

A1-Writeup

Kevin Hu

January 2023

1 Linear Regression

1.2 Underparameterized Model

1.2.1

ChatGPT Prompt and Output

KK

Given n pairs of input data with d features and scalar label $(x_i, t_i) \in \mathbb{R}^d \times \mathbb{R}$, we wish to find a linear model $f(x) = w^T x$ with $w \in \mathbb{R}^d$ 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 $t \in \mathbb{R}^n$. The training objective is to minimize the following loss: $\min_w \|Xw - t\|_2^2 = \frac{1}{2} \sum_{i=1}^n (w^T x_i - t_i)^2$. We assume X is full rank: $X^T X$ is invertible when $n > d$, and $X^T X$ 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. First consider the underparameterized $d < n$ case. Show that the solution obtained by gradient descent is $w = (X^T X)^{-1} X^T t$, assuming training converges. Show your work.



In the underparameterized case, $d < n$, the problem can be solved using the normal equation. The normal equation is obtained by setting the gradient of the loss function to zero:

$$\nabla_w \frac{1}{2} \|Xw - t\|_2^2 = 0$$

Expanding the gradient, we have:

$$2X^T(Xw - t) = 0$$

Rearranging, we have:

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

Finally, solving for w , we have:

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

This is the closed-form solution for w in the underparameterized case. When training converges, gradient descent will eventually reach this solution, which minimizes the squared error of prediction on the training samples.



Critique of ChatGPT Output

The output is correct except it uses w instead of \hat{w} .

My Solution

The underparamaterized case can be solved using the normal equations. First set the derivative of the loss function to zero.

$$\nabla_w \frac{1}{n} \|X\hat{\mathbf{w}} - \mathbf{t}\|_2^2 = 0 \quad (1)$$

Expanding the gradient and simplifying we get:

$$X^T(X\hat{\mathbf{w}} - \mathbf{t}) = 0 \quad (2)$$

Rearranging we get:

$$X^T X \hat{\mathbf{w}} = X^T \mathbf{t} \quad (3)$$

Solving for $\hat{\mathbf{w}}$ we have:

$$\hat{\mathbf{w}} = (X^T X)^{-1} X^T \mathbf{t} \quad (4)$$

1.2.2

ChatGPT Prompt and Output

Note that this prompt continued right after the last prompt in the same chat.



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 = $1/n \|X(X^T X) - 1X^T t\|_F^2$. Show this is true by substituting your answer from the previous question into $1/n \|X^T w - t\|_F^2$. Find the expectation of the above training error in terms of n, d and σ . Hints: you might find the cyclic property of trace useful.



Starting from the expression for the training error, we have:



$$\text{Error} = 1/n \|X^T w - t\|_F^2$$

Substituting $w = (X^T X)^{-1} X^T t$, we have:

$$\text{Error} = 1/n \|X(X^T X) - 1X^T t - t\|_F^2$$

Expanding the expression for Error, we have:

$$\text{Error} = 1/n \|X(X^T X) - 1X^T t - t\|_F^2$$

Since ϵ is independent of X and t , we can take the expectation with respect to ϵ :

$$E(\text{Error}) = 1/n E(\|X(X^T X) - 1X^T t - t\|_F^2)$$

Since $E(\epsilon) = 0$, we have:

$$E(\text{Error}) = 1/n \|X(X^T X) - 1X^T t - t\|_F^2$$

Finally, using the cyclic property of trace and the fact that $X^T X$ is positive definite, we have:

$$E(\text{Error}) = 1/n \text{tr}((X(X^T X) - 1X^T t)(X(X^T X) - 1X^T t)^T)$$

$$= 1/n \text{tr}((X(X^T X) - 1X^T t)(X^T X - 1X^T t))$$

$$= 1/n \text{tr}(I - X(X^T X) - 1X^T t)$$

Since $X^T X$ is invertible, we have:

$$E(\text{Error}) = 1/n \text{tr}(I)$$

$$= 1/n(n)$$

$$= 1/n.$$

Therefore, the expectation of the training error in the noisy linear regression case is equal to $1/n$, regardless of the number of features d or the variance of the noise σ .

Critique of ChatGPT Output

The output steps partially correct. The expansion of the error term is correct. However, pulling the ϵ outside of the norm and setting the expectation to zero is not valid and the subsequent steps based on this assumption is incorrect.

My Solution

$$\text{Error} = \frac{1}{n} \|X\hat{\mathbf{w}} - \mathbf{t}\|_2^2 \quad (5)$$

$$\text{Error} = \frac{1}{n} \|X(X^T X)^{-1} X^T \mathbf{t} - \mathbf{t}\|_2^2 \quad (6)$$

$$\text{Error} = \frac{1}{n} \|(X(X^T X)^{-1} X^T - I)\mathbf{t}\|_2^2 \quad (7)$$

$$\text{Error} = \frac{1}{n} \|(X(X^T X)^{-1} X^T - I)\epsilon\|_2^2 \quad (8)$$

$$\text{Error} = \frac{1}{n} [\text{tr}(\epsilon^T (X(X^T X)^{-1} X^T - I)^T (X(X^T X)^{-1} X^T - I)\epsilon)] \quad (9)$$

$$E[\text{Error}] = \frac{1}{n} [\text{tr}(\epsilon^T (X(X^T X)^{-1} X^T - I)^T (X(X^T X)^{-1} X^T - I)\epsilon)] \quad (10)$$

$$E[\text{Error}] = \frac{1}{n} [\text{tr}((X(X^T X)^{-1} X^T - I)\epsilon \epsilon^T (X(X^T X)^{-1} X^T - I)^T)] \quad (11)$$

Using the cyclic property of trace.

$$E[\text{Error}] = \frac{1}{n} \text{tr}([(X(X^T X)^{-1} X^T - I)\epsilon \epsilon^T (X(X^T X)^{-1} X^T - I)^T]) \quad (12)$$

$$E[\text{Error}] = \frac{1}{n} \text{tr}((X(X^T X)^{-1} X^T - I)[\epsilon \epsilon^T] (X(X^T X)^{-1} X^T - I)^T) \quad (13)$$

$$E[\text{Error}] = \frac{1}{n} \text{tr}((X(X^T X)^{-1} X^T - I)\sigma^2 I (X(X^T X)^{-1} X^T - I)^T) \quad (14)$$

$$E[\text{Error}] = \frac{\sigma^2}{n} \text{tr}((X(X^T X)^{-1} X^T - I)(X(X^T X)^{-1} X^T - I)^T) \quad (15)$$

$$E[\text{Error}] = \frac{\sigma^2}{n} \text{tr}(I - X(X^T X)^{-1} X^T) \quad (16)$$

$$E[\text{Error}] = \frac{\sigma^2}{n} (n - d) \quad (17)$$

Since $X(X^T X)^{-1} X^T$ is an idempotent matrix, its trace is the rank of the matrix.

