

CSC413 A1 Writeup

Richard Yan

January 2023

Program Trace for EC:

1.3.2:

R Now, let's generalize the previous 2D case to the general $d > n$ case. Show that gradient descent from zero initialization (i.e. $\hat{b}w(0) = 0$) finds a unique minimizer if it converges.

In the general case where $d > n$, the data matrix X is tall and skinny and thus has a null space. This means that there exists a non-zero vector \mathbf{v} such that $X \mathbf{v} = 0$. Therefore, if we initialize the parameter vector $\hat{b}w$ to zero, it will lie in the null space of X , and the gradient of the loss function with respect to $\hat{b}w$ will be proportional to \mathbf{v} .

Now, at each iteration of gradient descent, we update the parameter vector by subtracting the gradient of the loss function multiplied by a learning rate α :

$$\hat{b}w(t+1) = \hat{b}w(t) - \alpha \nabla L(\hat{b}w(t))$$

As long as the learning rate α is positive and non-zero, the updated parameter vector $\hat{b}w(t+1)$ will not be in the null space of X , since the gradient of the loss function is proportional to \mathbf{v} , which is not zero.

Therefore, at each iteration, the parameter vector is projected out of the null space of X , and as the loss function is convex, the parameter vector will converge to a unique minimum, assuming the convergence of the gradient descent.

In summary, when $d > n$ and the gradient descent is initialized with zero, the parameter vector will be moved out of the null space of X at each iteration and converge to a unique minimizer, assuming that the gradient descent converges.

R Assume $d > n$, and since X is full rank hence X^{\top} is invertible. Show that the solution by gradient decent is $\hat{b} = X^{\top}(X X^{\top})^{-1} b$.

You can assume that the gradient is spanned by the rows of X and write \hat{b} :

$= X^{\top} \mathbf{a}$ for some \mathbf{a} in $(R)^n$.

Take gradient of the loss function with respect to \hat{b} first, then substitute $\hat{b} = X^{\top} \mathbf{a}$.

We can start with the gradient of the loss function with respect to \hat{b} :



$$\frac{\partial}{\partial \hat{b}} \frac{1}{n} \sum_{i=1}^n \|X \hat{b} - b\|^2 = \frac{2}{n} X^{\top} (X \hat{b} - b)$$

Now we substitute $\hat{b} = X^{\top} \mathbf{a}$:

$$\frac{\partial}{\partial \hat{b}} \frac{1}{n} \sum_{i=1}^n \|X X^{\top} \mathbf{a} - b\|^2 = \frac{2}{n} X^{\top} (X X^{\top} \mathbf{a} - b)$$

Since the gradient is spanned by the rows of X , the solution \mathbf{a} that minimizes the loss function must satisfy the equation above.

$$X X^{\top} \mathbf{a} = b$$

Solving for \mathbf{a} gives:

$$\mathbf{a} = (X X^{\top})^{-1} b$$

Therefore, the solution by gradient decent is:

$$\hat{b} = X^{\top} \mathbf{a} = X^{\top} (X X^{\top})^{-1} b$$

It's worth to mention that this solution is also known as the "normal equation" and it's a closed-form solution to the linear regression problem that can be computed efficiently using matrix inversion.

1 Linear Regression

1.1 Deriving the Gradient

The gradient of the loss w.r.t. $\hat{\mathbf{w}}$ is:

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

(From ChatGPT Jan 9 Ver.)

Program Trace:

R Given \$n\$ pairs of input data with \$d\$ features and scalar label \$\\{(bx_{-i}, t_{-i})|i \\in [1, n] \\}^{(d)}\$, we wish to find a linear model \$f(bx) = \\{\\hat{w}_j \\}^{(d)} \\cdot bx_j + b\$ that minimizes the squared error of prediction on the training samples defined below.

For concise notation, denote the data matrix \$X \\in \\mathbb{R}^{n \\times d}\$ and the corresponding label vector \$\\{t_i| i \\in [1, n]\\}\$.

