

CSC413A1

siweitang

February 9, 2023

1 Linear Regression (2.5pts)

The reading on linear regression located at <https://uoft-csc413.github.io/2023/assets/readings/L01a.pdf> may be useful for this question.

Given n pairs of input data with d features and scalar label $(\mathbf{x}_i, t_i) \in \mathbb{R}^d \times \mathbb{R}$, we wish to find a linear model $f(\mathbf{x}) = \hat{\mathbf{w}}^\top \mathbf{x}$ with $\hat{\mathbf{w}} \in \mathbb{R}^d$ that minimizes the squared error of prediction on the training samples defined below. This is known as an empirical risk minimizer. For concise notation, denote the data matrix $X \in \mathbb{R}^{n \times d}$ and the corresponding label vector $\mathbf{t} \in \mathbb{R}^n$. The training objective is to minimize the following loss:

$$\min_{\hat{\mathbf{w}}} \frac{1}{n} \sum_{i=1}^n (\hat{\mathbf{w}}^\top \mathbf{x}_i - t_i)^2 = \min_{\hat{\mathbf{w}}} \frac{1}{n} \|X\hat{\mathbf{w}} - \mathbf{t}\|_2^2.$$

We assume X is full rank: $X^\top X$ is invertible when $n > d$, and XX^\top is invertible otherwise. Note that when $d > n$, the problem is *underdetermined*, i.e. there are less training samples than parameters to be learned. This is analogous to learning an *overparameterized* model, which is common when training of deep neural networks.

a Deriving the Gradient [0pt]

Write down the gradient of the loss w.r.t. the learned parameter vector $\hat{\mathbf{w}}$.

b Underparameterized Model

i [0.5pt]

First consider the underparameterized $d < n$ case. Show that the solution obtained by gradient descent is $\hat{\mathbf{w}} = (X^\top X)^{-1} X^\top \mathbf{t}$, assuming training converges. Show your work.

$$\begin{aligned} \frac{2}{n}(X^\top X \hat{\mathbf{w}} - X^\top \mathbf{t}) &= 0 \\ X^\top X \hat{\mathbf{w}} - X^\top \mathbf{t} &= 0 \\ \hat{\mathbf{w}} &= (X^\top X)^{-1} X^\top \mathbf{t} \end{aligned}$$

chatGPT:

s

First consider the underparameterized $d < n$ case. Show that the solution obtained by gradient descent is $\hat{w} = (X^T X)^{-1} X^T t$, assuming training converges. Show your work.



Let's start by considering the cost function in linear regression:



$$J(w) = \frac{1}{2m} \sum_{i=1}^m (h(x^{(i)}) - y^{(i)})^2$$

Where $h(x)$ is the hypothesis function given by $h(x) = Xw$.

In gradient descent, we want to minimize $J(w)$ by iteratively updating w as follows:

$$w = w - \alpha \nabla J(w)$$

Where α is the learning rate and $\nabla J(w)$ is the gradient of $J(w)$ with respect to w .

Calculating the gradient of $J(w)$, we get:

$$\nabla J(w) = \frac{1}{m} \sum_{i=1}^m (h(x^{(i)}) - y^{(i)}) X^{(i)}$$

Now, let's assume that the training converges, meaning that at the end of the iteration process, the cost function $J(w)$ is minimized. This means that the gradient $\nabla J(w)$ is equal to zero.

Setting $\nabla J(w) = 0$, we get:

$$0 = \frac{1}{m} \sum_{i=1}^m (h(x^{(i)}) - y^{(i)}) X^{(i)}$$

Expanding the hypothesis function $h(x)$, we get:

$$0 = 1/m \sum_{i=1}^m (Xw - y^i) X^i$$

Now, we can rewrite the above equation as:

$$0 = X^T X w - X^T t$$

Finally, solving for w , we get:

$$w^* = (X^T X)^{-1} X^T t$$

This means that the solution obtained by gradient descent is the same as the normal equation solution when the number of features (n) is greater than the number of training examples (m).

ii [0.5pt]

Now consider the case of noisy linear regression. The training labels $t_i = \mathbf{w}^{*T} \mathbf{x}_i + \epsilon_i$ are generated by a ground truth linear target function, where the noise term, ϵ_i , is generated independently with zero mean and variance σ^2 . The final training error can be derived as a function of X and ϵ , as:

$$\text{Error} = \frac{1}{n} \| (X(X^T X)^{-1} X^T - I) \epsilon \|_2^2,$$

Show this is true by substituting your answer from the previous question into $\frac{1}{n} \| X \hat{w} - t \|_2^2$. Also, find the expectation of the above training error in terms of n, d and σ .

Hints: you might find the cyclic property¹ of trace useful.

$$\begin{aligned} \frac{1}{n} \| X \hat{w} - t \|_2^2 &= \frac{1}{n} \| X(X^T X)^{-1} X^T t - t \|_2^2 \\ t_i &= w *^T x_i + \epsilon_i \Rightarrow t = Xw^* + \epsilon \\ \frac{1}{n} \| X(X^T X)^{-1} X^T t - t \|_2^2 &= \frac{1}{n} \| X(X^T X)^{-1} X^T (Xw^* + \epsilon) - (Xw^* + \epsilon) \|_2^2 \\ &= \frac{1}{n} \| Xw^* + X(X^T X)^{-1} X^T \epsilon - Xw^* - \epsilon \|_2^2 \\ &= \frac{1}{n} \| (X(X^T X)^{-1} X^T - I) \epsilon \|_2^2 \\ E[\text{Error}] &= E[\text{tr}(\frac{1}{n} \| (X(X^T X)^{-1} X^T - I) \epsilon \|_2^2)] \\ &= \frac{1}{n} E[\text{tr}(((X(X^T X)^{-1} X^T - I) \epsilon)^T ((X(X^T X)^{-1} X^T - I) \epsilon))] \\ &= \frac{1}{n} E[\text{tr}(\epsilon^T \epsilon - \epsilon^T X(X^T X)^{-1} X^T \epsilon)] \\ &= \frac{1}{n} \text{tr}(E[\epsilon^T \epsilon]) - \frac{1}{n} \text{tr}(E[\epsilon^T X(X^T X)^{-1} X^T \epsilon]) \\ E(\epsilon^T \epsilon) &= E\left(\sum_{i=1}^n \epsilon_i^2\right) = n\sigma^2, E(\epsilon \epsilon^T) = \sigma^2 I_n \\ E[\text{Error}] &= \frac{1}{n} (n\sigma^2) - \frac{1}{n} \text{tr}(E[\epsilon \epsilon^T X(X^T X)^{-1} X^T]) = \sigma^2 - \frac{1}{n} \text{tr}(E[\epsilon \epsilon^T X(X^T X)^{-1} X^T]) \\ &= \sigma^2 - \frac{\sigma^2}{n} \text{tr}(X^T X(X^T X)^{-1}) = \sigma^2 - \frac{\sigma^2}{n} d \end{aligned}$$

chatGPT:

¹ [https://en.wikipedia.org/wiki/Trace_\(linear_algebra\)#Cyclic_property](https://en.wikipedia.org/wiki/Trace_(linear_algebra)#Cyclic_property)

Now consider the case of noisy linear regression. The training labels $t_i = w^T x_i + \epsilon_i$ are generated by a ground truth linear target function, where the noise term, ϵ_i , is generated independently with zero mean and variance σ^2 . The final training error can be derived as a function of X and ϵ , as: Error = $1/n \| (X(X^T X) - 1X^T - I)\epsilon \|_F^2$. Show this is true by substituting your answer from the previous question into $1/n \| Xw^* - t \|_F^2$. Also, find the expectation of the above training error in terms of n , d and σ .

The final training error can be derived as a function of X and ϵ , as Error = $1/n \| (X(X^T X) - 1X^T - I)\epsilon \|_F^2$. To show this, we can substitute $w^* = (X^T X)^{-1} X^T t$ into $1/n \| Xw^* - t \|_F^2$:

$$\begin{aligned}\text{Error} &= 1/n \| Xw^* - t \|_F^2 \\ &= 1/n \| X(X^T X) - 1X^T t - t \|_F^2 \\ &= 1/n \| (X(X^T X) - 1X^T - I)t + \epsilon \|_F^2 \\ &= 1/n \| (X(X^T X) - 1X^T - I)\epsilon \|_F^2\end{aligned}$$

The expectation of the training error in terms of n , d , and σ can be found by taking the expectation of the above equation:

The expectation of the training error in terms of n , d , and σ can be found by taking the expectation of the above equation:

$$\begin{aligned}E[\text{Error}] &= 1/n E[\| (X(X^T X) - 1X^T - I)\epsilon \|_F^2] \\ &= 1/n \| (X(X^T X) - 1X^T - I) \|_F^2 \sigma^2 \\ &= 1/n (\text{tr}((X(X^T X) - 1X^T - I)(X(X^T X) - 1X^T - I)^T)) \sigma^2 \\ &= 1/n (\text{tr}(I) - 2\text{tr}(X^T X(X^T X) - 1) + \text{tr}((X^T X)^{-1} X^T X(X^T X) - 1)) \sigma^2 \\ &= 1/n ((n - d) + d - d) \sigma^2 \\ &= 1/n (n - d) \sigma^2\end{aligned}$$

Thus, the expectation of the training error in terms of n , d , and σ is $E[\text{Error}] = 1/n (n - d) \sigma^2$.

c Overparameterized Model

i [0.5pt]

Now consider the overparameterized $d > n$ case. We first illustrate that there exist multiple empirical risk minimizers. For simplicity we let $n = 1$ and $d = 2$. Choose $\mathbf{x}_1 = [1; 1]$ and $t_1 = 3$, i.e. the one data point and all possible $\hat{\mathbf{w}}$ lie on a 2D plane. Show that there exists infinitely many $\hat{\mathbf{w}}$ satisfying $\hat{\mathbf{w}}^\top \mathbf{x}_1 = y_1$ on a real line. Write down the equation of the line.

$$[\hat{w}_1^T \quad \hat{w}_2^T] \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = 3$$

since \hat{w}_1^T and \hat{w}_2^T are scalar $\Rightarrow \hat{w}_1 + \hat{w}_2 = 3$

chatGPT:

s

Consider the overparameterized $d > n$ case. We first illustrate that there exist multiple empirical risk minimizers. For simplicity we let $n = 1$ and $d = 2$. Choose $x_1 = [1; 1]$ and $t_1 = 3$, for example one data point and all possible w^* lie on a 2D plane. Show that there exists infinitely many w^* satisfying $w^* \top x_1 = y_1$ on a real line. Write down the equation of the line.



In the overparameterized case where $d > n$, there can exist multiple empirical risk minimizers.

Let's consider the simple case where $n = 1$ and $d = 2$. If we have one data point $x1 = [1; 1]$ and $t1 = 3$, then the possible w^* values lie on a 2D plane.

