Sigma_{i1} & \ldots & \sum_{i=1}^{N} U_{1i}\Sigma_{iN} \\ & \ldots & \\ \sum_{i=1}^{N} U_{Di}\Sigma_{i1} & \ldots & \sum_{i=1}^{N} U_{Di}\Sigma_{iN} \end{bmatrix}$$

We can clearly see from above that we now only operate over the first N columns of U and the first N rows of Σ when computing $U\Sigma$. Thus, it is clear that $U'\Sigma'$ where U' has size $D \times N$ and has the first N columns of U and Σ' has size $N \times N$ and has the first N rows of Σ produces the same matrix as $U\Sigma$, as desired.

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: *We know that U' is of size $D \times N$ and thus, that $(U')^T$ is of size $N \times D$.*

We can find $M_1 = U'(U')^T$ to be of size $D \times D$ with entries $(M_1)_{ij} = \sum_{k=1}^{N} U'_{ik}(U')^T_{kj}$.

We can also find that $M_2 = (U')^TU'$ is of size $N \times N$ with entries $(M_2)_{ij} = \sum_{k=1}^{D}(U')^T_{ik}U'_{kj}$

Thus, according to our class definition, U' cannot be orthogonal as we have clearly not satisifed the case of $AA^T = A^TA$ since the size and entries of M_1 are not equal to the size and entries of M_2.

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 can demonstrate that $(U')^TU' = I_{N\times N}$ by the following:*

We know that U is orthogonal, and thus its columns form an orthonormal basis. Since U' is composed of the first N columns of U, we can conclude that the columns of U' and thus the rows of $(U')^T$ are orthonormal and linearly independent. Thus, when we consider $(U')^TU'$, we have diagonal entries of 1 since the dot product of the same column is 1 and all other entries of 0 since we are taking the dot product across two different linearly independent vectors. We also know that we have $I_{N\times N}$ by Problem F.

However, when we consider the rows of U', we can determine that it is impossible for each to be linearly independent since each row has N entries and there are D rows with $D > N$. Thus, by the definition of dimensionality and orthonormal bases, at least 2 rows within U' must be linearly dependent. Thus, when we calculate $U'(U')^T$, we will not only have non-zero entries in the diagonal, and thus we do not produce $I_{D\times D}$, by definition.

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: *Let us first demonstrate the inverse of Σ. We know that $\Sigma\Sigma^{-1} = I$. Thus, we have:*

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

It is clear that in order to produce the identity matrix, we need to satisfy the following requirements:

1. $\sigma_i * (\Sigma^{-1})_{ii} = 1$

2. $\sigma_i * (\Sigma^{-1})_{ij} = 0 \; \forall j \neq i$

Therefore we can determine that we need to have:

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

The above is clearly equal to Σ^+ as given in class where each diagonal entry $\sigma^+ = \frac{1}{\sigma}$ (or 0 if $\sigma \leq 0$). Thus, we have shown that $X^+ = V\Sigma^{-1}U^T$, as desired.

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 $V\Sigma^{-1}U^T$.

Solution I: *Let us break down our expression in terms of our SVD. Note that V is orthogonal and thus $V^T = V^{-1}$ and we have shown that $U^TU = I$ in Problem G:*

$$\begin{aligned}
(X^TX)^{-1}X^T &= \left((U\Sigma V^T)^T(U\Sigma V^T)\right)^{-1}(U\Sigma V^T)^T \\
&= \left((V\Sigma U^T)(U\Sigma V^T)\right)^{-1}(V\Sigma U^T) \\
&= \left(V\Sigma(I)\Sigma V^T\right)^{-1}(V\Sigma U^T) \\
&= \left(V\Sigma^2 V^T\right)^{-1}(V\Sigma U^T) \\
&= (V^T)^{-1}(\Sigma^2)^{-1}V^{-1}(V\Sigma U^T) \\
&= (V^T)^{-1}(\Sigma^2)^{-1}(I)\Sigma U^T \\
&= V(\Sigma^2)^{-1}\Sigma U^T \\
&= V\Sigma^{-1}U^T
\end{aligned}$$

We are able to simplify to our final line since Σ^2 produces the matrix with diagonal entries of σ_i^2 and thus $(\Sigma^2)^{-1}$ produces the matrix with diagonal entries of $\frac{1}{\sigma_i^2}$, and finally $(\Sigma^2)^{-1}\Sigma$ produces the matrix with diagonal entries $\frac{1}{\sigma_i}$ (equivalent to Σ^{-1}).