The training objective is to minimize the following loss:

```
begin{align*}
&underset{\\{\\hat{w}\\}}{min} \\{frac{1}{n} \\sum_{i=1}^n (\\{\\hat{w}\\} \\cdot bx_{-i} - t_{-i})^2}
&= \\
&underset{\\{\\hat{w}\\}}{min} \\{frac{1}{n} \\|X\\{\\hat{w}\\} - \\{t\\}\\|_2^2.
end{align*}
```

We assume \$X\$ is full rank: \$X^\top X\$ is invertible when \$n > d\$, and \$X^\top X\$ is invertible otherwise.

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

The gradient of the loss with respect to the learned parameter vector \$\\{\\hat{w}\\}\$ is given by:

$$\\{frac{\\{partial\\}}{\\{partial\\} \\{\\hat{w}\\}}} \\{frac{1}{n} \\|X\\{\\hat{w}\\} - \\{t\\}\\|_2^2 = \\{frac{2}{n} X^\top (X\\{\\hat{w}\\} - \\{t\\})\\}$$

1.2 Underparametrized Model

1.2.1

Answer from ChatGPT (Jan 9 ver.) and trace (dialogue continued from Q1.1):

R Consider $d < n$. Show that the solution obtained by gradient descent is $\hat{w} = (X^T X)^{-1} X^T b$, assuming training converges.

The gradient of the loss function is given by:

$$\frac{\partial}{\partial w} \frac{1}{n} \sum_{i=1}^n (y_i - w^T x_i)^2 = \frac{2}{n} X^T (Xw - b)$$

The gradient descent update rule is given by:

$$w := w - \alpha \frac{\partial}{\partial w} \frac{1}{n} \sum_{i=1}^n (y_i - w^T x_i)^2 = w - \alpha \frac{2}{n} X^T (Xw - b)$$

Plugging in the gradient of the loss function, we get:

$$\hat{w} := \hat{w} - \alpha \frac{2}{n} X^T (X\hat{w} - b)$$

Rearranging the terms, we get:

$$\hat{w} := (I - \alpha \frac{2}{n} X^T X) \hat{w} + \alpha \frac{2}{n} X^T b$$

This is the update rule for each iteration.
When the learning rate α is small enough, and the number of iterations is large enough, the update rule will converge and the solution will be $(I - \alpha \frac{2}{n} X^T X)^{-1} \alpha \frac{2}{n} X^T b = (X^T X)^{-1} X^T b$.
It is also known as the closed-form solution to Linear Regression.

Answer is correct, and it shows every step of gradient descent and the update rule of it.

Answer:

The gradient of the loss function is given by: $\frac{\partial}{\partial \hat{w}} \frac{1}{n} \|X\hat{w} - b\|_2^2 = \frac{2}{n} X^T (X\hat{w} - b)$

The gradient descent update rule is given by: $\hat{w} \leftarrow \hat{w} - \alpha \frac{\partial}{\partial \hat{w}} \frac{1}{n} \|X\hat{w} - b\|_2^2$

Plugging in the gradient of the loss function, we get: $\hat{w} \leftarrow \hat{w} - \alpha \frac{2}{n} X^T (X\hat{w} - b)$

Rearranging the terms, we get: $\hat{w} \leftarrow (I - \alpha \frac{2}{n} X^T X) \hat{w} + \alpha \frac{2}{n} X^T b$

This is the update rule for each iteration. When the learning rate α is small enough, and the number of iterations is large enough, the update rule will converge and the solution will be $(I - \alpha \frac{2}{n} X^T X)^{-1} \alpha \frac{2}{n} X^T b = (X^T X)^{-1} X^T b$. It is also known as the closed-form solution to Linear Regression.