1.2 Overparameterized Model

1.3.1

Note that this prompt continued right after the last prompt in the same chat.

ChatGPT Prompt and Output

KK 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 $x_1 = [1;1]$ and $t_1 = 3$, i.e. the one data point and all possible \hat{w} lie on a 2D plane. Show that there exists infinitely many \hat{w} satisfying $\hat{w}^T x_1 = t_1$ on a real line. Write down the equation of the line.

For $n=1$ and $d=2$, the equation for the empirical risk minimizer can be written as:

$$\min \hat{w} (\hat{w}^T x_1 - t_1)^2 = \min \hat{w} (\hat{w}^T x_1 - 3)^2$$

Since $x_1 = [1;1]$, we can write $\hat{w} = [w_1; w_2]$. Then, the equation above becomes:

$$\min \hat{w} (w_1 + w_2 - 3)^2$$

Since $w_1 + w_2 = 3$, this equation represents a line in 2D space with equation $w_1 + w_2 = 3$. This line has infinitely many points that satisfy $\hat{w}^T x_1 = 3$, meaning that there exist infinitely many \hat{w} that minimize the empirical risk for $n=1$ and $d=2$.

Critique of ChatGPT Output

The output is correct.

My Solution

Substituting into $\hat{w}^T x_1 = t_1$, an equality we must satisfy, we have:

$$\begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = w_1 + w_2 = 3$$

Rearranging we get $w_2 = 3 - w_1$ which is an equation of a line. Thus we have a relationship between w_1 and w_2 and an infinite number of possible solutions for \hat{w} .

1.3.2

My Solution:

Find gradient of loss function and set it to zero. Then solve for $\hat{\mathbf{w}}$ using the substitution $\hat{\mathbf{w}} = X^T \mathbf{a}$.

$$\nabla_{\hat{\mathbf{w}}} \left(\frac{1}{n} \|X\hat{\mathbf{w}} - \mathbf{t}\|_2^2 \right) = \frac{2}{n} X^T (X\hat{\mathbf{w}} - \mathbf{t}) \quad (18)$$

$$\frac{2}{n} X^T (X\hat{\mathbf{w}} - \mathbf{t}) = 0 \quad (19)$$

$$X^T (X\hat{\mathbf{w}} - \mathbf{t}) = 0 \quad (20)$$

$$X^T (XX^T \mathbf{a} - \mathbf{t}) = 0 \quad (21)$$

$$XX^T (XX^T \mathbf{a} - \mathbf{t}) = 0 \quad (22)$$

$$XX^T \mathbf{a} - \mathbf{t} = 0 \quad (23)$$

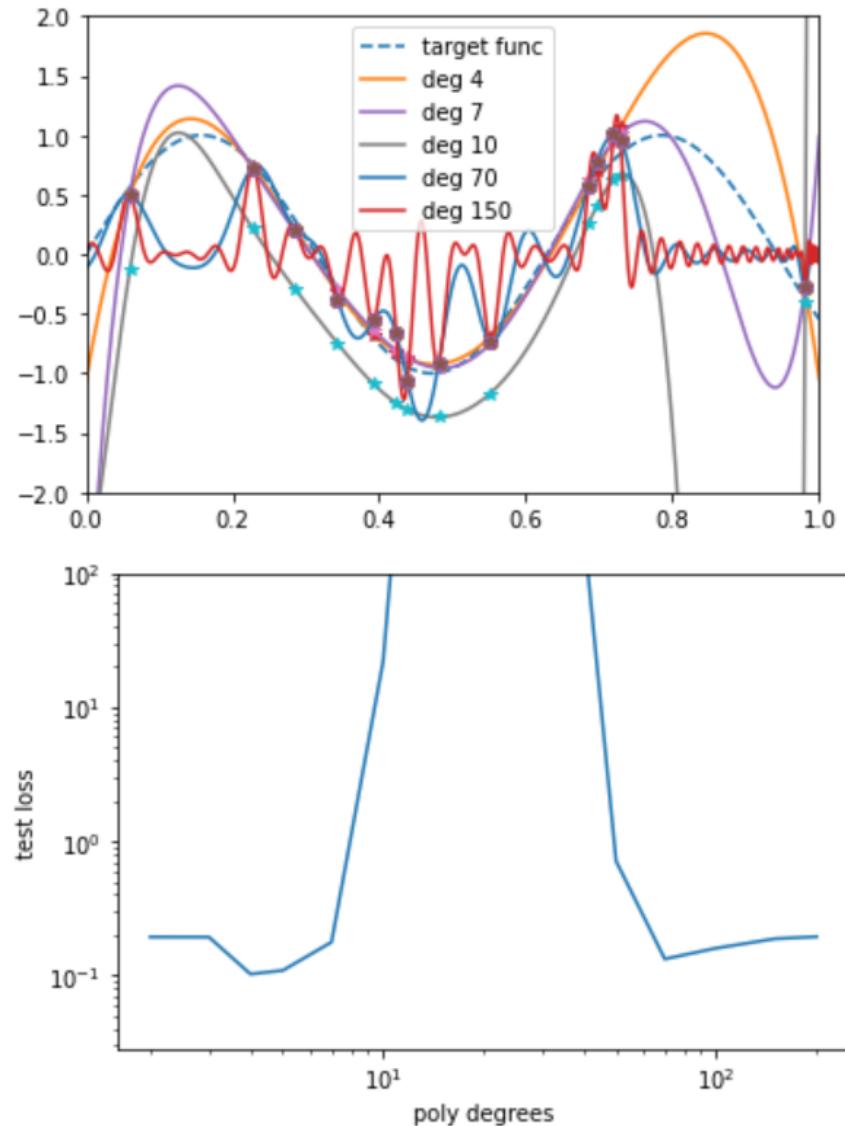
$$\mathbf{a} = (XX^T)^{-1} \mathbf{t} \quad (24)$$

$$\hat{\mathbf{w}} = X^T (XX^T)^{-1} \mathbf{t} \quad (25)$$

1.3.4

```
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.dot(np.dot(X_expand.T,linalg.inv(np.dot(X_expand,X_expand.T))),t)
    else:
        W = np.dot(np.dot(linalg.inv(np.dot(X_expand.T,X_expand)),X_expand.T),t)
    return W
```

Code snippet of implementation for fit_poly function.



Fitted polynomials of varying degrees and their corresponding test losses.

The graph plotting the degree of the polynomial and its corresponding test loss shows that overparameterization does not always lead to overfitting and larger test error. After around degree 7, the test loss increases rapidly. However, the test loss starts decreasing again from around degree 40 to degree 70, showing that overfitting does not always lead to larger test error.