Since $w^* \top x1 = t1$, we can write down the equation of the line as:

$$w_2 = t1 - w_1$$

This line represents all possible w^* values that satisfy the constraint $w^* \top x1 = t1$. Since there is an infinite number of points on this line, there are infinitely many w^* that satisfy the constraint, thus showing that there exist multiple empirical risk minimizers in the overparameterized case.

ii [0.5pt]

Now, let's generalize the previous 2D case to the general $d > n$. Show that gradient descent from zero initialization i.e. $\hat{\mathbf{w}}(0) = 0$ finds a unique minimizer if it converges. Show that the solution by gradient decent is $\hat{\mathbf{w}} = X^\top (X X^\top)^{-1} \mathbf{t}$. Show your work.

Hints: You can assume that the gradient is spanned by the rows of X and write $\hat{\mathbf{w}} = X^\top \mathbf{a}$ for some $\mathbf{a} \in \mathbb{R}^n$.

$$\begin{aligned}\hat{w} &= X^T a \\ \frac{\partial l}{\partial \hat{w}} &= \frac{2}{n} (X^T X \hat{w} - X^T t) = 0 \\ X^T (X \hat{w} - t) &= 0\end{aligned}$$

By assumption, X is full rank, that means X cannot be zero

$$X \hat{w} - t = X X^T a - t = 0$$

$$a = (X X^T)^{-1} t$$

chatGPT:

Let's generalize the 2D case to the general $d > n$. Show that gradient descent from zero initialization i.e. $w^*(0) = 0$ finds a unique minimizer if it converges. Show that the solution by gradient descent is $w^* = (X^T(XX^T)^{-1})t$. Show your work.

Hints: You can assume that the gradient is spanned by the rows of X and write $w^* = Xta$ for some $a \in \mathbb{R}^n$

In the overparameterized $d > n$ case, the solution obtained by gradient descent from zero initialization, i.e., $w^*(0) = 0$, finds a unique minimizer if it converges. We can assume that the gradient is spanned by the rows of X and write $w^* = Xta$ for some $a \in \mathbb{R}^n$.

The objective function to be minimized is the mean squared error:

$$J(w) = 1/n \|Xw - t\|^2 = 1/n (Xw - t)^T (Xw - t)$$

Taking the gradient with respect to w and setting it to zero, we have:

$$\nabla J(w) = 2X^T(Xw - t) = 0$$

Solving for w , we have:

$$X^T X w = X^T t$$

Rearranging the above equation, we get:

$$w = (X^T X)^{-1} X^T t$$

Thus, the solution by gradient descent is $w^* = (X^T X)^{-1} X^T t$. This solution is unique if $X^T X$ is invertible, which is the case for a full-rank design matrix X .

iii [0pt]

Repeat part ii for the overparameterized case.

iv [0.5pt]

Visualize and compare underparameterized with overparameterized polynomial regression:

https://colab.research.google.com/github/uoft-csc413/2023/blob/master/assets/assignments/LS_poly_regression.ipynb

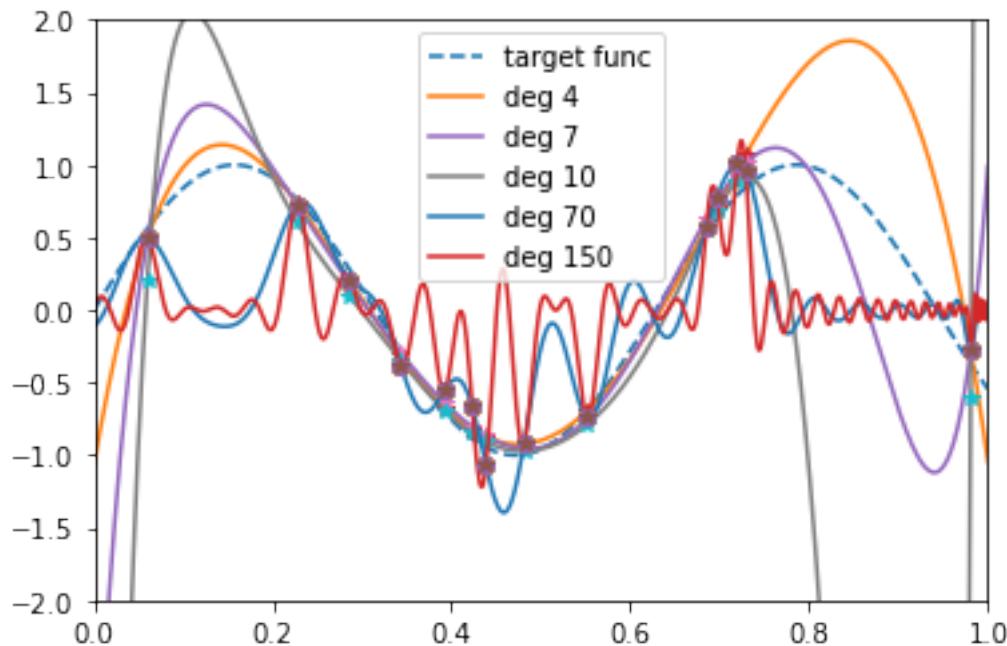
Include your code snippets for the `fit_poly` function in the write-up.

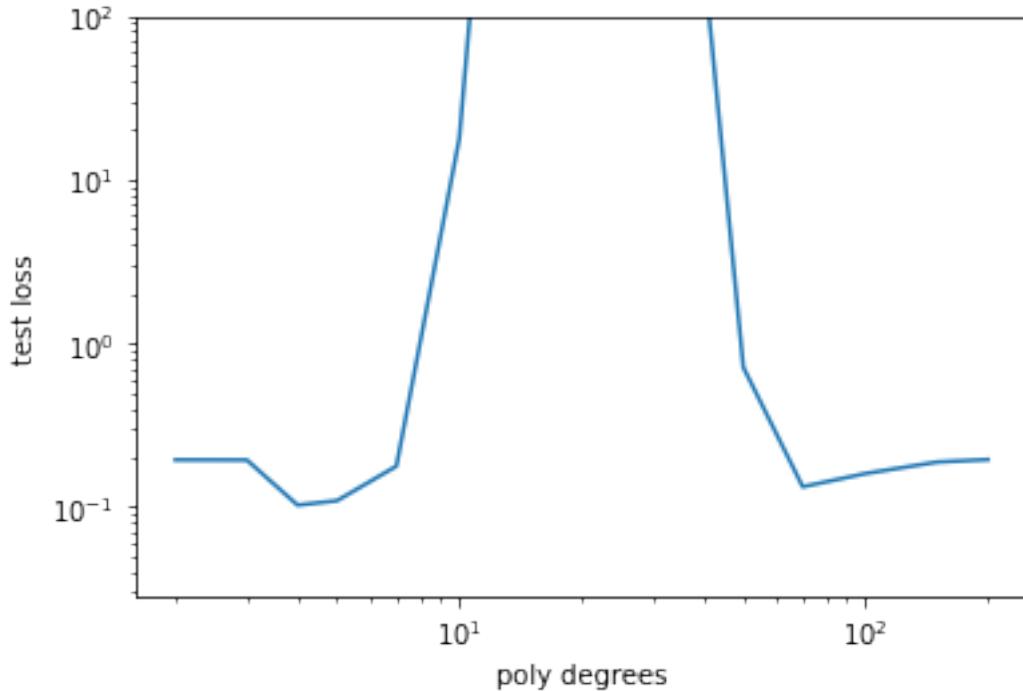
Does overparameterization (higher degree polynomial) always lead to overfitting, i.e. larger test error?

Overparameterization may not cause error. We can see that from the test loss graph. When poly degrees smaller than 10 and larger than 100, the loss is small.

```
▶ # to be implemented; fill in the derived solution for the underparameterized (d<n) and overparameterized (d>n) problem

def fit_poly(X, d,t):
    X_expand = poly_expand(X, d=d, poly_type = poly_type)
    n = X.shape[0]
    if d > n:
        W = np.matmul(np.matmul(np.transpose(X_expand), np.linalg.inv(np.matmul(X_expand, np.transpose(X_expand)))), t)
    else:
        W = np.matmul(np.matmul(np.linalg.inv(np.matmul(np.transpose(X_expand), X_expand)), np.transpose(X_expand)), t)
    return W
```





v [0pt]

Give n_1, n_2 with $n_1 \leq n_2$, and fixed dimension d for which $L_2 \geq L_1$, i.e. the loss with n_2 data points is greater than loss with n_1 data points. Explain the underlying phenomenon. Be sure to also include the error values L_1 and L_2 or provide visualization in your solution.

Hints: use your code to experiment with relevant parameters, then vary to find region and report one such setting.

2 Backpropagation (4pts)

This question helps you to understand the underlying mechanism of back-propagation. You need to have a clear understanding of what happens during the forward pass and backward pass and be able to reason about the time complexity and space complexity of your neural network. Moreover, you will learn a commonly used trick to compute the gradient norm efficiently without explicitly writing down the whole Jacobian matrix.

Note: The reading on backpropagation located at <https://uoft-csc413.github.io/2023/assets/readings/L02b.pdf> may be useful for this question.

a Automatic Differentiation

Consider a neural network defined with the following procedure:

$$\begin{aligned}
\mathbf{z}_1 &= \mathbf{W}^{(1)} \mathbf{x} + \mathbf{b}^{(1)} \\
\mathbf{h}_1 &= \text{ReLU}(\mathbf{z}_1) \\
\mathbf{z}_2 &= \mathbf{W}^{(2)} \mathbf{x} + \mathbf{b}^{(2)} \\
\mathbf{h}_2 &= \sigma(\mathbf{z}_2) \\
\mathbf{g} &= \mathbf{h}_1 \circ \mathbf{h}_2 \\
\mathbf{y} &= \mathbf{W}^{(3)} \mathbf{g} + \mathbf{W}^{(4)} \mathbf{x}, \\
\mathbf{y}' &= \text{softmax}(\mathbf{y}) \\
\mathcal{S} &= \sum_{k=1}^N (t = k) \log(\mathbf{y}'_k) \\
\mathcal{J} &= -\mathcal{S}
\end{aligned}$$

for input \mathbf{x} with class label t where $\text{ReLU}(\mathbf{z}) = \max(\mathbf{z}, 0)$ denotes the ReLU activation function, $\sigma(\mathbf{z}) = \frac{1}{1+e^{-\mathbf{z}}}$ denotes the Sigmoid activation function, both applied elementwise, and $\text{softmax}(\mathbf{y}) = \frac{\exp(\mathbf{y})}{\sum_{i=1}^N \exp(\mathbf{y}_i)}$. Here, \circ denotes element-wise multiplication.

i Computational Graph [0pt]

Draw the computation graph relating \mathbf{x} , t , \mathbf{z}_1 , \mathbf{z}_2 , \mathbf{h}_1 , \mathbf{h}_2 , \mathbf{g} , \mathbf{y} , \mathbf{y}' , \mathcal{S} and \mathcal{J} .

ii Backward Pass blue[1pt]

Derive the backprop equations for computing $\bar{\mathbf{x}} = \frac{\partial \mathcal{J}^\top}{\partial \mathbf{x}}$, one variable at a time, similar to the vectorized backward pass derived in Lec 2.