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: *Let us first calculate the condition number of Σ:*

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

Now, let us consider the condition number of X^TX:

$$\begin{aligned}
\kappa(X^TX) &= \frac{\sigma_{max}(X^TX)}{\sigma_{min}(X^TX)} \\
&= \frac{\sigma_{max}\left((U\Sigma V^T)^TU\Sigma V^T\right)}{\sigma_{min}\left((U\Sigma V^T)^TU\Sigma V^T\right)} \\
&= \frac{\sigma_{max}\left((V\Sigma U^T)U\Sigma V^T\right)}{\sigma_{min}\left((V\Sigma U^T)U\Sigma V^T\right)} \\
&= \frac{\sigma_{max}\left(V\Sigma^2V^T\right)}{\sigma_{min}\left(V\Sigma^2V^T\right)} \\
&= \frac{(\sigma_{max}(\Sigma))^2}{(\sigma_{min}(\Sigma))^2}
\end{aligned}$$

Since we obviously know that $\sigma_{max} > \sigma_{min}$, we can determine that $\kappa(X^TX) > \kappa(\Sigma)$ and thus, calculations when considering the inverse of X^TX are more error prone than those considering the inverse of Σ. Thus, computations in Part I is more prone to numerical errors.

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: *Let us first consider the gradient with respect to u_i:*

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

Now, let us consider the gradient with respect to v_j:

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

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: *Let us first consider the optimal u_i:*

$$\begin{aligned}
\partial_{u_i} &= 0 \\
\lambda u_i - \sum_j v_j(y_{ij} - u_i^T v_j) &= 0 \\
\lambda u_i &= \sum_j v_j(y_{ij} - u_i^T v_j) \\
\lambda u_i &= \sum_j v_j y_{ij} - v_j u_i^T v_j \\
\lambda u_i &= \sum_j v_j y_{ij} - \sum_j v_j u_i^T v_j \\
\lambda u_i + \sum_j v_j u_i^T v_j &= \sum_j v_j y_{ij} \\
\lambda u_i + u_i \sum_j v_j v_j^T &= \sum_j v_j y_{ij} \\
u_i(\lambda(I) + \sum_j v_j v_j^T) &= \sum_j v_j y_{ij} \\
u_i &= (\sum_j v_j y_{ij})\big(\lambda(I) + \sum_j v_j v_j^T\big)^{-1}
\end{aligned}$$

Since we have that ∂_{v_j} is essentially the same as ∂_{u_i} (except interchanging the first u_i value with the first v_j value and the variable in the sum). Thus, it is clear we can follow the same process to produce the optimal v_j:

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

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.

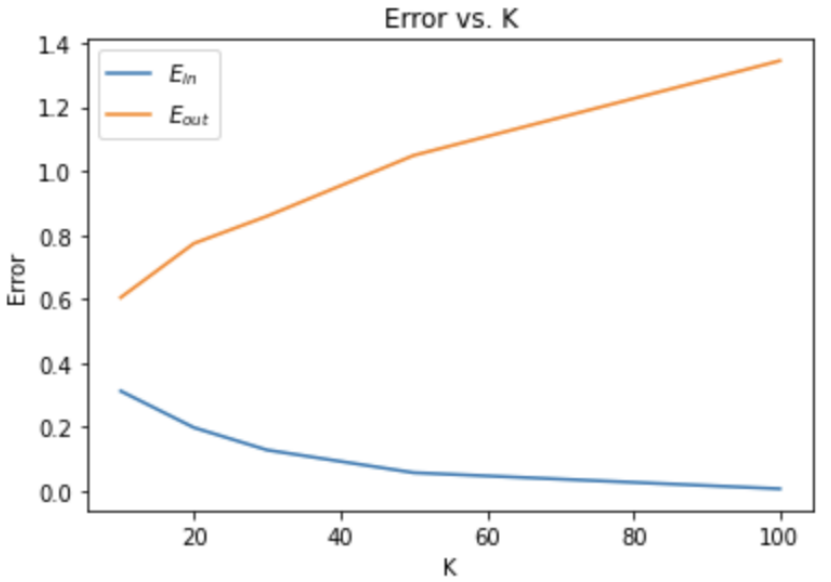
In your implementation, you should:

- 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 :* https://colab.research.google.com/drive/1I9AOCtfeBNNbcqvfMhSbecVmRAqAe?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, 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?

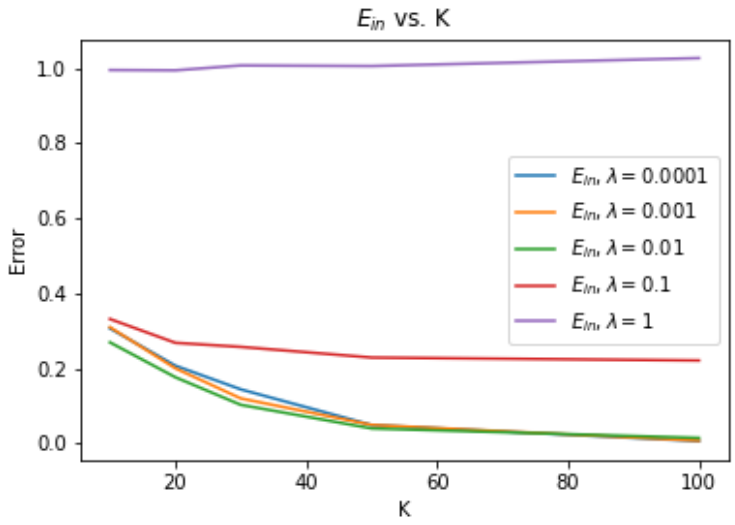
Solution D:

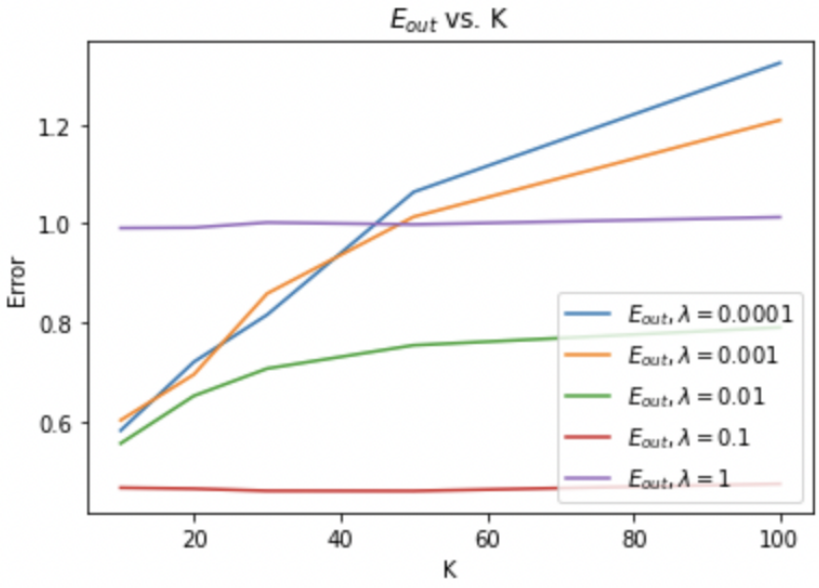


We can see that as k increases, E_{out} increases whereas E_{in} decreases. This can be explained by the fact that our model overfits the training data more for every increase in K, thus decreasing our training error (E_{in}) and increasing our testing error (E_{out}). This overfitting is consequence of the increase of values in each row of both our user and movie matrices U, V, and thus allow for more precise operations on the training set while sacrificing generality needed to perform well on the testing set.