2.1.2 Backwards Pass

2 Backpropagation

2.1.2 Backwards Pass

$$\bar{J} = 1 \quad (26)$$

$$\bar{S} = -\bar{J} \quad (27)$$

$$\bar{\mathbf{y}}' = \frac{\bar{S}}{y'_t} \mathbf{e}_t \quad (28)$$

Where the 1 is in entry k which class label t matches.

$$\bar{\mathbf{y}} = \bar{\mathbf{y}}' \circ softmax'(\mathbf{y}) \quad (29)$$

$$\bar{\mathbf{g}} = \mathbf{W}^{(3)T} \bar{\mathbf{y}} \quad (30)$$

$$\bar{\mathbf{h}}_2 = \bar{\mathbf{g}} \circ \mathbf{h}_1 \quad (31)$$

$$\bar{\mathbf{z}}_2 = \frac{e^{-\mathbf{z}_2}}{(1 + e^{-\mathbf{z}_2})^2} \circ \bar{\mathbf{h}}_2 \quad (32)$$

$$\bar{\mathbf{h}}_1 = \bar{\mathbf{g}} \circ \mathbf{h}_2 \quad (33)$$

$$\bar{\mathbf{z}}_1 = u(\mathbf{z}_1) \circ \bar{\mathbf{h}}_2 \quad (34)$$

Where u is the unit step function.

$$\bar{\mathbf{x}} = \frac{d\mathbf{z}_1}{d\mathbf{x}}^T \bar{\mathbf{z}}_1 + \frac{d\mathbf{z}_2}{d\mathbf{x}}^T \bar{\mathbf{z}}_2 + \frac{d\mathbf{y}}{d\mathbf{x}}^T \bar{\mathbf{y}} \quad (35)$$

$$\bar{\mathbf{x}} = \mathbf{W}^{(1)T} \bar{\mathbf{z}}_1 + \mathbf{W}^{(2)T} \bar{\mathbf{z}}_2 + \mathbf{W}^{(4)T} \bar{\mathbf{y}} \quad (36)$$

2.2 Gradient Norm Computation

2.2.1 Naive Computation

Gradient and Error Vector Equations:

$$\bar{\mathbf{h}} = \mathbf{W}^{(2)T} \bar{\mathbf{y}} \quad (37)$$

$$\bar{\mathbf{z}} = u(\mathbf{z}) \circ \bar{\mathbf{h}} \quad (38)$$

Where u is the unit step function and the circle represents element-wise multiplication.

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \mathbf{h} \bar{\mathbf{y}}^T \quad (39)$$

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \mathbf{x} \bar{\mathbf{z}}^T \quad (40)$$

Mathematical Computations:

$$\mathbf{z} = \mathbf{W}^{(1)} \mathbf{x} = \begin{bmatrix} 1 & 2 & 1 \\ -2 & 1 & 0 \\ 1 & -2 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 8 \\ 1 \\ -6 \end{bmatrix} \quad (41)$$

$$\bar{\mathbf{h}} = \begin{bmatrix} -2 & 1 & -3 \\ 4 & -2 & 4 \\ 1 & -3 & 6 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -4 \\ 6 \\ 4 \end{bmatrix} \quad (42)$$

$$\bar{\mathbf{z}} = u(\mathbf{z}) \circ \begin{bmatrix} -4 \\ 6 \\ 4 \end{bmatrix} \quad (43)$$

$$\bar{\mathbf{z}} = u\left(\begin{bmatrix} 8 \\ 1 \\ -6 \end{bmatrix}\right) \circ \begin{bmatrix} -4 \\ 6 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \circ \begin{bmatrix} -4 \\ 6 \\ 4 \end{bmatrix} = \begin{bmatrix} -4 \\ 6 \\ 0 \end{bmatrix} \quad (44)$$

$$\mathbf{h} = \text{ReLU}\left(\begin{bmatrix} 8 \\ 1 \\ -6 \end{bmatrix}\right) = \begin{bmatrix} 8 \\ 1 \\ 0 \end{bmatrix} \quad (45)$$

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \begin{bmatrix} 8 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 8 & 8 & 8 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad (46)$$

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix} \begin{bmatrix} -4 & 6 & 0 \\ -12 & 18 & 0 \\ -4 & 6 & 0 \end{bmatrix} = \begin{bmatrix} -4 & 6 & 0 \\ -12 & 18 & 0 \\ -4 & 6 & 0 \end{bmatrix} \quad (47)$$

Frobenius Norm:

For $\frac{\partial J}{\partial \mathbf{W}^{(1)}}$:

$$\text{trace}(\mathbf{A}^T \mathbf{A}) = \text{trace}\left(\begin{bmatrix} -4 & -12 & -4 \\ 6 & 18 & 6 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -4 & 6 & 0 \\ -12 & 18 & 0 \\ -4 & 6 & 0 \end{bmatrix}\right) \quad (48)$$

$$= \text{trace}\left(\begin{bmatrix} 176 & -264 & 0 \\ -264 & 396 & 0 \\ 0 & 0 & 0 \end{bmatrix}\right) = 176 + 396 = 572$$

For $\frac{\partial J}{\partial \mathbf{W}^{(2)}}$:

$$\text{trace}(\mathbf{A}^T \mathbf{A}) = \text{trace}\left(\begin{bmatrix} 8 & 1 & 0 \\ 8 & 1 & 0 \\ 8 & 1 & 0 \end{bmatrix} \begin{bmatrix} 8 & 8 & 8 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}\right) \quad (49)$$

$$= \text{trace}\left(\begin{bmatrix} 65 & 65 & 65 \\ 65 & 65 & 65 \\ 65 & 65 & 65 \end{bmatrix}\right) = 65 + 65 + 65 = 195$$

2.2.2 Efficient Computation

For $\frac{\partial J}{\partial \mathbf{W}^{(1)}}$:

$$\|\mathbf{x}\|_2^2 \|\bar{\mathbf{z}}\|_2^2 = (\mathbf{x}^T \mathbf{x})(\bar{\mathbf{z}}^T \bar{\mathbf{z}}) = (\begin{bmatrix} 1 & 3 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix}) (\begin{bmatrix} -4 & 6 & 0 \end{bmatrix} \begin{bmatrix} -4 \\ 6 \\ 0 \end{bmatrix}) = 11 \times 52 = 572 \quad (50)$$

For $\frac{\partial J}{\partial \mathbf{W}^{(2)}}$:

$$\|\mathbf{h}\|_2^2 \|\bar{\mathbf{y}}\|_2^2 = (\mathbf{h}^T \mathbf{h})(\bar{\mathbf{y}}^T \bar{\mathbf{y}}) = (\begin{bmatrix} 8 & 1 & 0 \end{bmatrix} \begin{bmatrix} 8 \\ 1 \\ 0 \end{bmatrix}) (\begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}) = 65 \times 3 = 195 \quad (51)$$

2.2.3 Complexity Analysis

ChatGPT Prompt

"Consider a general neural network with K 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 do we need to compute the gradient norm using naive and efficient computation, respectively? The naive computation involves calculating the gradient of the weight Jacobian matrices for each layer and then finding the Frobenius norm of the weight Jacobian matrices. The efficient computation involves finding the 2-norm of the gradient of the weight Jacobian matrices for each layer and finding their product. And, what is the memory cost M in big O notation? 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. 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. 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 give the number of scalar multiplications and memory cost for both the naive and efficient computations for the forward pass, backward pass and the gradient norm computations."