Hints: Be careful about the transpose and shape! Assume all vectors (including error vector) are column vector and all Jacobian matrices adopt numerator-layout notation². You can use $\text{softmax}'(\mathbf{y})$ for the Jacobian matrix of softmax.

$$\begin{aligned}
\bar{\mathcal{J}} &= 1, \bar{\mathcal{S}} = -1, \bar{\mathbf{y}'} = \bar{\mathcal{S}} \frac{\partial \mathcal{S}}{\partial \mathbf{y}'}^\top = \begin{bmatrix} 0 & \dots & -\frac{1}{y'_k} & \dots & 0 \end{bmatrix}^\top \\
\bar{\mathbf{y}} &= \bar{\mathbf{y}'} \frac{\partial \mathbf{y}'}{\partial \mathbf{y}}^\top = \text{softmax}'(\mathbf{y})^\top \bar{\mathbf{y}'}, \bar{\mathbf{g}} = \bar{\mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{g}}^\top = \mathbf{W}^{(3)\top} \bar{\mathbf{y}} \\
\bar{\mathbf{h}}_1 &= \mathbf{h}_2 \circ \bar{\mathbf{g}}, \bar{\mathbf{h}}_2 = \mathbf{h}_1 \circ \bar{\mathbf{g}} \\
\text{ReLU}'(\mathbf{x}) &= \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} (1), \bar{\mathbf{z}}_1 = \bar{\mathbf{h}}_1 \circ \text{ReLU}'(\mathbf{z}_1) \\
\bar{\mathbf{z}}_2 &= \bar{\mathbf{h}}_2 \circ \sigma'(\mathbf{z}_2) \\
\bar{\mathbf{x}} &= \mathbf{W}^{(1)\top} \bar{\mathbf{z}}_1 + \mathbf{W}^{(2)\top} \bar{\mathbf{z}}_2 + \mathbf{W}^{(4)\top} \bar{\mathbf{y}}
\end{aligned}$$

b Gradient Norm Computation

Many deep learning algorithms require you to compute the L^2 norm of the gradient of a loss function with respect to the model parameters for every example in a minibatch. Unfortunately, most differentiation functionality provided by most software frameworks (Tensorflow, PyTorch) does not support computing gradients for individual samples in a minibatch. Instead, they only give one gradient per minibatch that aggregates individual gradients for you. A naive way to get the per-example gradient norm is to use a batch size of 1 and repeat the back-propagation N times, where N is the minibatch size. After that, you can compute the L^2 norm of each gradient vector. As you can imagine, this approach is very inefficient. It can not exploit the

²Numerator-layout notation: https://en.wikipedia.org/wiki/Matrix_calculus#Numerator_layout_notation

parallelism of minibatch operations provided by the framework.

In this question, we will investigate a more efficient way to compute the per-example gradient norm and reason about its complexity compared to the naive method. For simplicity, let us consider the following two-layer neural network.

$$\begin{aligned}\mathbf{z} &= \mathbf{W}^{(1)}\mathbf{x} \\ \mathbf{h} &= \text{ReLU}(\mathbf{z}) \\ \mathbf{y} &= \mathbf{W}^{(2)}\mathbf{h},\end{aligned}$$

$$\text{where } \mathbf{W}^{(1)} = \begin{pmatrix} 1 & 2 & 1 \\ -2 & 1 & 0 \\ 1 & -2 & -1 \end{pmatrix} \text{ and } \mathbf{W}^{(2)} = \begin{pmatrix} -2 & 4 & 1 \\ 1 & -2 & -3 \\ -3 & 4 & 6 \end{pmatrix}.$$

i Naive Computation [1pt]

Let us assume the input $\mathbf{x} = (1 \ 3 \ 1)^\top$ and the error vector $\bar{\mathbf{y}} = \frac{\partial \mathcal{J}}{\partial \mathbf{y}}^\top = (1 \ 1 \ 1)^\top$. In this question, write down the Jacobian matrix (numerical value) $\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}}$ and $\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(2)}}$ using back-propagation. Then, compute the square of Frobenius Norm of the two Jacobian matrices, $\|A\|_F^2$. The square of Frobenius norm of a matrix A is defined as follows:

$$\|A\|_F^2 = \sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2 = \text{trace}(A^\top A)$$

Hints: Be careful about the transpose. Show all your work for partial marks.

$$\begin{aligned}x &= (1 \ 3 \ 1)^\top, \mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} = (8 \ 1 \ -6)^\top, \text{ReLU}'(\mathbf{z}) = (1 \ 1 \ 0)^\top, \bar{\mathbf{h}} = \mathbf{W}^{(2)^\top} \bar{\mathbf{y}} == \begin{pmatrix} -4 \\ 6 \\ 4 \end{pmatrix} \\ \bar{\mathbf{z}} &= \begin{pmatrix} -4 \\ 6 \\ 4 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} -4 \\ 6 \\ 0 \end{pmatrix}, \overline{\mathbf{W}^{(1)}} = (\bar{\mathbf{z}}\mathbf{x}^\top)^\top = \begin{pmatrix} -4 & 6 & 0 \\ -12 & 18 & 0 \\ -4 & 6 & 0 \end{pmatrix}, \overline{\mathbf{W}^{(2)}} = (\bar{\mathbf{y}}\bar{\mathbf{h}}^\top)^\top = \begin{pmatrix} 8 & 8 & 8 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} \\ \|\mathbf{W}^{(1)}\|_F^2 &= \text{trace} \left(\begin{pmatrix} 176 & -264 & 0 \\ -264 & 396 & 0 \\ 0 & 0 & 0 \end{pmatrix} \right) = 572, \|\mathbf{W}^{(2)}\|_F^2 = \text{trace} \left(\begin{pmatrix} 65 & 65 & 65 \\ 65 & 65 & 65 \\ 65 & 65 & 65 \end{pmatrix} \right) = 195\end{aligned}$$

ii Efficient Computation [0.5pt]

Notice that weight Jacobian can be expressed as the outer product of the error vector and activation $\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}} = (\bar{\mathbf{z}}\mathbf{x}^\top)^\top$ and $\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(2)}} = (\bar{\mathbf{y}}\bar{\mathbf{h}}^\top)^\top$. We can compute the Jacobian norm more efficiently using the following trick:

$$\begin{aligned}\left\| \frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}} \right\|_F^2 &= \text{trace} \left(\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}}^\top \frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}} \right) \quad (\text{Definition}) \\ &= \text{trace}(\bar{\mathbf{z}}\mathbf{x}^\top \mathbf{x}\bar{\mathbf{z}}^\top) \\ &= \text{trace}(\mathbf{x}^\top \mathbf{x}\bar{\mathbf{z}}^\top \bar{\mathbf{z}}) \quad (\text{Cyclic Property of Trace}) \\ &= (\mathbf{x}^\top \mathbf{x})(\bar{\mathbf{z}}^\top \bar{\mathbf{z}}) \quad (\text{Scalar Multiplication})\end{aligned}$$

Compute the **square** of the Frobenius Norm of the two Jacobian matrices by plugging the value into the above trick.

Hints: Verify the solution is the same as naive computation. Show all your work for partial marks.

$$\left\| \frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}} \right\|_F^2 = \|\mathbf{x}\|_2^2 \|\bar{\mathbf{z}}\|_2^2 = 52 \times 11 = 572$$

$$\left\| \frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(2)}} \right\|_F^2 = \|\bar{\mathbf{y}}\|_2^2 \|\mathbf{h}\|_2^2 = 65 \times 3 = 195$$

iii Complexity Analysis blue[1.5pt]

Now, let us consider a general neural network with $K - 1$ hidden layers (K weight matrices). All input units, output units, and hidden units have a dimension of D . Assume we have N input vectors. How many scalar multiplications T (integer) do we need to compute the per-example gradient norm using naive and efficient computation, respectively? And, what is the memory cost M (big \mathcal{O} notation)?

For simplicity, you can ignore the activation function and loss function computation. You can assume the network does not have a bias term. You can also assume there are no in-place operations. Please fill up the table below.

	T (Naive)	T (Efficient)	M (Naive)	M (Efficient)
Forward Pass	NKD^2	NKD^2	$O(KD^2 + NKD)$	$O(KD^2 + NKD)$
Backward Pass	$N(2K-1)D^2$	NKD^2	$O(NKD^2)$	$O(NKD)$
Gradient Norm Computation	NKD^2	$NK(2D+1)$	$O(NKD^2)$	$O(NKD)$

Hints: The forward pass computes all the activations and needs memory to store model parameters and activations. The backward pass computes all the error vectors. Moreover, you also need to compute the parameter's gradient in naive computation. During the Gradient Norm Computation, the naive method needs to square the gradient before aggregation. In contrast, the efficient method relies on the trick. Thinking about the following questions may be helpful. 1) Do we need to store all activations in the forward pass? 2) Do we need to store all error vectors in the backward pass? 3) Why standard backward pass is twice more expensive than the forward pass? Don't forget to consider K and N in your answer.

chatGPT:

s

Consider a general neural network with $K - 1$ hidden layers (K weight matrices). All input units, output units, and hidden units have a dimension of D . Assume we have N input vectors. How many scalar multiplications T (integer) do we need to compute the per-example gradient norm using naive and efficient computation, respectively? And, what is the memory cost M (big O notation)? For simplicity, you can ignore the activation function and loss function computation. You can assume the network does not have a bias term. You can also assume there are no in-place operations.

Please fill up the table below.