1.2.2

Answer from ChatGPT (Jan 9 ver.) and trace (dialogue continued from above):

R Now consider the case of noisy linear regression. The training labels $\$t_{-i} = \{bx^i + \text{top}_i\}_{i=1}^n$ are generated by a ground truth linear target function, where the noise term, $\$epsilon_i$, is generated independently with zero mean and variance $\$sigma^2$. The final training error can be derived as a function of $\$X$ and $\$b$ by summing up $\$epsilon_i$'s, as:

```

\begin{align*}
& \text{(textit{Error})} = && \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^{m-1} \sum_{k=j+1}^m \left| \hat{y}_{ijk} - y_{ijk} \right|^2 \\
& \approx && \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^{m-1} \sum_{k=j+1}^m \left( \hat{y}_{ijk} - y_{ijk} \right)^2 \\
& \approx && \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^{m-1} \sum_{k=j+1}^m \left( \hat{y}_{ijk} - \bar{y}_{ijk} + \bar{y}_{ijk} - y_{ijk} \right)^2 \\
& = && \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^{m-1} \sum_{k=j+1}^m \left( \hat{y}_{ijk} - \bar{y}_{ijk} \right)^2 + \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^{m-1} \sum_{k=j+1}^m \left( \bar{y}_{ijk} - y_{ijk} \right)^2 \\
& = && \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^{m-1} \sum_{k=j+1}^m \left( \hat{y}_{ijk} - \bar{y}_{ijk} \right)^2 + \text{error} \\
& \approx && \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^{m-1} \sum_{k=j+1}^m \left( \hat{y}_{ijk} - \bar{y}_{ijk} \right)^2
\end{align*}

```

We can start by substituting $\hat{b}w = (X^{\top} X)^{-1} X^{\top} b$ into $\frac{1}{n} \|X \hat{b} - b\|_2^2$:

The last step is true because $\$(X(X^{\mid}top\ X)^{\{-1\}}\ X^{\mid}top - l)X\mid bw^{*} = 0\$$

This shows that the final training error is the same as the formula given.

R Find the expectation of the above final training error in terms of n , d and σ .

The expectation of the final training error is the expected value of the square of the norm of $\$(X^{\wedge}[\text{top } X] - \{1\} X^{\wedge}[\text{top } - 1])\$$. We can find this by noting that $\$|\boldsymbol{\text{boldsymbol}}[\text{epsilon}]\$$ is a zero-mean, independent, and identically distributed (iid) random variable with variance $\$|\sigma|^2\$$. Therefore, the expectation of the final training error is:

Where tr denotes the trace of a matrix

Now

$$\$text{tr}\{(-X(X^top X)^{-1} X^top) = n - d\$$$

therefore

$$\text{Therefore } \mathbb{E}[\text{Error}] = \frac{\sigma^2}{n}(1 - \frac{d}{n})$$

This is the expected final training error in terms of n , d , and σ .

Answer can be improved by little bit, but overall on the right track and give correct answers. The deriving part could be improved.

Answer:

We can start by substituting $\hat{\mathbf{w}} = (X^\top X)^{-1} X^\top \mathbf{t}$ into $\frac{1}{n} \|X\hat{\mathbf{w}} - \mathbf{t}\|_2^2$:

$$\begin{aligned}\frac{1}{n} \|X(X^\top X)^{-1} X^\top \mathbf{t} - \mathbf{t}\|_2^2 &= \frac{1}{n} \|(X(X^\top X)^{-1} X^\top - I)\mathbf{t}\|_2^2 \\ &= \frac{1}{n} \|(X(X^\top X)^{-1} X^\top - I)(X\mathbf{w}^* + \boldsymbol{\epsilon})\|_2^2 \\ &= \frac{1}{n} \|(X(X^\top X)^{-1} X^\top - I)\boldsymbol{\epsilon}\|_2^2\end{aligned}$$

The last step is true because

$$\begin{aligned}(X(X^\top X)^{-1} X^\top - I)X\mathbf{w}^* &= X(X^\top X)^{-1} X^\top X\mathbf{w}^* - IX\mathbf{w}^* \\ &= X\mathbf{w}^* - X\mathbf{w}^* = 0\end{aligned}$$

This shows that the final training error is the same as the formula given.