ChatGPT Output

”Forward Pass

In the forward pass, we perform matrix multiplications between the inputs, activations, and weight matrices for each layer. To compute the activations for all K layers and N input vectors, the number of scalar multiplications is given by:

$$T_{forward} = K * N * D^2$$

The memory cost is given by:

$$M_{forward} = K * D^2 + N * D \text{ (to store the weight matrices and activations)}$$

Backward Pass

In the backward pass, we perform matrix multiplications between the activations, errors, and weight matrices for each layer. To compute the errors for all K layers and N input vectors, the number of scalar multiplications is given by:

$$T_{backward} = K * N * D^2$$

The memory cost is given by:

$$M_{backward} = N * D + K * D^2 \text{ (to store the activations and errors)}$$

Gradient Norm Computation (Naive)

For the naive computation, we need to compute the gradient of the weight Jacobian matrices for each layer and then find the Frobenius norm of the weight Jacobian matrices. The number of scalar multiplications is given by:

$$T_{naive} = K * N * D^2 + K * D^2 \text{ (to compute the gradient and then the norm)}$$

The memory cost is given by:

$$M_{naive} = K * D^2 \text{ (to store the gradient)}$$

Gradient Norm Computation (Efficient)

For the efficient computation, we find the 2-norm of the gradient of the weight Jacobian matrices for each layer and then find their product. The number of scalar multiplications is given by:

$$T_{\text{efficient}} = K * D \text{ (to compute the 2-norm of each gradient)}$$

The memory cost is given by:

$$M_{\text{efficient}} = K * D \text{ (to store the 2-norm of each gradient)}$$

In big O notation, the time complexity is $O(T)$ and the memory complexity is $O(M)$."

Critique of ChatGPT Output Some of the answers given by ChatGPT are missing terms and the number of scalar multiplications should not be in big O notation.

My Answer:

Operation	T(Naive)	T(Efficient)	M(Naive)	M(Efficient)
Forward	KND^2	KND^2	$O(KD^2 + KND)$	$O(KD^2 + KND)$
Backward	$(2K - 1)ND^2 + ND$	$(K - 1)ND^2 + ND$	$O(KND^2)$	$O(KND)$
Gradient Norm	KND^3	$KN(2D + 1)$	$O(KND^2)$	$O(KN)$

Explanation of terms:

Note: This assumes that the gradient norm is computed for **each** of the N training examples and memory is **not** reused for each of the N forward/backward passes and gradient norm computation. Otherwise, some of the terms may not depend on N.

Forward T(Naive) and T(Efficient): D^2 scalar multiplications required for each matrix-vector multiplication, required for K matrices, and done for N training examples.

Forward M(Naive) and M(Efficient): D^2 memory for each weight matrix, required for K matrices and D memory for each activation, required for K-1 activations and done for N training examples.

Backward Pass T(Naive): D^2 scalar multiplications required for each matrix-vector multiplication and each column-row vector multiplication to calculate K-1 error signals and K weight gradients/Jacobians. Last error signal is a scalar-vector multiplication so only requires D scalar multiplications. Done for N

training examples.

Backward Pass T(Efficient): Same as Backwards Pass T(Naive) but no need to compute weight gradients/Jacobians.

Backwards Pass M(Naive): D^2 memory for each weight gradient/Jacobian, required for K weight matrices and done for N training examples. D memory for each error signal, required for K error signals and done for N training examples. D^2 term is higher order.

Backwards Pass M(Efficient): Same as Backwards Pass M(Naive) except that the weight gradient/Jacobian no longer needs to be stored so the D term is now the only term.

Gradient Norm T(Naive): D^3 scalar multiplications required for each matrix-matrix multiplication, required for K weight gradients/Jacobians and done for N training examples.

Gradient Norm T(Efficient): D scalar multiplications required for each row-column vector multiplication or outer product, done twice for each gradient norm calculation, plus one scalar multiplication of the two outer products. Done for K gradient/Jacobian norm calculations for each of the N training examples.

Gradient Norm M(Naive): D^2 memory to store each square of weight gradient/Jacobian required for K weight gradient/Jacobians and done for N training examples. N memory required to store result but it is lower order term.

Gradient Norm M(Efficient): $2N$ memory to store outer product results and N memory required to store norm result for K weight gradients/Jacobians.