	T (Naive)	T (Efficient)	M (Naive)	M (Efficient)
--	-------------	-----------------	-------------	-----------------

Forward Pass

Backward Pass

Gradient Norm Computation

Hints: The forward pass computes all the activations and needs memory to store model parameters and activations. The backward pass computes all the error vectors. Moreover, you also need to compute the parameter's gradient in naive computation. During the Gradient Norm Computation, the naive method needs to square the gradient before aggregation. In contrast, the efficient method relies on the trick. Thinking about the following questions may be helpful. 1) Do we need to store all activations in the forward pass? 2) Do we need to store all error vectors in the backward pass? 3) Why standard backward pass is twice more expensive than the forward pass? Don't forget to consider K and N in your answer



The table below summarizes the number of scalar multiplications (T) and memory cost (M) for a general neural network with K - 1 hidden layers and D-dimensional input units, output units, and hidden units for N input vectors, for both naive and efficient computation:

Computation	T (Naive)	T (Efficient)	M (Naive)	M (Efficient)
Forward Pass	KD^2N	KD^2N	KD^2 + ND	KD^2 + ND
Backward Pass	2KD^2N	KD^2N	2KD^2 + ND	KD^2 + ND
Gradient Norm Computation	KDN	KDN	KDN	KDN

Explanation:

- For the forward pass, the naive method requires KD^2N scalar multiplications to compute the activations and requires memory to store the model parameters (KD^2) and activations (ND). The efficient method also requires KD^2N scalar multiplications and requires the same amount of memory.
- For the backward pass, the naive method requires 2KD^2N scalar multiplications to compute the error vectors for all the layers and requires memory to store the model parameters (KD^2) and error vectors (ND). The efficient method requires KD^2N scalar multiplications and requires the same amount of memory to store the model parameters (KD^2) and error vectors (ND).
- For the gradient norm computation, both the naive and efficient methods require KDN scalar multiplications and memory to store the gradient (KDN). The difference between the two methods lies in how the gradient norm is computed. The naive method squares the gradient before aggregation, while the efficient method relies on the trick.

c Inner product of Jacobian: JVP and VJP [0pt]

A more general case of computing the gradient norm is to compute the inner product of the Jacobian matrices computed using two different examples. Let f_1, f_2 and y_1, y_2 be the final outputs and layer outputs of two different examples respectively. The inner product Θ of Jacobian matrices of layer parameterized by θ is defined as:

$$\Theta_\theta(f_1, f_2) := \frac{\partial f_1}{\partial \theta} \frac{\partial f_2}{\partial \theta}^\top = \underbrace{\frac{\partial f_1}{\partial y_1}}_{\mathbf{O} \times \mathbf{Y}} \underbrace{\frac{\partial y_1}{\partial \theta}}_{\mathbf{Y} \times \mathbf{P}} \underbrace{\frac{\partial y_2}{\partial \theta}}_{\mathbf{P} \times \mathbf{Y}} \underbrace{\frac{\partial f_2}{\partial y_2}}_{\mathbf{Y} \times \mathbf{O}}^\top = \underbrace{\frac{\partial f_1}{\partial y_1}}_{\mathbf{O} \times \mathbf{Y}} \underbrace{\frac{\partial y_1}{\partial \theta}}_{\mathbf{Y} \times \mathbf{P}} \underbrace{\frac{\partial y_2}{\partial \theta}}_{\mathbf{P} \times \mathbf{Y}} \underbrace{\frac{\partial f_2}{\partial y_2}}_{\mathbf{Y} \times \mathbf{O}}^\top,$$

Where $\mathbf{O}, \mathbf{Y}, \mathbf{P}$ represent the dimension of the final output, layer output, model parameter respectively. How to formulate the above computation using Jacobian Vector Product (JVP) and Vector Jacobian Product (VJP)? What are the computation cost using the following three ways of contracting the above equation?

- (a) Outside-in: $M_1M_2M_3M_4 = ((M_1M_2)(M_3M_4))$
- (b) Left-to-right and right-to-left: $M_1M_2M_3M_4 = (((M_1M_2)M_3)M_4) = (M_1(M_2(M_3M_4)))$
- (c) Inside-out-left and inside-out-right: $M_1M_2M_3M_4 = ((M_1(M_2M_3))M_4) = (M_1((M_2M_3)M_4))$

3 Hard-Coding Networks (2.5pts)

Can we use neural networks to tackle coding problems? Yes! In this question, you will build a neural network to find the k^{th} smallest number from a list using two different approaches: sorting and counting (Optional). You will start by constructing a two-layer perceptron “Sort_2” to sort two numbers and then use it as a building block to perform your favorite sorting algorithm (e.g., Bubble Sort, Merge Sort). Finally, you will output the k^{th} element from the sorted list as the final answer.

Note: Before doing this problem, you need to have a basic understanding of the key components of neural networks (e.g., weights, activation functions). The reading on multilayer perceptrons located at <https://uoft-csc413.github.io/2023/assets/readings/L02a.pdf> may be useful.

a Sort two numbers blue[1pt]

In this problem, you need to find a set of weights and bias for a two-layer perceptron “Sort_2” that sorts two numbers. The network takes a pair of numbers (x_1, x_2) as input and output a sorted pair (y_1, y_2) , where $y_1 \leq y_2$. You may assume the two numbers are distinct and positive for simplicity. You will use the following architecture:

Please specify the weights and activation functions for your network. Your answer should include:

- Two weight matrices: $\mathbf{W}^{(1)}, \mathbf{W}^{(2)} \in \mathbb{R}^{2 \times 2}$
- Two bias vector: $\mathbf{b}^{(1)}, \mathbf{b}^{(2)} \in \mathbb{R}^2$
- Two activation functions: $\phi^{(1)}(z), \phi^{(2)}(z)$

You do not need to show your work.

Hints: Sorting two numbers is equivalent to finding the min and max of two numbers.

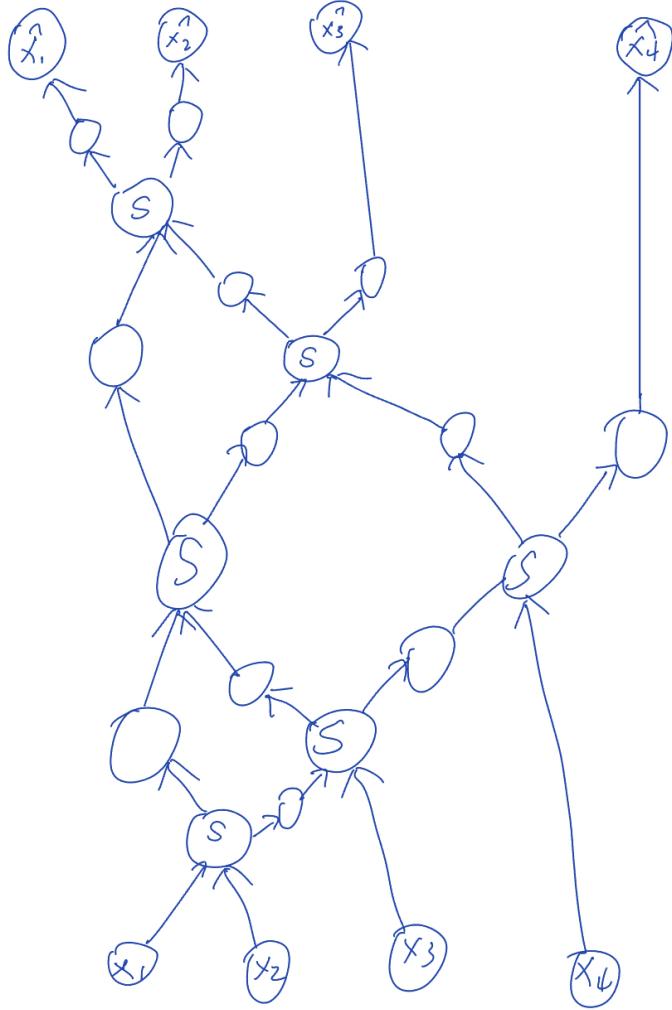
$$\max(x_1, x_2) = \frac{1}{2}(x_1 + x_2) + \frac{1}{2}|x_1 - x_2|, \quad \min(x_1, x_2) = \frac{1}{2}(x_1 + x_2) - \frac{1}{2}|x_1 - x_2|$$

$$\begin{aligned}\phi^{(1)}(z) &= |z|, \phi^{(2)}(z) = z \\ \mathbf{b}^{(1)} &= \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \mathbf{b}^{(2)} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\ \mathbf{W}^{(1)} &= \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \mathbf{W}^{(2)} = \begin{pmatrix} \frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix},\end{aligned}$$

b Perform Sort blue[1.5pt]

Draw a computation graph to show how to implement a sorting function $\hat{f} : 4 \rightarrow 4$ where $\hat{f}(x_1, x_2, x_3, x_4) = (\hat{x}_1, \hat{x}_2, \hat{x}_3, \hat{x}_4)$ where $(\hat{x}_1, \hat{x}_2, \hat{x}_3, \hat{x}_4)$ is (x_1, x_2, x_3, x_4) in sorted order. Let us assume $\hat{x}_1 \leq \hat{x}_2 \leq \hat{x}_3 \leq \hat{x}_4$ and x_1, x_2, x_3, x_4 are positive and distinct. Implement \hat{f} using your favourite sorting algorithms (e.g. Bubble Sort, Merge Sort). Let us denote the “Sort_2” module as S , please complete the following computation graph. Your answer does not need to give the label for intermediate nodes, but make sure to index the “Sort_2” module.

Hints: Bubble Sort needs 6 “Sort_2” blocks, while Merge Sort needs 5 “Sort_2” blocks.



c Find the k^{th} smallest number [0pt]

Based on your sorting network, you may want to add a new layer to output your final result (k^{th} smallest number). Please give the weight $\mathbf{W}^{(3)}$ for this output layer when $k = 3$.

Hints: $\mathbf{W}^{(3)} \in \mathbb{R}^{1 \times 4}$.

d Counting Network [0pt]

The idea of using a counting network to find the k^{th} smallest number is to build a neural network that can determine the rank of each number and output the number with the correct rank. Specifically, the counting network will count how many elements in a list are less than a value of interest. And you will apply the counting network to all numbers in the given list to determine their rank. Finally, you will use another layer to output the number with the correct rank.

The counting network has the following architecture, where y is the rank of x_1 in a list containing x_1, x_2, x_3, x_4 . Please specify the weights and activation functions for your counting network. Draw a diagram to show how you will use the counting network and give a set of weights and biases for the final layer to find the k^{th} smallest number. In other words, repeat the process of sections 1.1, 1.2, 1.3 using the counting idea.

Hints: You may find the following two activation functions useful.

1) Hard threshold activation function:

$$\phi(z) = (z \geq 0) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

2) Indicator activation function:

$$\phi(z) = (z \in [-1, 1]) = \begin{cases} 1 & \text{if } z \in [-1, 1] \\ 0 & \text{otherwise} \end{cases}$$

4

4.2

$$L(\mathbf{W}, \tilde{\mathbf{W}}, b, \tilde{b}) = \|\mathbf{W}\tilde{\mathbf{W}}^\top + b\mathbf{1}^\top + \mathbf{1}\tilde{b}^\top - \log X\|_F^2$$

chatGPT:

s

In practice, we concatenate the V embedding vectors into matrices $\mathbf{W}, \mathbf{W} \in RV \times d$ and bias (column) vectors $b, b \in RV$, where V denotes the number of distinct words as described in the introduction. Rewrite the loss function L (Eq. 1) in a vectorized format in terms of $\mathbf{W}, \mathbf{W}^\top, b, b^\top, \mathbf{X}$. You are allowed to use elementwise operations such as addition and subtraction as well as matrix operations such as the Frobenius norm and/or trace operator in your answer.

Hint: Use the all-ones column vector $\mathbf{1} = [1 \dots 1]^\top \in RV$. You can assume the bias vectors are column vectors, i.e. implicitly a matrix with V rows and 1 column: $b, b \in RV \times 1$

Hint: To prompt a GPT-like model, try naively copy-pasting the question, generate several answers several times, and check the generated answers.