The expectation of the final training error is the expected value of the square of the norm of $(X(X^\top X)^{-1} X^\top - I)\boldsymbol{\epsilon}$. We can find this by noting that $\boldsymbol{\epsilon}$ is a zero-mean, independent, and identically distributed (iid) random variable with variance σ^2 . Therefore, the expectation of the final training error is:

$$\begin{aligned}\mathbb{E}[Error] &= \frac{1}{n^2} \mathbb{E}[\|(X(X^\top X)^{-1} X^\top - I)\boldsymbol{\epsilon}\|_2^2] \\ &= \frac{1}{n^2} \mathbb{E}[\boldsymbol{\epsilon}^\top (X(X^\top X)^{-1} X^\top - I)^\top (X(X^\top X)^{-1} X^\top - I)\boldsymbol{\epsilon}] \\ &= \frac{1}{n^2} \mathbb{E}[\boldsymbol{\epsilon}^\top (I - X(X^\top X)^{-1} X^\top)^2 \boldsymbol{\epsilon}] \\ &= \frac{1}{n^2} \mathbb{E}[\boldsymbol{\epsilon}^\top (I - X(X^\top X)^{-1} X^\top) \boldsymbol{\epsilon}] \\ &= \frac{1}{n^2} \mathbb{E}[\boldsymbol{\epsilon}^\top \boldsymbol{\epsilon}] \mathbb{E}[(I - X(X^\top X)^{-1} X^\top)] \\ &= \frac{\sigma^2}{n^2} \text{tr}(I - X(X^\top X)^{-1} X^\top)\end{aligned}$$

Where tr denotes the trace of a matrix.

Now $\text{tr}(I - X(X^\top X)^{-1} X^\top) = n - d$,

therefore $\mathbb{E}[Error] = \frac{\sigma^2}{n}(1 - \frac{d}{n})$.

This is the expected final training error in terms of n , d , and σ .

1.3 Overparametrized Model

1.3.1

Answer from ChatGPT (Jan 9 ver.)

R Now consider $d > n$. We first illustrate that there exist multiple empirical risk minimizers.
For simplicity we let $n=1$ and $d=2$.
Choose $\mathbf{bx}_1 = [1; 1]$ 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 $\hat{\mathbf{w}}^\top \mathbf{bx}_1 = t_1$
on a real line.
Write down the equation of the line.

In this case, we have $n=1$ and $d=2$, so the data point and the learned parameter vector $\hat{\mathbf{w}}$ lie in a 2D plane. The training objective is to find a $\hat{\mathbf{w}}$ that satisfies $\hat{\mathbf{w}}^\top \mathbf{bx}_1 = t_1$.
We are given $\mathbf{bx}_1 = [1; 1]$ and $t_1 = 3$, so the equation $\hat{\mathbf{w}}^\top \mathbf{bx}_1 = t_1$ becomes:
$$\hat{\mathbf{w}}_1 + \hat{\mathbf{w}}_2 = 3$$
where $\hat{\mathbf{w}} = [\hat{\mathbf{w}}_1; \hat{\mathbf{w}}_2]$
This equation represents a line in the 2D plane, and as we have $d > n$ there are infinitely many solutions that lie on this line, with any point on the line corresponding to a valid solution for $\hat{\mathbf{w}}$.
The equation of the line is given by:
$$\hat{\mathbf{w}}_1 + \hat{\mathbf{w}}_2 = 3$$
or in slope-intercept form
$$\hat{\mathbf{w}}_2 = 3 - \hat{\mathbf{w}}_1$$

Answer is correct.

Answer:

In this case, we have $n = 1$ and $d = 2$, so the data point and the learned parameter vector $\hat{\mathbf{w}}$ lie in a 2D plane. The training objective is to find a $\hat{\mathbf{w}}$ that satisfies $\hat{\mathbf{w}}^\top \mathbf{x}_1 = t_1$.