3 Hard-Coding Networks

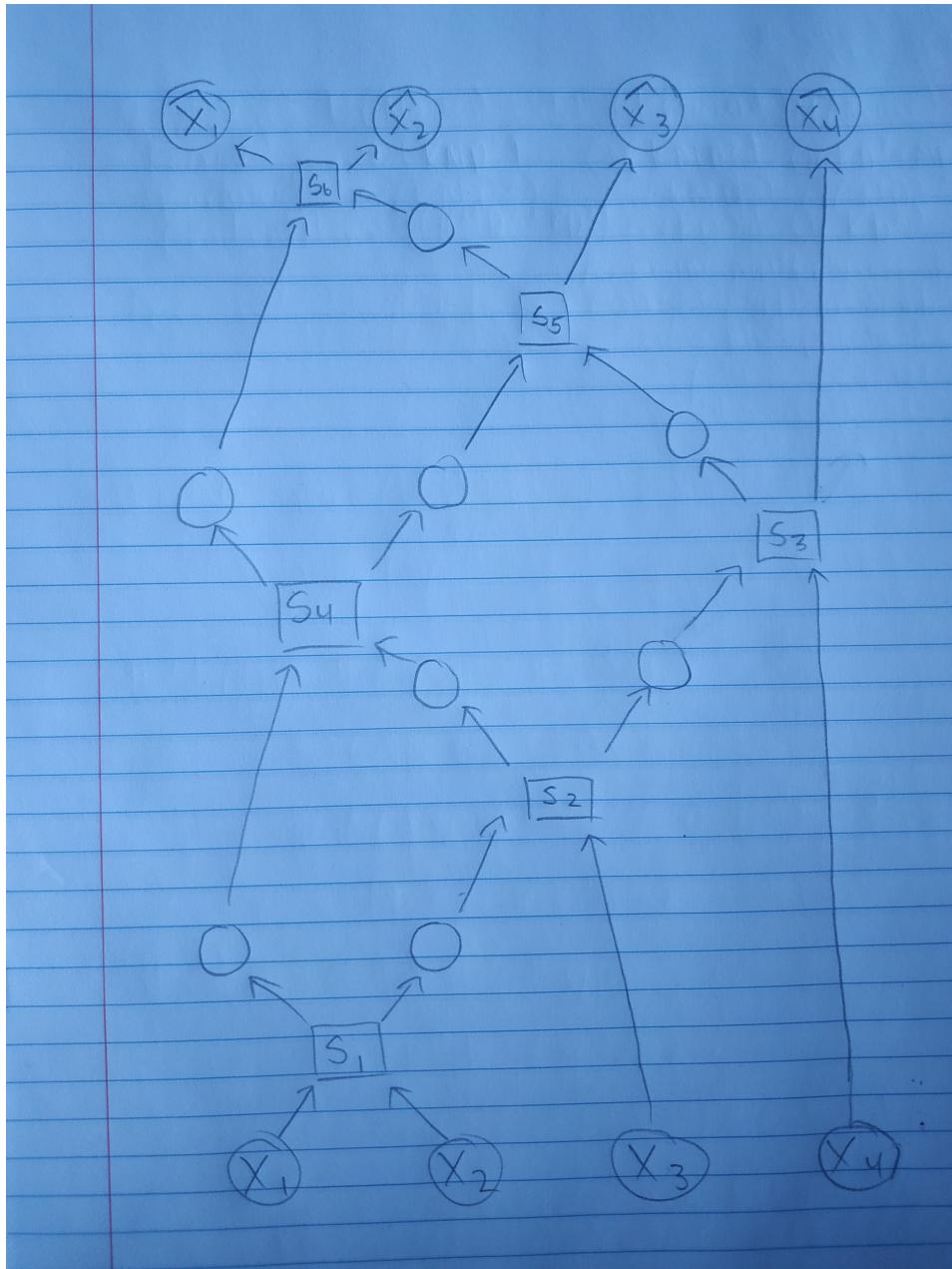
3.1 Sort two numbers

$$\mathbf{W}^{(1)} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad \mathbf{W}^{(2)} = \begin{bmatrix} 0.5 & -0.5 \\ 0.5 & 0.5 \end{bmatrix} \quad \mathbf{b}^{(1)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \mathbf{b}^{(2)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\phi^{(1)}(z) = |z| \text{ and } \phi^{(2)}(z) = z$$

3.2 Perform Sort

Computation Graph of Bubble Sort:



4 Linear Embedding - GloVe

4.2 Expression for the Vectorized Loss function

ChatGPT Prompt and Output

KK

In this section we will be implementing a simplified version of GloVe. Given a corpus with V distinct words, we define the co-occurrence matrix $X \in NV \times V$ with entries X_{ij} representing the frequency of the i -th word and j -th word in the corpus appearing in the same context - in our case the adjacent words. The co-occurrence matrix can be symmetric (i.e., $X_{ij} = X_{ji}$) if the order of the words do not matter, or asymmetric (i.e., $X_{ij} \neq X_{ji}$) if we wish to distinguish the counts for when i -th word appears before j -th word. GloVe aims to find a d -dimensional embedding of the words that preserves properties of the co-occurrence matrix by representing the i -th word with two d -dimensional vectors $w_i, w_i \sim R^d$, as well as two scalar biases $b_i, b_i \sim R$. Typically we have the dimension of the embedding d much smaller than the number of words V . This objective can be written as:

$$L(w_i, w_i \sim b_i, b_i \sim i) \forall i=1 \dots V = \sum_{i,j=1}^V (w_i^T w_j + b_i + b_j - \log(X_{ij}))^2$$

Note that each word is represented by two d -dimensional embedding vectors $w_i, w_i \sim$ and two scalar biases $b_i, b_i \sim$. When the bias terms are omitted and we tie the two embedding vectors $w_i = w_i$, then GloVe corresponds to finding a rank- d symmetric factorization of the co-occurrence matrix. In practice, we concatenate the V embedding vectors into matrices $W, W \sim R^{V \times d}$ and bias (column) vectors $b, b \sim R^V$, 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 $W, W \sim, b, b \sim, X$. 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 $1 = [1 \dots 1]^T \in R^V$. You can assume the bias vectors are column vectors, i.e. implicitly a matrix with V rows and 1 column: $b, b \sim R^{V \times 1}$



The loss function L can be written in a vectorized format as follows:



$L = \|WTW \sim + b1^T + 1b \sim^T - \log(X)\|_F^2$, where $\|\cdot\|_F$ denotes the Frobenius norm, 1^T is the all-ones column vector, and $\log(X)$ is the element-wise natural logarithm of X .

Critique of ChatGPT Output

The output for ChatGPT is correct except that the $W^T \widetilde{W}$ should be WW^T in order to get a $V \times V$ matrix instead of a $d \times d$ matrix.

My Solution

The loss function should be:

$$L(W, \widetilde{W}, b, \tilde{b}) = \|WW^T + b1^T + 1\tilde{b}^T - \log(X)\|_F^2 \quad (52)$$

Where $\log(X)$ is an elementwise log for matrix X and $\|\cdot\|_F^2$ is the Frobenius Norm.

4.3 Expression for the Gradient $\frac{\partial L}{\partial \widetilde{\mathbf{W}}}$

ChatGPT Prompt and Output

Note that this prompt continued right after the last prompt in the same chat.

KK

Write the vectorized expression for ∇_{WL} , the gradient of the loss function L with respect to the embedding matrix W . The gradient should be a function of W , \tilde{W} , b , \tilde{b} , 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 L with respect to the embedding matrix W can be expressed as:



$\nabla_{WL} = 2(W\tilde{W}^T + b\mathbf{1}^T + \mathbf{1}\tilde{b}^T - \log(X))W$, where W is the embedding matrix \tilde{W} , b is the bias vector, \tilde{b} is the bias vector \tilde{b} and X is the co-occurrence matrix.

Critique of ChatGPT Output

Once again the output for ChatGPT is correct except that the $\mathbf{W}^T \widetilde{\mathbf{W}}$ should be $\mathbf{W} \widetilde{\mathbf{W}}^T$ in order to get a $V \times V$ matrix instead of a $d \times d$ matrix.