The loss function in equation 1 can be rewritten in a vectorized format as:



$$L = \| \mathbf{X} - (\mathbf{W} * \mathbf{W}^\top + b * \mathbf{1}^\top + \mathbf{1} * b^\top) \|_F^2$$

where \mathbf{X} is an $RV \times d$ matrix representing the input word vectors, \mathbf{W} is an $RV \times d$ matrix representing the left embedding weights, \mathbf{W}^\top is an $RV \times d$ matrix representing the right embedding weights, b is an $RV \times 1$ vector representing the left bias, b^\top is an $RV \times 1$ vector representing the right bias, and $\mathbf{1}$ is an $RV \times 1$ all-ones vector. The Frobenius norm is used to measure the distance between \mathbf{X} and the dot product of $\mathbf{W}, \mathbf{W}^\top, b, b^\top$, and $\mathbf{1}$.

4.3

$$\frac{\partial l}{\partial \hat{w}} = 2(\mathbf{W}\tilde{\mathbf{W}}^\top + b\mathbf{1}^\top + \mathbf{1}\tilde{\mathbf{b}}^\top - \log X)\tilde{\mathbf{W}}$$

chatGPT:

s

Write the vectorized expression for $\partial L / \partial W$, the gradient of the loss function L with respect to the embedding matrix W . The gradient should be a function of $W, W \sim, b, b \sim, X$.

Hint: Make sure that the shape of the gradient is equivalent to the shape of the matrix. You can use the all-ones vector as in the previous question.



The gradient of the loss function with respect to the embedding matrix W can be expressed as follows: 👍 🔍

$$\partial L / \partial W = 2(WX - W \sim X)X^T + 2\lambda W$$

Where X is the one-hot representation of the input text, $W \sim$ is the corresponding word embedding matrix, λ is the regularization parameter, and the Frobenius norm is being used. The gradient is a matrix with the same dimensions as the W matrix, i.e. $V \times d$.

4.4

```
def loss_GloVe(W, W_tilde, b, b_tilde, log_co_occurrence):
    """ Compute the GloVe loss given the parameters of the model. When W_tilde
    and b_tilde are not given, then the model is symmetric (i.e. W_tilde = W,
    b_tilde = b).

    Args:
        W: word embedding matrix, dimension V x d where V is vocab size and d
            is the embedding dimension
        W_tilde: for asymmetric GloVe model, a second word embedding matrix, with
            dimensions V x d
        b: bias vector, dimension V x 1
        b_tilde: for asymmetric GloVe model, a second bias vector, dimension V x 1
        log_co_occurrence: V x V log co-occurrence matrix (log X)

    Returns:
        loss: a scalar (float) for GloVe loss
    """
    n,_ = log_co_occurrence.shape
    ones = np.ones(n).reshape(-1, 1)
    # Symmetric Case, no W_tilde and b_tilde
    if W_tilde is None and b_tilde is None:
        # Symmetric model
        ##### YOUR CODE HERE #####
        W_tilde = W
        b_tilde = b
        inter = np.matmul(W, np.transpose(W_tilde)) + np.matmul(b, np.transpose(ones)) + np.matmul(ones, np.transpose(b_tilde)) - log_co_occurrence
        loss = np.trace(np.matmul(np.transpose(inter), inter))
        #####
    else:
        # Asymmetric model
        ##### YOUR CODE HERE #####
        inter = np.matmul(W, np.transpose(W_tilde)) + np.matmul(b, np.transpose(ones)) + np.matmul(ones, np.transpose(b_tilde)) - log_co_occurrence
        loss = np.trace(np.matmul(np.transpose(inter), inter))
        #####
    return loss
```

chatGPT:

```
s def loss_GloVe(W, W_tilde, b, b_tilde, log_co_occurrence):  
    """ Compute the GloVe loss given the parameters of the model. When W_tilde  
    and b_tilde are not given, then the model is symmetric (i.e. W_tilde = W,  
    b_tilde = b).  
    """
```

Args:

W: word embedding matrix, dimension V x d where V is vocab size and d
is the embedding dimension
W_tilde: for asymmetric GloVe model, a second word embedding matrix, with
dimensions V x d
b: bias vector, dimension V x 1
b_tilde: for asymmetric GloVe model, a second bias vector, dimension V x 1
log_co_occurrence: V x V log co-occurrence matrix (log X)

Returns:

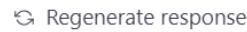
loss: a scalar (float) for GloVe loss

"""

```
n,_ = log_co_occurrence.shape  
ones = np.ones(n).reshape(-1, 1)  
# Symmetric Case, no W_tilde and b_tilde  
if W_tilde is None and b_tilde is None:  
    # Symmetric model  
    ##### YOUR CODE HERE #####
```

```
#####  
else:  
    # Asymmetric model  
    ##### YOUR CODE HERE #####
```

```
#####  
return loss
```

 Regenerate response



The GloVe loss function is given by:



$$L = \sum_{i,j} (f(X_{ij}) (W_i^T W_j^{\text{Tilde}} + b_i + b_j - \log(X_{ij}))^2)$$

Here, $f(X_{ij})$ is a weighting function that determines the contribution of each log co-occurrence count X_{ij} to the total loss. The weighting function can be set to be equal to 1 for all i, j .

For the symmetric case, where $W_{\text{tilde}} = W$ and $b_{\text{tilde}} = b$, the loss function can be written as:

$$L = \sum_{i,j} (f(X_{ij}) (2W_i^T W_j + b_i + b_j - \log(X_{ij}))^2)$$

In this case, the implementation of the loss function would be:

python

Copy code

```
if W_tilde is None and b_tilde is None:  
    dot_product = np.dot(W, W.T)  
    dot_product *= 2  
    bias = b + b.T  
    inner = dot_product + bias - log_co_occurrence  
    loss = np.sum(f(log_co_occurrence) * (inner**2))
```

For the asymmetric case, the implementation of the loss function would be:

sql

Copy code

```
else:  
    dot_product = np.dot(W, W_tilde.T)  
    bias = b + b_tilde.T  
    inner = dot_product + bias - log_co_occurrence  
    loss = np.sum(f(log_co_occurrence) * (inner**2))
```

4.5

```

def grad_GloVe(W, W_tilde, b, b_tilde, log_co_occurrence):
    """Return the gradient of GloVe objective w.r.t its parameters

    Args:
        W: word embedding matrix, dimension V x d where V is vocab size and d
            is the embedding dimension
        W_tilde: for asymmetric GloVe model, a second word embedding matrix, with
            dimensions V x d
        b: bias vector, dimension V x 1
        b_tilde: for asymmetric GloVe model, a second bias vector, dimension V x 1
        log_co_occurrence: V x V log co-occurrence matrix (log X)

    Returns:
        grad_W: gradient of the loss wrt W, dimension V x d
        grad_W_tilde: gradient of the loss wrt W_tilde, dimension V x d. Return
            None if W_tilde is None.
        grad_b: gradient of the loss wrt b, dimension V x 1
        grad_b_tilde: gradient of the loss wrt b_tilde, dimension V x 1. Return
            None if b_tilde is None.

    """
    n,_ = log_co_occurrence.shape
    ones = np.ones(n).reshape(-1, 1)
    if W_tilde is None and b_tilde is None:
        # Symmetric case
        ##### YOUR CODE HERE #####
        grad_W_tilde = None
        grad_b_tilde = None
        W_matrix = np.matmul(b, np.transpose(ones)) + np.matmul(ones, np.transpose(b)) - log_co_occurrence
        b_matrix = np.matmul(W, np.transpose(W)) - log_co_occurrence

        grad_W = 4 * (np.matmul(W, np.matmul(np.transpose(W), W))) + 2 * np.matmul((W_matrix + np.transpose(W_matrix)), W)
        grad_b = 4 * np.matmul(b, np.matmul(np.transpose(ones), ones)) + 4 * np.matmul(ones, np.matmul(np.transpose(b), ones)) + 2 * np.matmul(b_matrix, ones) + 2 * np.matmul(np.transpose(b_matrix), ones)
    ##### #####
    else:
        # Asymmetric case
        ##### YOUR CODE HERE #####
        W_matrix = np.matmul(W, np.transpose(W_tilde)) + np.matmul(b, np.transpose(ones)) + np.matmul(ones, np.transpose(b_tilde)) - log_co_occurrence
        b_matrix = np.matmul(W, np.transpose(W_tilde)) + np.matmul(ones, np.transpose(b_tilde)) - log_co_occurrence
        b_tilde_matrix = np.matmul(W, np.transpose(W_tilde)) + np.matmul(b, np.transpose(ones)) - log_co_occurrence

        grad_W = 2 * np.matmul(W_matrix, W_tilde)
        grad_W_tilde = 2 * np.matmul(np.transpose(W_matrix), W)
        grad_b = 2 * np.matmul(np.matmul(b, np.transpose(ones)), ones) + 2 * np.matmul(b_matrix, ones)
        grad_b_tilde = 2 * np.matmul(np.matmul(b_tilde, np.transpose(ones)), ones) + 2 * np.matmul(np.transpose(b_tilde_matrix), ones)
    ##### #####
    return grad_W, grad_W_tilde, grad_b, grad_b_tilde

```

chatGPT:

```
s def grad_GloVe(W, W_tilde, b, b_tilde, log_co_occurrence):
    """Return the gradient of GloVe objective w.r.t its parameters

    Args:
        W: word embedding matrix, dimension V x d where V is vocab size and d
            is the embedding dimension
        W_tilde: for asymmetric GloVe model, a second word embedding matrix, with
            dimensions V x d
        b: bias vector, dimension V x 1
        b_tilde: for asymmetric GloVe model, a second bias vector, dimension V x 1
        log_co_occurrence: V x V log co-occurrence matrix (log X)

    Returns:
        grad_W: gradient of the loss wrt W, dimension V x d
        grad_W_tilde: gradient of the loss wrt W_tilde, dimension V x d. Return
            None if W_tilde is None.
        grad_b: gradient of the loss wrt b, dimension V x 1
        grad_b_tilde: gradient of the loss wrt b_tilde, dimension V x 1. Return
            None if b_tilde is None.

    """
    n_ = log_co_occurrence.shape[0]
    ones = np.ones(n).reshape(-1, 1)
    if W_tilde is None and b_tilde is None:
        # Symmetric case
        ##### YOUR CODE HERE #####
    else:
        # Asymmetric case
        ##### YOUR CODE HERE #####
    return grad_W, grad_W_tilde, grad_b, grad_b_tilde
```

```

def grad_GloVe(W, W_tilde, b, b_tilde, log_co_occurrence):
    """Return the gradient of GloVe objective w.r.t its parameters

    Args:
        W: word embedding matrix, dimension V x d where V is vocab size and d
            is the embedding dimension
        W_tilde: for asymmetric GloVe model, a second word embedding matrix, with
            dimensions V x d
        b: bias vector, dimension V x 1
        b_tilde: for asymmetric GloVe model, a second bias vector, dimension V x 1
        log_co_occurrence: V x V log co-occurrence matrix (log X)

    Returns:
        grad_W: gradient of the loss wrt W, dimension V x d
        grad_W_tilde: gradient of the loss wrt W_tilde, dimension V x d. Return
            None if W_tilde is None.
        grad_b: gradient of the loss wrt b, dimension V x 1
        grad_b_tilde: gradient of the loss wrt b, dimension V x 1. Return
            None if b_tilde is None.
    """
    n, _ = log_co_occurrence.shape
    ones = np.ones(n).reshape(-1, 1)
    if W_tilde is None and b_tilde is None:
        # Symmetric case
        # vectorized form of the loss
        f = (W @ W.T + b @ ones.T + ones @ b.T - log_co_occurrence * 2) ** 2

        # gradient wrt W
        grad_W = 4 * W @ f
        # gradient wrt b
        grad_b = 4 * b @ f

        grad_W_tilde = None
        grad_b_tilde = None