We are given $\mathbf{x}_1 = [1; 1]$ and $t_1 = 3$, so the equation $\hat{\mathbf{w}}^\top \mathbf{x}_1 = t_1$ becomes:

$$\hat{\mathbf{w}}_1 + \hat{\mathbf{w}}_2 = 3$$

$$\text{where } \hat{\mathbf{w}} = [\hat{\mathbf{w}}_1; \hat{\mathbf{w}}_2]$$

This equation represents a line in the 2D plane, and as we have $d > n$ there are infinitely many solutions that lie on this line, with any point on the line corresponding to a valid solution for $\hat{\mathbf{w}}$.

The equation of the line is given by: $\hat{\mathbf{w}}_1 + \hat{\mathbf{w}}_2 = 3$

or in slope-intercept form $\hat{\mathbf{w}}_2 = 3 - \hat{\mathbf{w}}_1$.

1.3.2

From ChatGPT (Jan 9 ver.):

 Now, let's generalize the previous 2D case to the general $d > n$ case. Show that gradient descent from zero initialization (i.e. $\hat{b}(0) = 0$) finds a unique minimizer if it converges.

 In the general case where $d > n$, the data matrix X is tall and skinny and thus has a $\perp \!\!\! \perp$ null space. This means that there exists a non-zero vector v such that $Xv = 0$. Therefore, if we initialize the parameter vector \hat{b} to zero, it will lie in the null space of X , and the gradient of the loss function with respect to \hat{b} will be proportional to v .

Now, at each iteration of gradient descent, we update the parameter vector by subtracting the gradient of the loss function multiplied by a learning rate α :

$$\hat{b}(t+1) = \hat{b}(t) - \alpha \nabla L(\hat{b}(t))$$

As long as the learning rate α is positive and non-zero, the updated parameter vector $\hat{b}(t+1)$ will not be in the null space of X , since the gradient of the loss function is proportional to v , which is not zero.

Therefore, at each iteration, the parameter vector is projected out of the null space of X , and as the loss function is convex, the parameter vector will converge to a unique minimum, assuming the convergence of the gradient descent.

In summary, when $d > n$ and the gradient descent is initialized with zero, the parameter vector will be moved out of the null space of X at each iteration and converge to a unique minimizer, assuming that the gradient descent converges.

 Assume $d > n$, and since X is full rank hence $X^{\top}X$ is invertible. Show that the solution by gradient decent is $\hat{b} = X^{\top}(XX^{\top})^{-1}b$.

You can assume that the gradient is spanned by the rows of X and write $\hat{b} = X^{\top}(X^{\top}X)^{-1}b$.

Take gradient of the lost function with respect to \hat{b} first, then substitute $\hat{b} = X^{\top}(X^{\top}X)^{-1}b$.

 We can start with the gradient of the loss function with respect to \hat{b} :

$$\frac{\partial}{\partial b} \frac{1}{2} \|X\hat{b} - b\|^2 = \frac{1}{2} X^{\top}(X\hat{b} - b)$$

Now we substitute $\hat{b} = X^{\top}(X^{\top}X)^{-1}b$:

$$\frac{\partial}{\partial b} \frac{1}{2} \|X(X^{\top}(X^{\top}X)^{-1}b) - b\|^2 = \frac{1}{2} X^{\top}(X^{\top}X)^{-1}X^{\top}(X^{\top}X)^{-1}b - b$$

Since the gradient is spanned by the rows of X , the solution a that minimizes the loss function must satisfy the equation above.

$$X^{\top}Xa = b$$

Solving for a gives:

$$a = (X^{\top}X)^{-1}b$$

Therefore, the solution by gradient decent is:

$$\hat{b} = X^{\top}(X^{\top}X)^{-1}b = X^{\top}(X^{\top}X)^{-1}X^{\top}Xa = a$$

It's worth to mention that this solution is also known as the "normal equation" and it's a closed-form solution to the linear regression problem that can be computed efficiently using matrix inversion.