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.

Solution E:





First, let us consider E_{in}. We can see from the above that $\lambda \in \{0.0001, 0.001, 0.01\}$ produces very similar trajectories with continually decreasing (approaching 0) error as k increases. For $\lambda = 0.1$, we still see decreasing behavior as k increases, but it is much flatter and only approaches about 0.2. For $\lambda = 1$, we see that our training error holds steady around a value of $1.0 \forall k$.

Now, let us consider E_{out}. We can see from the above that $\lambda \in \{0.0001, 0.001\}$ produces similar trajectories

with continually increasing error as k increases. When $\lambda = 0.01$, we have a flatter, but still increasing, trajectory than the lower values. When $\lambda = 0.1$, our testing error holds steady around a value of $0.5 \forall k$ and when $\lambda = 1$, our testing error holds steady around a value of $1.0 \forall k$.

These observations make sense considering that in our previous plots, we observed a general upwards trend of E_{out} and a general downwards trend of E_{in}. Thus, most of our E_{out} trajectories are generally increasing and most of our E_{in} trajectories are generally decreasing, with our regularization term effecting the steepness of such trajectories (until we hit $\lambda = 1$, which does not allow for learning).

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: *Let us first examine the time complexity of calculating the gradient for a single w_O, w_I pair. Thus, let us demonstrate what our gradients become:*

$$\begin{aligned}
\partial_{v'_{w_O}} &= \partial_{v'_{w_O}} \left(\log p(w_{t+j} \mid w_t)\right) \\
&= \partial_{v'_{w_O}} \left(\log exp(v'^T_{w_O} v_{w_I}) - \log \sum_{w=1}^{W} exp(v'^T_w v_{w_I})\right) \\
&= \partial_{v'_{w_O}} (v'^T_{w_O} v_{w_I}) - \frac{1}{\sum_{w=1}^{W} exp(v'^T_w v_{w_I})} \cdot \partial_{v'_{w_O}} \left(\sum_{w=1}^{W} exp(v'^T_w v_{w_I})\right) \\
&= v_{w_I} - \frac{v_{w_I} exp(v'^T_{w_O} v_{w_I})}{\sum_{w=1}^{W} exp(v'^T_w v_{w_I})}
\end{aligned}$$

By the same process, we can also produce:

$$\partial_{v_{w_I}} = v'_{w_O} - \frac{\sum_{w=1}^{W} v'_w exp(v'^T_w v_{w_I})}{\sum_{w=1}^{W} exp(v'^T_w v_{w_I})}$$

When computing calculations for each pair, it is clear that our time complexity is dominated by the numerator of $\partial_{v_{w_I}}$. In this calculation we are iterating over W words, where we compute the dot product of v'^T_w, v_{w_I} and multiply that result by v'_w again. Thus, for W words, we operate over two vectors of dimensions D, and multiply the result by a vector of dimension D. Thus, our time complexity for a single w_O, w_I pair is $\mathcal{O}(W * D)$. It is clear from here, that when considering all possible w_O, w_I pairs that there are $W \times W$ possibilities and thus our total time complexity is $\mathcal{O}(W * W * W * D) = \mathcal{O}(W^3 D)$.

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".