```

```

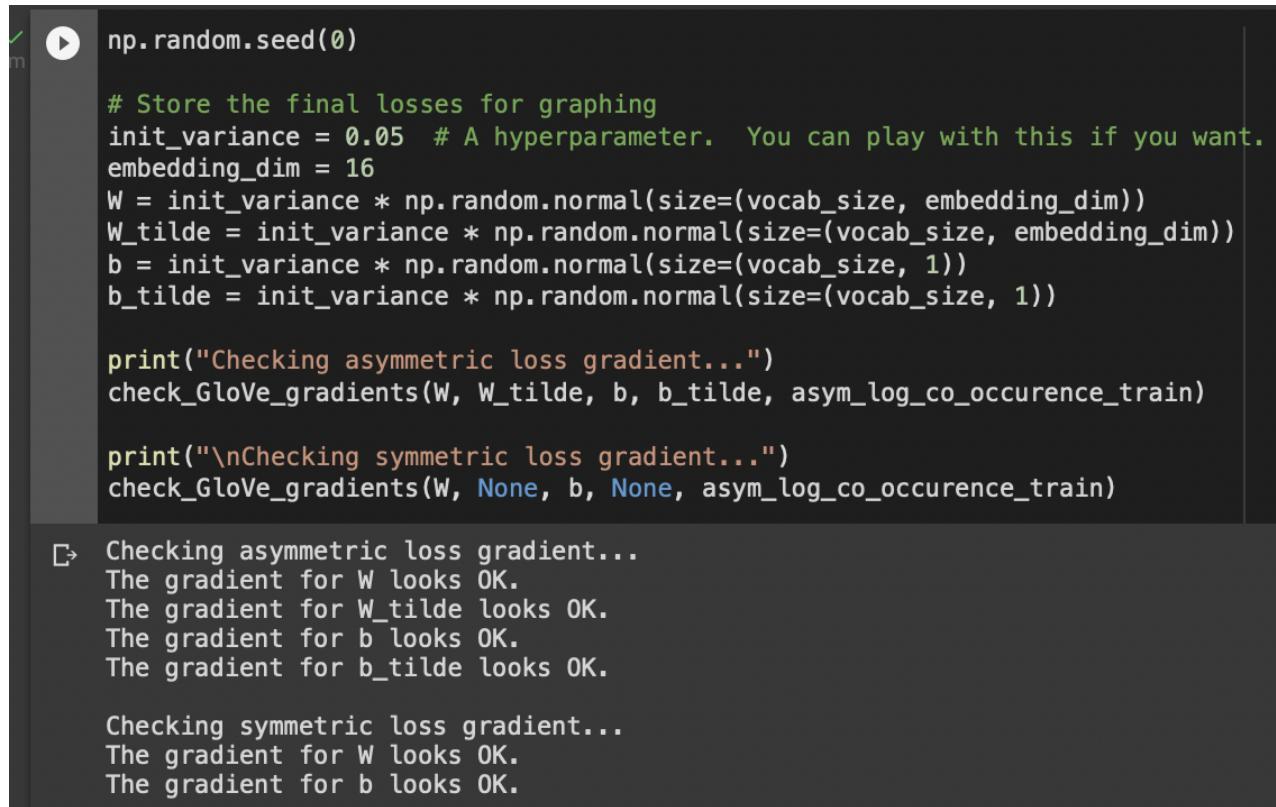
else:
    # Asymmetric case
    # vectorized form of the loss
    f = (W @ W_tilde.T + b @ ones.T + ones @ b_tilde.T - log_co_occurrence) ** 2

    # gradient wrt W
    grad_W = 4 * W @ f
    # gradient wrt W_tilde
    grad_W_tilde = 4 * W_tilde @ f
    # gradient wrt b
    grad_b = 4 * b @ f
    # gradient wrt b_tilde
    grad_b_tilde = 4 * b_tilde @ f

return grad_W, grad_W_tilde, grad_b, grad_b_tilde

```

output:



```

np.random.seed(0)

# Store the final losses for graphing
init_variance = 0.05 # A hyperparameter. You can play with this if you want.
embedding_dim = 16
W = init_variance * np.random.normal(size=(vocab_size, embedding_dim))
W_tilde = init_variance * np.random.normal(size=(vocab_size, embedding_dim))
b = init_variance * np.random.normal(size=(vocab_size, 1))
b_tilde = init_variance * np.random.normal(size=(vocab_size, 1))

print("Checking asymmetric loss gradient...")
check_GloVe_gradients(W, W_tilde, b, b_tilde, asym_log_co_occurrence_train)

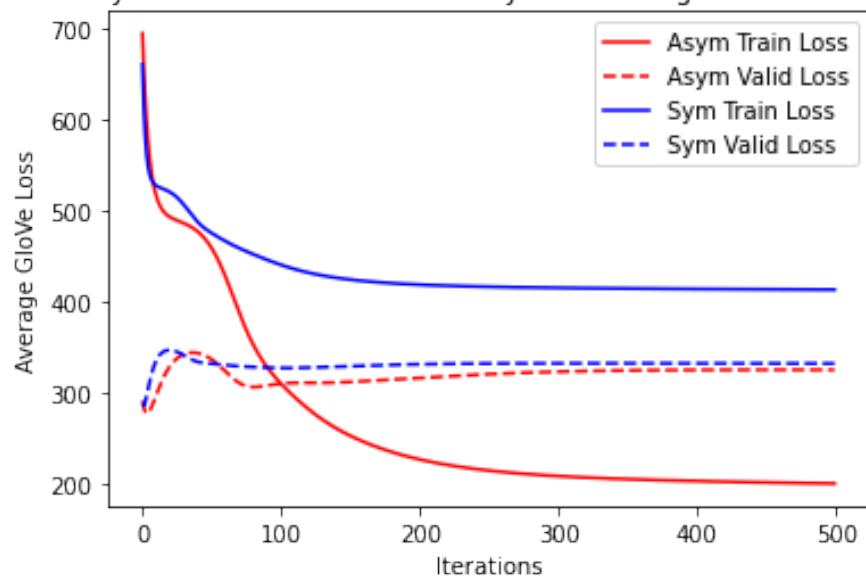
print("\nChecking symmetric loss gradient...")
check_GloVe_gradients(W, None, b, None, asym_log_co_occurrence_train)

Checking asymmetric loss gradient...
The gradient for W looks OK.
The gradient for W_tilde looks OK.
The gradient for b looks OK.
The gradient for b_tilde looks OK.

Checking symmetric loss gradient...
The gradient for W looks OK.
The gradient for b looks OK.

```

Asymmetric and Symmetric GloVe Model on Asymmetric Log Co-Occurrence (Emb Dim=16)



5

5.1

5.1

Answer:

word_embedding_weights: ND
embed_to_hid_weights: DNH
hid_bias: H
hid_to_output_weights: NVH
output_bias: NV

The total number of trainable parameters is the sum of all parameters above, which is $ND + DNH + H + NVH + NV$ and the largest number of trainable parameters is hid_to_output_weights since V is much greater than other three variables.

chatGPT:

s

We will train a neural language model. However, we will modify the architecture slightly, inspired by the Masked Language Modeling (MLM) objective introduced in BERT. The network takes in N consecutive words, where one of the words is replaced with a [MASK] token¹². The aim of the network is to predict the masked word in the corresponding output location. The network consists of an input layer, embedding layer, hidden layer and output layer. The input consists of a sequence of N consecutive words, with each word given as integer valued indices.(e.g., the 250 words in our dictionary are arbitrarily assigned integer values from 0 to 249.) The embedding layer maps each word to its corresponding vector representation. Each of the N context words are mapped independently using the same word embedding weights matrix. The embedding layer has $N \times D$ units, where D is the embedding dimension of a single word. The embedding layer is fully connected to the hidden layer with H units, which uses a logistic nonlinearity. The hidden layer in turn is connected to the logits output layer, which has $N \times V$ units. Finally, softmax over V logit output units is applied to each consecutive V logit output units, where V is the number of words in the dictionary (including the [MASK] token). The trainable parameters of the model consist of 3 weight matrices and 2 sets of biases. What is the total number of trainable parameters in the model, as a function of V, N, D, H? In the diagram given above, which part of the model (i.e., word_embedding_weights, embed_to_hid_weights, hid_to_output_weights, hid_bias, or output_bias) has the largest number of trainable parameters if we have the constraint that $V \gg H > D > N$? Explain your reasoning.



The total number of trainable parameters in the model is a function of V , N , D , and H :

- word_embedding_weights: $V \times D$
- embed_to_hid_weights: $D \times H$
- hid_to_output_weights: $H \times (N \times V)$
- hid_bias: $H \times 1$
- output_bias: $N \times V$

So the total number of trainable parameters is $V \times D + D \times H + H \times (N \times V) + H + N \times V$.

If we have the constraint that $V \gg H > D > N$, then the number of parameters in hid_to_output_weights ($H \times (N \times V)$) is the largest among the five parts of the model. This is because H , the number of units in the hidden layer, is smaller than V , the number of words in the dictionary, but larger than N , the number of consecutive words taken as input. Meanwhile, $N \times V$, the number of logits output units, is larger than H and D .

5.2

5.2 Number of parameters in n -gram model [1pt] [Type 1]

Another method for predicting the next words is an *n -gram model*, which was mentioned in Lecture 3. If we wanted to use an n -gram model with the same context length $N - 1$ as our network (since we mask 1 of the N words in our input), we'd need to store the counts of all possible N -grams. If we stored all the counts explicitly and suppose that we have V words in the dictionary, how many entries would this table have?