This actually gives pretty much correct answer, only needing some aesthetic fixes and some small reasoning fixes. In fact, in the second part where I ask it to show the solution of gradient descent, by specifying the steps you want it to

take really help it to get the correct answer (but not non-sense answer).

Answer:

In the general case where $d > n$, the data matrix X is tall and skinny and thus has a null space. This means that there exists a non-zero vector \mathbf{v} such that $X\mathbf{v} = 0$. Therefore, if we initialize the parameter vector $\hat{\mathbf{w}}$ to zero, it will lie in the null space of X , and the gradient of the loss function with respect to $\hat{\mathbf{w}}$ will be proportional to \mathbf{v} .

Now, at each iteration of gradient descent, we update the parameter vector by subtracting the gradient of the loss function multiplied by a learning rate α :

$$\hat{\mathbf{w}}^{(t+1)} = \hat{\mathbf{w}}^{(t)} - \alpha \nabla L(\hat{\mathbf{w}}^{(t)})$$

As long as the learning rate α is positive and non-zero, the updated parameter vector $\hat{\mathbf{w}}^{(t+1)}$ will not be in the null space of X , since the gradient of the loss function is proportional to \mathbf{v} , which is not zero.

Therefore, at first iteration, the parameter vector is projected out of the null space of X , and as the loss function is convex, the parameter vector will converge to a unique minimum, assuming the convergence of the gradient descent.

In summary, when $d > n$ and the gradient descent is initialized with zero, the parameter vector will be moved out of the null space of X at each iteration and converge to a unique minimizer, assuming that the gradient descent converges.

To show the solution of gradient descent, we can start with the gradient of the loss function with respect to $\hat{\mathbf{w}}$:

$$\nabla L(\hat{\mathbf{w}}) = \frac{2}{n} X^\top (X\hat{\mathbf{w}} - \mathbf{t})$$

Now we substitute $\hat{\mathbf{w}} = X^\top \mathbf{a}$:

$$\nabla L(\hat{\mathbf{w}}) = \frac{2}{n} X^\top (X X^\top \mathbf{a} - \mathbf{t})$$

Since the gradient is spanned by the rows of X , the solution \mathbf{a} that minimizes the loss function must satisfy the equation: $X X^\top \mathbf{a} = \mathbf{t}$.

Since by assumption $X X^\top$ is invertible, solving for \mathbf{a} gives: $\mathbf{a} = (X X^\top)^{-1} \mathbf{t}$

Therefore, the solution by gradient descent is:

$$\hat{\mathbf{w}} = X^\top \mathbf{a} = X^\top (X X^\top)^{-1} \mathbf{t}$$

1.3.3

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

We can see in the results that overparametrization doesn't always lead to overfitting, as when the degree is highly enough (for example $d=150$), the function it predicts actually match up every input data point while not oscillating

at a big scale as d=10 will do. However oscillating still is presenting, but in a smaller scale, where the curve it predicts seemed like it's not even smooth. The test error actually decreased as the degree is significantly overparametrizing (i.e. d=150 rather than d=10).

1.3.5

2 Back Propagation

2.1 Automatic Differentiation

2.1.1 Computational Graph

2.1.2 Backward Pass

$$\bar{\mathcal{J}} = 1$$

$$\bar{\mathcal{S}} = -\bar{\mathcal{J}}$$

$$\bar{\mathbf{y}}' = \bar{\mathcal{S}} \frac{\mathbb{I}(t = k)}{\mathbf{y}'} \quad (\text{where } \frac{1}{\mathbf{y}'} \text{ is element-wise reciprocal})$$

$$\bar{\mathbf{y}} = \text{softmax}'(\mathbf{y}) \bar{\mathbf{y}}'$$

$$\bar{\mathbf{g}} = \mathbf{W}^{(3)\top} \bar{\mathbf{y}}$$

$$\bar{\mathbf{h}}_1 = \text{diag}(\mathbf{h}_2) \bar{\mathbf{g}} \quad (\text{where } \text{diag}(\mathbf{h}_2) \text{ means a matrix with diagonal elements from } \mathbf{h}_2)$$