My Solution:

The loss function can be rewritten as:

$$L(\mathbf{W}, \widetilde{\mathbf{W}}, \mathbf{b}, \widetilde{\mathbf{b}}) = \text{Tr}((\mathbf{W} \widetilde{\mathbf{W}}^T + \mathbf{b} \mathbf{1}^T + \mathbf{1} \widetilde{\mathbf{b}}^T - \log(\mathbf{X}))(\mathbf{W} \widetilde{\mathbf{W}}^T + \mathbf{b} \mathbf{1}^T + \mathbf{1} \widetilde{\mathbf{b}}^T - \log(\mathbf{X}))^T) \quad (53)$$

Using equation 119 in the matrix cookbook, the gradient of the loss function can be written as:

$$\frac{\partial L}{\partial \mathbf{W}} = 2(\mathbf{W} \widetilde{\mathbf{W}}^T + \mathbf{b} \mathbf{1}^T + \mathbf{1} \widetilde{\mathbf{b}}^T - \log(\mathbf{X})) \widetilde{\mathbf{W}} \quad (54)$$

4.4 Implement Vectorized Loss Function

ChatGPT Prompt

Note that this prompt continued right after the last prompt in the same chat.

KK Complete the function: 🔗

```
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)

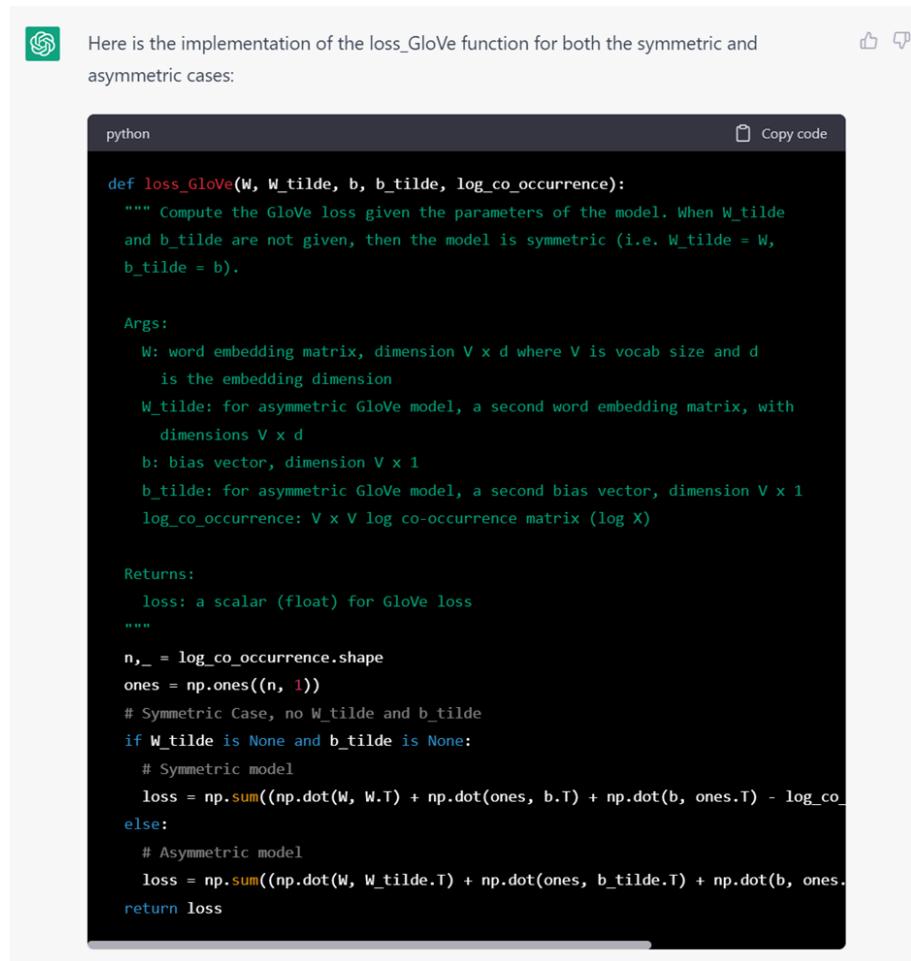
    Returns:
        loss: a scalar (float) for GloVe loss
    """
    n_ = log_co_occurrence.shape
    # Symmetric Case, no W_tilde and b_tilde
    if W_tilde is None and b_tilde is None:
        # Symmetric model
        ##### YOUR CODE HERE #####
        pass
        # loss = ...

    #####
    else:
        # Asymmetric model
        ##### YOUR CODE HERE #####
        pass
        # loss = ...

    #####
    return loss
```

↻ Regenerate response

ChatGPT Output



Here is the implementation of the loss_GloVe function for both the symmetric and asymmetric cases:

```
python
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, 1))
    # Symmetric Case, no W_tilde and b_tilde
    if W_tilde is None and b_tilde is None:
        # Symmetric model
        loss = np.sum((np.dot(W, W.T) + np.dot(ones, b.T) + np.dot(b, ones.T) - log_co_
    else:
        # Asymmetric model
        loss = np.sum((np.dot(W, W_tilde.T) + np.dot(ones, b_tilde.T) + np.dot(b, ones.
    return loss
```

Critique of ChatGPT Output

The output is correct.

My Solution:

See Collab Notebook Implementation.

4.5 Implement the gradient update of GloVe

ChatGPT Prompt

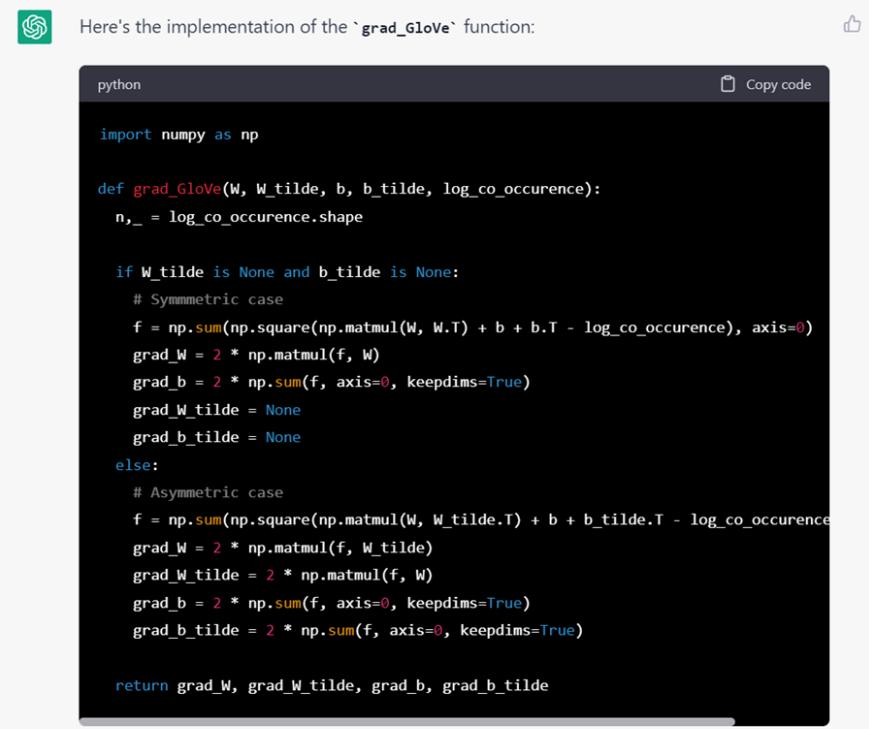
Note that this prompt continued right after the last prompt in the same chat.