5.2 Answer: V^N (V words and N word positions)

chatGPT:

s

We will train a neural language model. However, we will modify the architecture slightly, inspired by the Masked Language Modeling (MLM) objective introduced in BERT. The network takes in N consecutive words, where one of the words is replaced with a [MASK] token¹². The aim of the network is to predict the masked word in the corresponding output location. The network consists of an input layer, embedding layer, hidden layer and output layer. The input consists of a sequence of N consecutive words, with each word given as integer valued indices.(e.g., the 250 words in our dictionary are arbitrarily assigned integer values from 0 to 249.) The embedding layer maps each word to its corresponding vector representation. Each of the N context words are mapped independently using the same word embedding weights matrix. The embedding layer has $N \times D$ units, where D is the embedding dimension of a single word. The embedding layer is fully connected to the hidden layer with H units, which uses a logistic nonlinearity. The hidden layer in turn is connected to the logits output layer, which has $N \times V$ units. Finally, softmax over V logit output units is applied to each consecutive V logit output units, where V is the number of words in the dictionary (including the [MASK] token). Another method for predicting the next words is an n-gram model. If we wanted to use an n-gram model with the same context length $N - 1$ as our network, we'd need to store the counts of all possible N-grams. If we stored all the counts explicitly and suppose that we have V words in the dictionary, how many entries would this table have?



The total number of trainable parameters in the model is given by:

thumb up thumb down

- Word embedding weights: $V \times D$
- Embed to hidden layer weights: $H \times (N \times D)$
- Hidden layer biases: $H \times 1$
- Hidden to output layer weights: $V \times H$
- Output layer biases: $V \times 1$

The word embedding weights matrix has the largest number of parameters, if we have the constraint that V is greater than H, which is greater than D, which is greater than N. This is because it has $V \times D$ parameters, and V is the largest among the four values.

In an n-gram model, each N-gram is a sequence of N words, and we need to store the count of each N-gram in the table. If we have V words in the dictionary, there will be V^N possible N-grams, and hence the table would have V^N entries.

6.1

```
def compute_loss(self, output_activations, expanded_target_batch, target_mask):
    """Compute the total cross entropy loss over a mini-batch.

    Args:
        output_activations: [batch_size x (context_len * vocab_size)] matrix,
            for the activations of the output layer, i.e. the  $y_j$ 's.
        expanded_target_batch: [batch_size (context_len * vocab_size)] matrix,
            where expanded_target_batch[i,n*V:(n+1)*V] is the indicator vector for
            the n-th context target word position, i.e. the  $(i, j + n*V)$  entry is 1 if the
            i'th example, the context word at position n is j, and 0 otherwise. matrix obtained
        target_mask: A [batch_size x context_len x 1] tensor, where target_mask[i,n,0] = 1
            if for the i'th example the n-th context word is a target position, otherwise 0

    Returns:
        loss: a scalar for the total cross entropy loss over the batch,
            defined in Part 3
    ....
    ##### YOUR CODE HERE #####
    target = expanded_target_batch.reshape(-1, self.context_len, len(self.vocab))
    output = output_activations.reshape(-1, self.context_len, len(self.vocab))
    loss = - np.sum(target * np.log(output + TINY))

    #####
    return loss
```

chatGPT:

Question: write the code of compute_loss

class Model(object):

"""A class representing the language model itself. This class contains various methods used in training

the model and visualizing the learned representations. It has two fields:

params, a Params instance which contains the model parameters

vocab, a list containing all the words in the dictionary; vocab[0] is the word with index 0, and so on."""

```
def __init__(self, params, vocab):
```

```
    self.params = params
```

```
    self.vocab = vocab
```

```
    self.vocab_size = len(vocab)
```

```
    self.embedding_dim = self.params.word_embedding_weights.shape[1]
```

```
    self.embedding_layer_dim = self.params.embed_to_hid_weights.shape[1]
```

```
    self.context_len = self.embedding_layer_dim // self.embedding_dim
```

```
    self.num_hid = self.params.embed_to_hid_weights.shape[0]
```

```
def copy(self):
```

```
    return self.__class__(self.params.copy(), self.vocab[:])
```

```
@classmethod
```

```
def random_init(cls, init_wt, vocab, context_len, embedding_dim, num_hid):
```

"""Constructor which randomly initializes the weights to Gaussians with standard deviation init_wt

and initializes the biases to all zeros."""

```
    params = Params.random_init(init_wt, len(vocab), context_len, embedding_dim,  
    num_hid)
```

```
    return Model(params, vocab)
```

Regenerate response

```

def indicator_matrix(self, targets, mask_zero_index=True):
    """Construct a matrix where the (v + n*V)th entry of row i is 1 if the n-th target word
    for example i is v, and all other entries are 0.

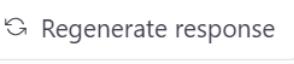
    Note: if the n-th target word index is 0, this corresponds to the [MASK] token,
          and we set the entry to be 0.
    """
    batch_size, context_len = targets.shape
    expanded_targets = np.zeros((batch_size, context_len * len(self.vocab)))
    offset = np.repeat((np.arange(context_len) * len(self.vocab))[np.newaxis, :], batch_size,
axis=0) # [[0, V, 2V], [0, V, 2V], ...]
    targets_offset = targets + offset

    for c in range(context_len):
        expanded_targets[np.arange(batch_size), targets_offset[:,c]] = 1.
    if mask_zero_index:
        # Note: Set the targets with index 0, V, 2V to be zero since it corresponds to the
        # [MASK] token
        expanded_targets[np.arange(batch_size), offset[:,c]] = 0.
    return expanded_targets

def compute_loss(self, output_activations, expanded_target_batch, target_mask):
    """Compute the total cross entropy loss over a mini-batch.

```

Args:

output_activations: [batch_size x (context_len * vocab_size)] matrix,
 for the activations of the output layer, i.e. the y_j 's.
 expanded_target_batch: [batch_size (context_len * vocab_size)] matrix,
 where expanded_target_batch[i,n*V:(n+1)*V] is the indicator vector for
 the n-th context target word position, i.e. the $(i, j + n*V)$ entry is 1 if the
 i'th example, the context
 target_mask: A [batch_size x ] matrix obtained
 here target_mask[i,n,0] = 1

Args:

output_activations: [batch_size x (context_len * vocab_size)] matrix,
for the activations of the output layer, i.e. the y_j 's.
expanded_target_batch: [batch_size (context_len * vocab_size)] matrix,
where expanded_target_batch[i,n*V:(n+1)*V] is the indicator vector for
the n-th context target word position, i.e. the $(i, j + n*V)$ entry is 1 if the
i'th example, the context word at position n is j, and 0 otherwise. matrix obtained
target_mask: A [batch_size x context_len x 1] tensor, where target_mask[i,n,0] = 1
if for the i'th example the n-th context word is a target position, otherwise 0

Returns:

loss: a scalar for the total cross entropy loss over the batch,
defined in Part 3

YOUR CODE HERE
#####

 Regenerate response

Here is the implementation of the `compute_loss` function using vectorized operations:



python

Copy code

```
def compute_loss(params, inputs, target_word_indices, N):
    """Compute the total cross-entropy loss on a mini-batch of examples.

    params: an instance of the Params class, containing the model parameters.
    inputs: a matrix of size V x N, where V is the number of words in the vocabulary
            inputs[c, i] = 1 if the c-th word of the vocabulary appears in the i-th
    target_word_indices: a matrix of size N x 1, where target_word_indices[i, 0] is
            N: the batch size.

    Returns: a scalar, the average cross-entropy loss over the N examples in the mini-batch.
    """
    # Compute hidden activations
    embedded_inputs = np.dot(inputs, params.word_embedding_weights)
    embedded_inputs = embedded_inputs.reshape((N, -1))
    hid = np.dot(embedded_inputs, params.embed_to_hid_weights.T) + params.hid_bias
    hid = np.maximum(hid, 0)

    # Compute output activations
    output = np.dot(hid, params.hid_to_output_weights.T) + params.output_bias
    output = output.reshape((N, -1))

    # Compute cross-entropy loss
    exp_output = np.exp(output)
    probs = exp_output / np.sum(exp_output, axis=1, keepdims=True)
    cross_entropy_loss = -np.log(probs[np.arange(N), target_word_indices.reshape(-1)])
    return np.mean(cross_entropy_loss)
```

6.2

```
# The matrix with values dC / dz_j, where dz_j is the input to the jth hidden unit,
# i.e. h_j = 1 / (1 + e^{-z_j})
hid_deriv = np.dot(loss_derivative, self.params.hid_to_output_weights) \
            * activations.hidden_layer * (1. - activations.hidden_layer)

hid_to_output_weights_grad = np.dot(loss_derivative.T, activations.hidden_layer)

#####
# YOUR CODE HERE #####
output_bias_grad = loss_derivative.sum(0)
embed_to_hid_weights_grad = np.dot(hid_deriv.T, activations.embedding_layer)
#####

hid_bias_grad = hid_deriv.sum(0)

# The matrix of derivatives for the embedding layer
embed_deriv = np.dot(hid_deriv, self.params.embed_to_hid_weights)

# Word Embedding Weights gradient
word_embedding_weights_grad = np.dot(self.indicator_matrix(input_batch.reshape([-1,1]), mask_zero_index=False).T,
                                         embed_deriv.reshape([-1, self.embedding_dim]))

return Params(word_embedding_weights_grad, embed_to_hid_weights_grad, hid_to_output_weights_grad,
              hid_bias_grad, output_bias_grad)
```

chatGPT:

s back_propagate is the function which computes the gradient of the loss with respect to model parameters using backpropagation. It uses the derivatives computed by compute_loss_derivative. Some parts are already filled in for you, but you need to compute the matrices of derivatives for embed_to_hid_weights and output_bias. These matrices have the same sizes as the parameter matrices. Look for the `## YOUR CODE HERE ##` comment for where to complete the code. In order to implement backpropagation efficiently, you need to express the computations in terms of matrix operations, rather than for loops. You should first work through the derivatives on pencil and paper. First, apply the chain rule to compute the derivatives with respect to individual units, weights, and biases. Next, take the formulas you've derived, and express them in matrix form. You should be able to express all of the required computations using only matrix multiplication, matrix transpose, and element-wise operations — no for loops! If you want inspiration, read through the code for Model.compute_activations and try to understand how the matrix operations correspond to the computations performed by all the units in the network.

Hints: Your implementations should also be similar to hid_to_output_weights_grad, hid_bias_grad in the same function call

```
def back_propagate(self, input_batch, activations, loss_derivative):
```

```
    """Compute the gradient of the loss function with respect to the trainable parameters  
    of the model.
```

Part of this function is already completed, but you need to fill in the derivative computations for hid_to_output_weights_grad, output_bias_grad, embed_to_hid_weights_grad, and hid_bias_grad. See the documentation for the Params class for a description of what these matrices represent.