$$\bar{\mathbf{h}}_2 = \text{diag}(\mathbf{h}_1) \bar{\mathbf{g}}$$

$$\bar{\mathbf{z}}_1 = \begin{cases} 0 & \text{if } z_1 \leq 0 \\ \bar{\mathbf{h}}_1 & \text{if } z_1 > 0 \end{cases} \quad (\text{element-wise for the piecewise condition and the value of error signal})$$

$$\bar{\mathbf{z}}_2 = (\sigma(\mathbf{z}_2)(1 - \sigma(\mathbf{z}_2))^\top) \bar{\mathbf{h}}_2 \quad (\text{sigmoid and subtraction applied element-wise})$$

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

2.2 Gradient Norm Computation

2.2.1 Naive Computation

$$\bar{\mathbf{y}} = (1 \ 1 \ 1)^\top$$

$$\bar{\mathbf{W}}^{(2)} = \bar{\mathbf{y}} \mathbf{h}^\top$$

$$\bar{\mathbf{h}} = \mathbf{W}^{(2)\top} \bar{\mathbf{y}}$$

$$\bar{\mathbf{z}} = \begin{cases} 0 & \text{if } z_i \leq 0 \\ \bar{\mathbf{h}} & \text{if } z_i > 0 \end{cases} \quad (\text{Element-wise value i.e. negative entries from } \bar{\mathbf{h}} \text{ become zero})$$

$$\bar{\mathbf{W}}^{(1)} = \bar{\mathbf{z}} \mathbf{x}^\top$$

So

$$\begin{aligned}
\mathbf{z} &= \mathbf{W}^{(1)} \mathbf{x} \\
&= \begin{pmatrix} 1 & 2 & 1 \\ -2 & 1 & 0 \\ 1 & -2 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 3 \\ 1 \end{pmatrix} \\
&= \begin{pmatrix} 8 \\ 1 \\ -6 \end{pmatrix} \\
\mathbf{h} &= \text{ReLU}(\mathbf{z}) = \begin{pmatrix} 8 \\ 1 \\ 0 \end{pmatrix}
\end{aligned}$$

$$\begin{aligned}
\bar{\mathbf{h}} &= \mathbf{W}^{(2)\top} \bar{\mathbf{y}} \\
&= \begin{pmatrix} -2 & 1 & -3 \\ 4 & -2 & 4 \\ 1 & -3 & 6 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \\
&= \begin{pmatrix} -4 \\ 6 \\ 4 \end{pmatrix} \\
\bar{\mathbf{z}} &= \begin{pmatrix} -4 \\ 6 \\ 0 \end{pmatrix} \\
\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}} &= (\bar{\mathbf{z}} \mathbf{x}^\top)^\top \\
&= \left(\begin{pmatrix} -4 \\ 6 \\ 0 \end{pmatrix} (1 \ 3 \ 1) \right)^\top \\
&= \begin{pmatrix} -4 & -12 & -4 \\ 6 & 18 & 6 \\ 0 & 0 & 0 \end{pmatrix}^\top = \begin{pmatrix} -4 & 6 & 0 \\ -12 & 18 & 0 \\ -4 & 6 & 0 \end{pmatrix} \\
\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(2)}} &= (\bar{\mathbf{y}} \mathbf{h}^\top)^\top \\
&= \left(\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} (8 \ 1 \ 0) \right)^\top \\
&= \begin{pmatrix} 8 & 1 & 0 \\ 8 & 1 & 0 \\ 8 & 1 & 0 \end{pmatrix}^\top = \begin{pmatrix} 8 & 8 & 8 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}
\end{aligned}$$

And the Frobenius norm: $\|\overline{\mathbf{W}^{(1)}}\|_{\text{F}}^2 = 572$, $\|\overline{\mathbf{W}^{(2)}}\|_{\text{F}}^2 = 195$.