Table 1: Words and frequencies for Problem B

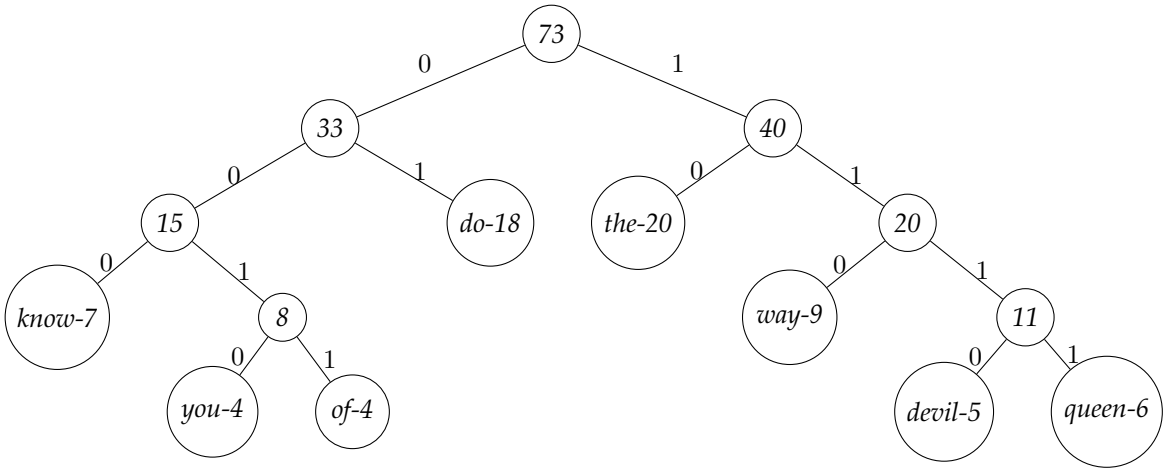
Word	Occurrences
do	18
you	4
know	7
the	20
way	9
of	4
devil	5
queen	6

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.

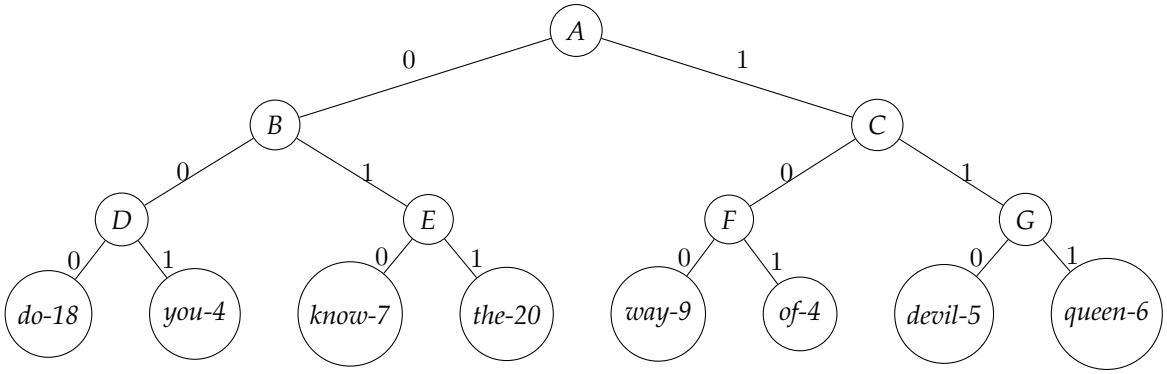
The representation length of a word is then the length of the path (the number of edges) from the root to the leaf node corresponding to the word. For each tree you constructed, compute the expected representation length (averaged over the actual frequencies of the words).

Solution B:

Below is our representation for the Huffman tree:



Below is our representation of a balanced binary tree of depth 3:



It is clear from the above that our Huffman tree has an average path length of

$$\frac{3*7+4*4+4*4+2*18+2*20+3*9+4*5+4*6}{18+4+7+20+9+4+5+6} = \frac{200}{73} = 2.740$$

and our balanced binary tree has an average path length of 3 *(since every path length is* 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: *I expect that the value of the training objective will increase as D increases, since more dimensions could lead to more accurate representations and predictions of the training data. Thus, we would not want to use a very large D in order to avoid consequential overfitting of the training data and to optimize the speed of our model by avoiding additional computational cost.*

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 $s = 2$:

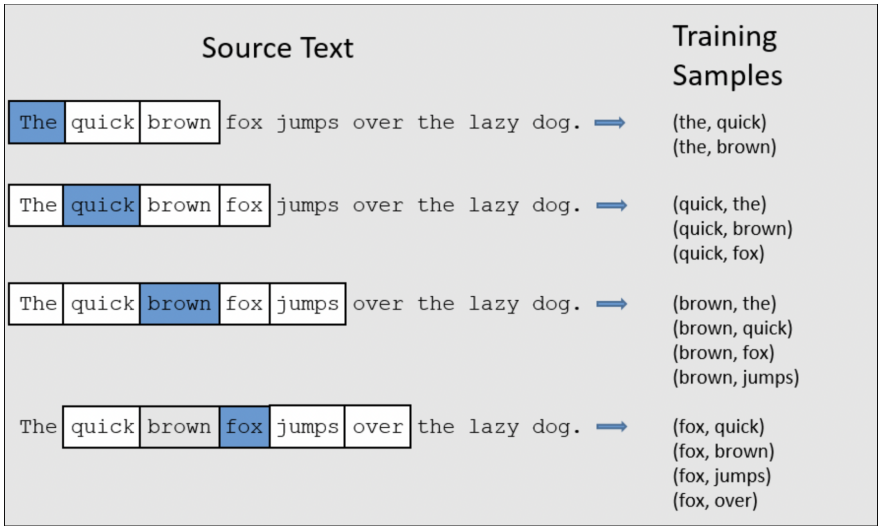


Figure 1: Generating Word2Vec Training Points

(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:

Colab Link : https://colab.research.google.com/drive/1Vafu3OosFr656bGCgcZnMsJ

5ZYnf6tlL?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 weight matrix of my hidden layer is of dimension* (308×10)*, correlating to* 308 *rows to operate on* 308 *words and* 10 *columns for the number of latent factors.*

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

Solution F: *The weight matrix of my output layer is of dimension* (10×308)*, correlating to* 10 *rows to operate on the* 10 *latent factors and* 308 *columns for the conversion back to vocab size.*

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

Solution G:

```
Textfile contains 308 unique words

Weight Dimensions of Hidden Layer:  torch.Size([308, 10])
Weight Dimensions of Output Layer:  torch.Size([10, 308])

Pair(more, eight), Similarity: 0.984567
Pair(eight, more), Similarity: 0.984567
Pair(five, took), Similarity: 0.9807174
Pair(took, five), Similarity: 0.9807174
Pair(hills, girls), Similarity: 0.9714212
Pair(girls, hills), Similarity: 0.9714212
Pair(wave, tomorrow), Similarity: 0.96945786
Pair(tomorrow, wave), Similarity: 0.96945786
Pair(sad, work), Similarity: 0.9672082
Pair(work, sad), Similarity: 0.9672082
Pair(eggs, ham), Similarity: 0.965816
Pair(ham, eggs), Similarity: 0.965816
Pair(jump, cats), Similarity: 0.96065414
Pair(cats, jump), Similarity: 0.96065414
Pair(quiet, joe), Similarity: 0.96053225
Pair(joe, quiet), Similarity: 0.96053225
Pair(fingers, pull), Similarity: 0.95847815
Pair(pull, fingers), Similarity: 0.95847815
Pair(bike, hair), Similarity: 0.95805043
Pair(hair, bike), Similarity: 0.95805043
Pair(green, eggs), Similarity: 0.9551929
Pair(upon, grows), Similarity: 0.9511192
Pair(grows, upon), Similarity: 0.9511192
Pair(nine, eight), Similarity: 0.9501251
Pair(anything, five), Similarity: 0.9489164
Pair(us, must), Similarity: 0.9477003
Pair(must, us), Similarity: 0.9477003
Pair(ten, more), Similarity: 0.9470631
Pair(sticks, cat), Similarity: 0.946355
Pair(cat, sticks), Similarity: 0.946355
```

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

Solution H: *I notice that the resulting pairs of words often appear twice, as the complement of themselves but with the same similarity value. I also notice that a lot of the words make sense together in context, such as took/five, eight/more, or nine/eight. It can additionally be seen that some word pairs contain words that are common pairings in Dr. Seuss's work such as eggs/ham and green/eggs.*