Args:

input_batch: A [batch_size x context_length] matrix containing the indices of the context words

activations: an Activations object representing the output of Model.compute_activations

loss_derivative: A [batch_size x (context_len * vocab_size)] matrix, where loss_derivative[i,0:vocab_size] contains the gradient dC / dz_0 for the i-th training example gradient for 1st output context word, and loss_derivative[i,vocab_size:2*vocab_size] for the 2nd output context word of the i-th training example, etc.
Obtained from calling compute_loss_derivative()

Returns:

```
    Params object containing the gradient for word_embedding_weights_grad,
    embed_to_hid_weights_grad, hid_to_output_weights_grad,
    hid_bias_grad, output_bias_grad
    ....
# The matrix with values dC / dz_j, where dz_j is the input to the jth hidden unit,
# i.e. h_j = 1 / (1 + e^{-z_j})
hid_deriv = np.dot(loss_derivative, self.params.hid_to_output_weights) \
    * activations.hidden_layer * (1. - activations.hidden_layer)

hid_to_output_weights_grad = np.dot(loss_derivative.T, activations.hidden_layer)

#####
##### YOUR CODE HERE
#####

#####
#
hid_bias_grad = hid_deriv.sum(0)

# The matrix of derivatives for the embedding layer
embed_deriv = np.dot(hid_deriv, self.params.embed_to_hid_weights)

# Word Embedding Weights gradient
word_embedding_weights_grad = np.dot(self.indicator_matrix(input_batch.reshape([-1,1]),
mask_zero_index=False).T,
embed_deriv.reshape([-1, self.embedding_dim]))

return Params(word_embedding_weights_grad, embed_to_hid_weights_grad,
hid_to_output_weights_grad,
hid_bias_grad, output_bias_grad)
```

```

def sample_input_mask(self, batch_size):
    """Samples a binary mask for the inputs of size batch_size x context_len
    For each row, at most one element will be 1.
    """
    mask_idx = np.random.randint(self.context_len, size=(batch_size,))
    mask = np.zeros((batch_size, self.context_len), dtype=np.int)# Convert to one hot B x N, B batch
size, N context len
    mask[np.arange(batch_size), mask_idx] = 1
    return mask

def evaluate(self, inputs, batch_size=100):
    """Compute the average cross-entropy over a dataset.

    inputs: matrix of shape D x N"""
    ndata = inputs.shape[0]

    total = 0.
    for input_batch in get_batches(inputs, batch_size):
        mask = self.sample_input_mask(batch_size)
        input_batch_masked = input_batch * (1 - mask)
        activations = self.compute_activations(input_batch_masked)
        expanded_target_batch = self.indicator_matrix(input_batch)
        target_mask = np.expand_dims(mask, axis=2)
        cross_entropy = self.compute_loss(activations.output_layer, expanded_target_batch,
target_mask)
        total += cross_entropy

    return total / float(ndata)

def display_nearest_words(self, word, k=10):
    """List the k words nearest to a given word, along with their distances."""

    if word not in self.vocab:
        print('Word "{}" not in vocabulary.'.format(word))
        return

```

```

# Compute distance to every other word.
idx = self.vocab.index(word)
word_rep = self.params.word_embedding_weights[idx, :]
diff = self.params.word_embedding_weights - word_rep.reshape((1, -1))
distance = np.sqrt(np.sum(diff ** 2, axis=1))

# Sort by distance.
order = np.argsort(distance)
order = order[1:1 + k] # The nearest word is the query word itself, skip that.
for i in order:
    print('{}: {}'.format(self.vocab[i], distance[i]))

def word_distance(self, word1, word2):
    """Compute the distance between the vector representations of two words."""

    if word1 not in self.vocab:
        raise RuntimeError('Word "{}" not in vocabulary.'.format(word1))
    if word2 not in self.vocab:
        raise RuntimeError('Word "{}" not in vocabulary.'.format(word2))

    idx1, idx2 = self.vocab.index(word1), self.vocab.index(word2)
    word_rep1 = self.params.word_embedding_weights[idx1, :]
    word_rep2 = self.params.word_embedding_weights[idx2, :]
    diff = word_rep1 - word_rep2
    return np.sqrt(np.sum(diff ** 2))

```



To implement the back_propagate function:



1. Fill in the calculation of the hid_bias_grad by taking the sum of the hid_deriv along the 0th axis.
2. Fill in the calculation of the embed_to_hid_weights_grad by taking the dot product of hid_deriv and activations.embedding_layer.transpose().
3. Fill in the calculation of the output_bias_grad by taking the sum of loss_derivative along the 0th axis.

Example code:

```
SCSS Copy code  
  
def back_propagate(self, input_batch, activations, loss_derivative):  
    hid_deriv = np.dot(loss_derivative, self.params.hid_to_output_weights) \  
               * activations.hidden_layer * (1. - activations.hidden_layer)  
  
    hid_to_output_weights_grad = np.dot(loss_derivative.T, activations.hidden_layer)  
    hid_bias_grad = np.sum(hid_deriv, axis=0)  
  
    embed_deriv = np.dot(hid_deriv, self.params.embed_to_hid_weights)  
    embed_to_hid_weights_grad = np.dot(activations.embedding_layer.T, embed_deriv)  
    output_bias_grad = np.sum(loss_derivative, axis=0)  
  
    return Params(word_embedding_weights_grad, embed_to_hid_weights_grad, hid_to_out  
                  hid_bias_grad, output_bias_grad)
```

6.3

```
▶ # Run this to check if your implement gradients matches the finite difference within toler
  # Note: this may take a few minutes to go through all the checks
check_gradients()

<ipython-input-43-a832ca508c04>:216: DeprecationWarning: `np.int` is a deprecated alias fo
Deprecat
mask = np.zeros((batch_size, self.context_len), dtype=np.int)# Convert to one hot B x N,
The loss derivative looks OK.
The gradient for word_embedding_weights looks OK.
The gradient for embed_to_hid_weights looks OK.
The gradient for hid_to_output_weights looks OK.
The gradient for hid_bias looks OK.
The gradient for output_bias looks OK.

▶ # Run this to print out the gradients
print_gradients()

⇒ loss_derivative[46, 785] 0.7137561447745507
loss_derivative[46, 766] -0.9661570033238931
loss_derivative[5, 42] -0.0
loss_derivative[5, 31] 0.0

param_gradient.word_embedding_weights[27, 2] 0.0
param_gradient.word_embedding_weights[43, 3] 0.011596892511489458
param_gradient.word_embedding_weights[22, 4] -0.0222670623817297
param_gradient.word_embedding_weights[2, 5] 0.0

param_gradient.embed_to_hid_weights[10, 2] 0.3793257091930164
param_gradient.embed_to_hid_weights[15, 3] 0.01604516132110917
param_gradient.embed_to_hid_weights[30, 9] -0.4312854367997419
param_gradient.embed_to_hid_weights[35, 21] 0.06679896665436337

param_gradient.hid_bias[10] 0.023428803123345148
param_gradient.hid_bias[20] -0.024370452378874197

param_gradient.output_bias[0] 0.000970106146902794
param_gradient.output_bias[1] 0.16868946274763222
param_gradient.output_bias[2] 0.0051664774143909235
param_gradient.output_bias[3] 0.15096226471814364
<ipython-input-43-a832ca508c04>:216: DeprecationWarning: `np.int` is a deprecated alias fo
Deprecat
mask = np.zeros((batch_size, self.context_len), dtype=np.int)# Convert to one hot B x N,
```

7.1

```
[64] def weat_association_score(w, A, B, glove):
    """Given a target word w, the set of attribute words A and B,
    and the GloVe embeddings, returns the association score s(w, A, B).
    """
    ##### YOUR CODE HERE #####
    sum_A = 0
    sum_B = 0
    for a in A:
        sum_A += glove.similarity(a,w)
    for b in B:
        sum_B += glove.similarity(b,w)
    avg_A = sum_A/len(A)
    avg_B = sum_B/len(B)
    return avg_A-avg_B
#####
```

Use the following code to check your implementation:

```
[65] np.isclose(weat_association_score("programmer", A, B, glove), 0.019615129)
0s   True
```

Now, compute the WEAT association score for each element of `occupations` and the attribute sets A and B. Include the printed out association scores in your pdf.

```
[66] # TODO: Print out the weat association score for each occupation
0s   ##### YOUR CODE HERE #####
   for word in occupations:
Interrupt execution (%/Ctrl+M)
cell executed since last change   _association_score(word, A, B, glove)
                                association score for {word} is {ass_score})
#####
started at 11:20 PM (0 minutes ago)
D> The WEAT association score for programmer is 0.01961511862464249
The WEAT association score for engineer is 0.053647358901798725
The WEAT association score for scientist is 0.06795816496014595
The WEAT association score for nurse is -0.09486919268965721
The WEAT association score for teacher is -0.01893029361963272
The WEAT association score for librarian is -0.024141337256878614
```

chatGPT:

s Implement the following function, weat_association_score which computes the association of a word w with the attribute:

```
s(w,A,B)=mean_a=Acos(w,a)–mean_b=Bcos(w,b)
def weat_association_score(w, A, B, glove):
    """Given a target word w, the set of attribute words A and B,
    and the GloVe embeddings, returns the association score s(w, A, B).
    """

```

Here's the implementation:

python

 Copy code

```
import numpy as np

def weat_association_score(w, A, B, glove):
    mean_a = np.mean([glove[a] for a in A if a in glove], axis=0)
    mean_b = np.mean([glove[b] for b in B if b in glove], axis=0)
    w_vector = glove[w] if w in glove else None

    if w_vector is None:
        return None

    cos_sim_a = np.dot(w_vector, mean_a) / (np.linalg.norm(w_vector) * np.linalg.norm(mean_a))
    cos_sim_b = np.dot(w_vector, mean_b) / (np.linalg.norm(w_vector) * np.linalg.norm(mean_b))

    return cos_sim_a - cos_sim_b
```

This implementation takes the target word `w`, the set of attribute words `A`, the set of attribute words `B`, and the GloVe embeddings `glove` as input. It then computes the mean embedding for each of the sets `A` and `B`, taking into account only those words that have a vector representation in the GloVe embeddings. If the target word `w` does not have a vector representation, the function returns `None`. Finally, the cosine similarity is computed between the target word `w` and each of the mean embeddings of `A` and `B`, and the difference between these similarity scores is returned as the association score `s(w, A, B)`.

7.3.1

Find 1-word subsets of the original A and B that reverse the sign of the association score for at least some of the occupations

```
[55] # Original sets provided here for convenience - try commenting out all but one word from C and D
      # Two sets of gendered attribute words, C and D
      C = [ "man",
             "#male",
             "#he",
             "#boyish"
           ]
      D = [ "woman",
             "#female",
             "#she",
             "#girlish"
           ]

      # TODO: Print out the weat association score for each word in occupations, with regards to
      ##### YOUR CODE HERE #####
      for job in occupations:
          ass_score = weat_association_score(job, C, D, glove)
          print(f"The WEAT association score for {word} is {ass_score}")
      #####
```

The WEAT association score for librarian is 0.04651723802089691
The WEAT association score for librarian is 0.06365096569061279
The WEAT association score for librarian is 0.05343189835548401
The WEAT association score for librarian is -0.14363163709640503
The WEAT association score for librarian is -0.045757412910461426
The WEAT association score for librarian is -0.0909675657749176

7.3.2

The WHEAT association score will change if we change the words in attributed set because target word(occupational word) might co-occur with some attributed words more frequently than others, although these words might have similar meanings.