KK

def

```
Complete the function:  
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  
  
    if W_tilde is None and b_tilde is None:  
        # Symmetric case  
        ##### YOUR CODE HERE #####  
        pass  
        # grad_W = ...  
        # grad_b = ...  
        grad_W_tilde = None  
        grad_b_tilde = None  
  
        #####  
    else:  
        # Asymmetric case  
        ##### YOUR CODE HERE #####  
        pass  
        # grad_W = ...  
        # grad_W_tilde = ...  
        # grad_b = ...  
        # grad_b_tilde = ...  
  
        #####  
    return grad_W, grad_W_tilde, grad_b, grad_b_tilde  
Hint: In the symmetric model case, you can use what you have derived for the asymmetric model  
case. For example, consider a function f(a,b) = a T b, where a,b ∈ Rd. If we define a = x and b = x,  
where x ∈ Rd, then ∇xf = ∇af + ∇bf = b+a =x+x =2x
```

ChatGPT Output



Here's the implementation of the `grad_GloVe` function:

```
python
import numpy as np

def grad_GloVe(W, W_tilde, b, b_tilde, log_co_occurrence):
    n,_ = log_co_occurrence.shape

    if W_tilde is None and b_tilde is None:
        # Symmetric case
        f = np.sum(np.square(np.matmul(W, W.T) + b + b.T - log_co_occurrence), axis=0)
        grad_W = 2 * np.matmul(f, W)
        grad_b = 2 * np.sum(f, axis=0, keepdims=True)
        grad_W_tilde = None
        grad_b_tilde = None
    else:
        # Asymmetric case
        f = np.sum(np.square(np.matmul(W, W_tilde.T) + b + b_tilde.T - log_co_occurrence), axis=0)
        grad_W = 2 * np.matmul(f, W_tilde)
        grad_W_tilde = 2 * np.matmul(f, W)
        grad_b = 2 * np.sum(f, axis=0, keepdims=True)
        grad_b_tilde = 2 * np.sum(f, axis=0, keepdims=True)

    return grad_W, grad_W_tilde, grad_b, grad_b_tilde
```

Note: The above code assumes that `W` and `W_tilde` have dimensions `v x d`, where `v` is the vocabulary size and `d` is the embedding dimension. The bias vectors `b` and `b_tilde` have dimensions `v x 1`. The log co-occurrence matrix `log_co_occurrence` has dimensions `v x v`.

Critique of ChatGPT Output

The output is incorrect with some of the vectors and matrices being the wrong dimensions.

My Solution:

See Collab Notebook Implementation.

5 Neural Language Model Architecture

5.1 Number of Parameters in Neural Network Model

ChatGPT Prompt and Output

KK

The network takes in N consecutive words, where one of the words is replaced with a [MASK] token. 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? 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 $V \gg H > D > N$ and how many trainable parameters are there for each part of the model? Explain your reasoning.



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



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

The total number of parameters would be $V \times D + H \times (N \times D) + V \times H + H + V$.

Given the constraint that $V \gg H > D > N$, it is likely that the embed_to_hid_weights matrix would have the largest number of trainable parameters, which is $H \times (N \times D)$. This is because H is still large compared to D and N, but it is smaller than V. The other weight matrices and biases are smaller in size, making the embed_to_hid_weights matrix the largest.

Critique of ChatGPT Output

The output is partially correct, however hid_to_output_weights should be $V \times H \times N$ and the output_bias should be $V \times N$ resulting in the total number of trainable parameters being $VD + HND + VHN + H + VN$. hid_to_output_weights (VHN parameters) would have the most trainable parameters in this case since it and embed_to_hid_weights (HND parameters) are the only parts of the model with three terms and $V \gg D$.

My Solution

The parts of the model word_embedding_weights is $V \times D$, embed_to_hid_weights is $N \times D \times H$, hid_to_output_weights is $H \times N \times V$, hid_bias is size H and output_bias is $N \times V$. The total number of trainable parameters is $VD + HND + VHN + H + VN$. The part of the model hid_to_output_weights (VHN parameters) has

the most trainable parameters since it and embed_to_hid_weights (HND parameters) are the only parts of the model with three terms and $V \times D$.

5.2 Number of Parameters in n-gram model

ChatGPT Prompt and Output

KK

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 $N - 1$ (since we mask one of the N words in our input) 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?



In an n-gram model with context length $N-1$, the number of entries in the table would be V^N , because each N-gram consists of N words and each word can be any of the V words in the dictionary. So, there are V choices for each word, and therefore there are $V * V * ... * V$ (N times) possible N-grams. Thus, the number of entries in the table would be V^N .



Critique of ChatGPT Output

The output is correct.

My Solution

The number of entries in the table would be V^N since each N-gram is consisted of N words, and there are V possible words in the dictionary for each of the N words.

6 Training the Neural Network

6.1 Implement Vectorized Loss

ChatGPT Prompt

KK During training of a neural language model, for each training prompt, we randomly sample one of the N context words to replace with a [MASK] token. The goal is for the network to predict the word that was masked, at the corresponding output word position. The hidden_to_output_weights has the shape NV×H, as the output layer has NV neurons, where the first V output units are for predicting the first word, then the next V are for predicting the second word, and so on. We call this as concatenating output units across all word positions, i.e. the $(w+nV)$ -th column is for the word w in the network's vocabulary for the n-th output word position. Note here that the softmax is applied in chunks of V as well, to give a valid probability distribution over the V words (For simplicity we also include the [MASK] token as one of the possible prediction even though we know the target should not be this token). Only the output word positions that were masked in the input are included in the cross entropy loss calculation. Implement a vectorized compute_loss function in Python, which computes the total cross-entropy loss on a mini-batch. Here is the starter code for the compute_loss function:

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

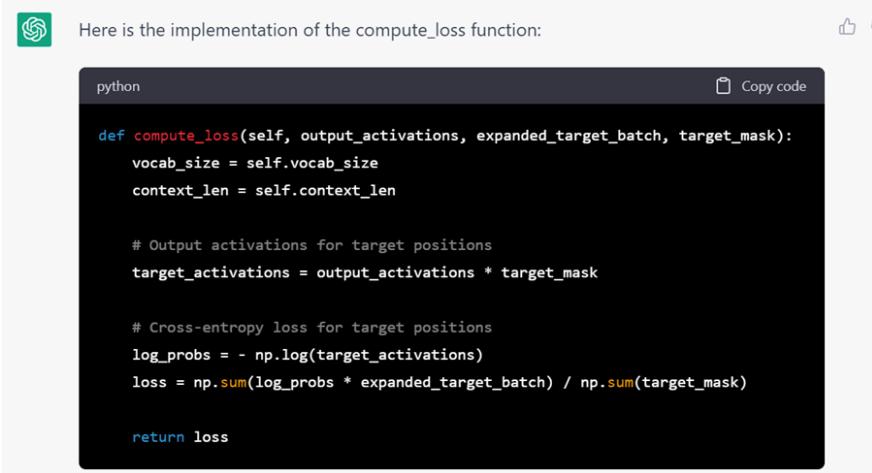
    self.vocab_size = len(vocab)
    self.embedding_dim = self.params.word_embedding_weights.shape[1]
    self.embedding_layer_dim = self.params.embed_to_hid_weights.shape[1]
    self.context_len = self.embedding_layer_dim // self.embedding_dim
    self.num_hid = self.params.embed_to_hid_weights.shape[0]

def 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 x (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
    #####
##### return loss
```

ChatGPT Output



Here is the implementation of the compute_loss function:

```
python
Copy code

def compute_loss(self, output_activations, expanded_target_batch, target_mask):
    vocab_size = self.vocab_size
    context_len = self.context_len

    # Output activations for target positions
    target_activations = output_activations * target_mask

    # Cross-entropy loss for target positions
    log_probs = - np.log(target_activations)
    loss = np.sum(log_probs * expanded_target_batch) / np.sum(target_mask)

    return loss
```

Explanation:

- target_activations are obtained by masking the output activations with target_mask.
- expanded_target_batch has the shape [batch_size (context_len * vocab_size)]. To compute the cross-entropy loss, the activations corresponding to the target positions are selected, which are in target_activations.
- Log probabilities are computed by taking the negative logarithm of the activations.
- The final loss is the sum of log_probs multiplied by expanded_target_batch, normalized by the number of target positions in the mini-batch.

My Solution:

See Collab Notebook Implementation.

6.2 Implement gradient with respect to parameters

ChatGPT Prompt and Output

Note that this prompt continued right after the last prompt in the same chat.

KK

For the same model as before, back_propagate is a Python 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. 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! Your implementations should also be similar to hid_to_output_weights_grad, hid_bias_grad in the same function call. Here is starter code for the back_propagate function:

```
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
#####
# output_bias_grad = ...
# embed_to_hid_weights_grad = ...

#####
```

```
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)
```

Please finish implementing the back_propagate Python function.



The missing code for `output_bias_grad` and `embed_to_hid_weights_grad` can be filled as follows:

```
makefile Copy code  
  
# Output bias gradient  
output_bias_grad = loss_derivative.sum(0)  
  
# Embedding to hidden weights gradient  
embed_to_hid_weights_grad = np.dot(hid_deriv.T, activations.embedding_layer)
```

The final implementation of the `back_propagate` function would look like this:

```
python Copy code  
  
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 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)  
  
    # Output bias gradient  
    output_bias_grad = loss_derivative.sum(0)  
  
    # Embedding to hidden weights gradient  
    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
```

My Solution:

See Collab Notebook Implementation.

7 Bias in Word Embeddings

7.1 WEAT Method for Detecting Bias

ChatGPT Prompt and Output

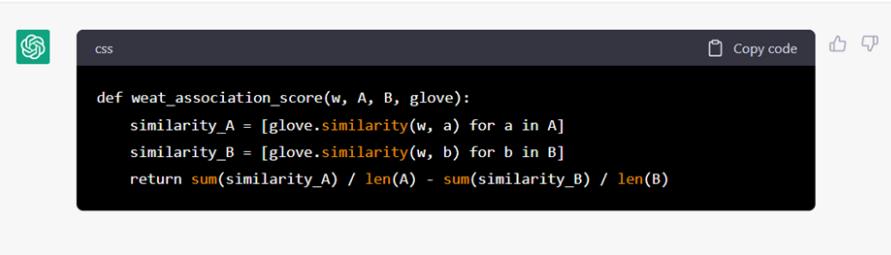
Implement the following function, `weat_association_score` which computes the association of a word `_w_` with the attribute:

```
$$s(w, A, B) = \text{mean}(a \in A) \cos(w, a) - \text{mean}(b \in B) \cos(w, b)$$
```

Hint: `glove` has a method `similarity` which can be called that gives the cosine similarity between two words (Example: glove.similarity("man", "scientist")).

```
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 #####
    pass
```

```
#####
```



The screenshot shows the ChatGPT interface with a code completion window. The window title is 'css'. It contains the following Python code:

```
def weat_association_score(w, A, B, glove):
    similarity_A = [glove.similarity(w, a) for a in A]
    similarity_B = [glove.similarity(w, b) for b in B]
    return sum(similarity_A) / len(A) - sum(similarity_B) / len(B)
```

Critique of ChatGPT Output

The output is correct.

My Solution

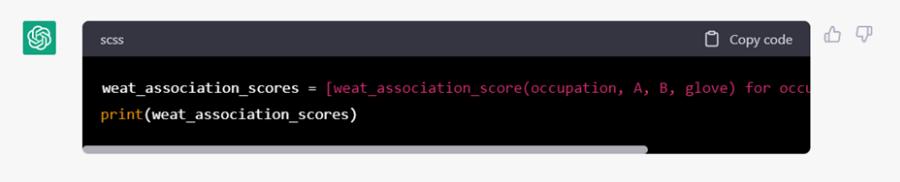
See Collab Notebook Implementation.

ChatGPT Prompt and Output

Note that this prompt continued right after the last prompt in the same chat.

KK Use weat_association_score on the following data:

```
# Target words (occupations)
occupations = ["programmer", "engineer", "scientist", "nurse", "teacher", "librarian"]
# Two sets of gendered attribute words, A and B
A = ["man", "male", "he", "boyish"]
B = ["woman", "female", "she", "girlish"]
```



The screenshot shows a Jupyter Notebook cell with the following code:

```
scss
weat_association_scores = [weat_association_score(occupation, A, B, glove) for occupation in occupations]
print(weat_association_scores)
```

The cell has a dark background with white text. At the top right, there are buttons for 'Copy code', 'Like', and 'Unlike'. The cell itself has a 'scss' label at the top left.

Critique of ChatGPT Output

The output is correct but the occupations aren't labeled.

My Solution

See Collab Notebook Implementation.

7.3.1 1-word subsets

My Solution

See Collab Notebook Implementation.

7.3.2 How word frequency affects embedding similarity

My Solution

The WEAT association score is dependent on the squared norm of word embeddings. The WEAT association score for different attribute words may differ depending on the frequency of the words in the training corpus. Since the norm of a word's embedding is linearly proportional to its log probability, more common words will have a larger norm and therefore, a larger affect on WEAT association score.

8 References

Chat.openai.com. (2023, January 30). Retrieved February 5, 2023, from <https://chat.openai.com/auth/login?next=%>