2.2.2 Efficient Computation

Using the trick, we have:

$$\begin{aligned}\|\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}}\|_F^2 &= \|\mathbf{x}\|_2^2 \|\bar{\mathbf{z}}\|_2^2 \\ &= (1^2 + 3^2 + 1^2)(-4^2 + 6^2 + 0^2) = 11 \cdot 52 = 572 \\ \|\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(2)}}\|_F^2 &= \|\mathbf{h}\|_2^2 \|\bar{\mathbf{y}}\|_2^2 \\ &= (8^2 + 1^2 + 0^2)(1^2 + 1^2 + 1^2) = 65 \cdot 3 = 195\end{aligned}$$

2.2.3 Complexity Analysis

From ChatGPT (Jan 9 ver.):

 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 forward and backward pass, 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 bias terms. You can also assume there are no in-place operations.

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.

 Let's start with the forward pass. We have K weight matrices, each of size $D \times D$, and N input vectors, each of size $D \times 1$. So, for each weight matrix, we need to perform $D \times D \times N$ scalar multiplications to compute the activations of all hidden units and output units.

Therefore, the total number of scalar multiplications for the forward pass is:

$$T_{\text{forward}} = K \times D^2 \times N$$

For the backward pass, we need to compute the error vectors for all the units, including the input units, hidden units, and output units. This requires us to perform $D^2 \times N$ scalar multiplications for each weight matrix.

Therefore, the total number of scalar multiplications for the backward pass is:

$$T_{\text{backward}} = K \times D^2 \times N$$

As for the memory cost, we need to store all the weight matrices, input vectors, activations, and error vectors. This is a total of $K \times D^2 + N \times D + K \times N \times D + N \times D$ scalars.

Therefore, the memory cost is:

$$M = O(K \times D^2 + N \times D + K \times N \times D)$$

R Do we need to store all activations in the forward pass?

G Yes, we need to store all activations in the forward pass because they are necessary for computing the gradients during the backward pass. The activations of each unit are used to compute the error vectors for that unit and its corresponding weight matrix, which is then used to update the weights. So, we need to keep track of all activations throughout the forward and backward pass.

R Do we need to store all error vectors in the backward pass?

G No, we do not need to store all error vectors in the backward pass. In the backward pass, we only need to store the error vectors for the output units, which are used to compute the gradients of the loss function with respect to the weights. The error vectors for the hidden units can be computed on-the-fly using the error vectors for the output units and the corresponding weight matrices.

This way, we only need to store the error vectors for the output units, and we can compute the error vectors for the hidden units as we go along, reducing the memory cost.

Therefore, the memory cost for the backward pass is:

$$M_{\text{backward}} = \mathcal{O}(N \times D)$$

It's correct for the computation time for the forward pass, but it ignore the fact that we also need to compute the gradient of each weight matrix for the backward pass (which is another D^2 scalar multiplication).

For the memory cost, it gets the difference between what we need to store in the forward pass and backward pass, but as we are performing gradient norm computation, some extra values need to be stored at that step.

Answer:

	T (Naive)	T (Efficient)	M (Naive)	M (Efficient)
Forward Pass	KD^2N	KD^2N	$\mathcal{O}(KD^2 + ND + KND)$	$\mathcal{O}(KD^2 + ND + KND)$
Backward Pass	$2KD^2N$	$2KD^2N$	$\mathcal{O}(ND)$	$\mathcal{O}(ND)$
Gradient Norm Computation	NKD^2	$2NKD$	$\mathcal{O}(NKD^2)$	$\mathcal{O}(NKD)$

Both algorithm perform the same on forward pass and backward pass. On gradient norm computation, the naive approach has to compute the square of each entry of the weight matrix, hence D^2 per weight matrix per example, while the efficient way only need to compute the norm of two vector with dimension D , hence $2D$ per weight per example. Naive need to store all Jacobian to perform norm computation per example, while efficient only need to store the error vector and activation per example.

2.3 Inner product of Jacobian: JVP and VJP

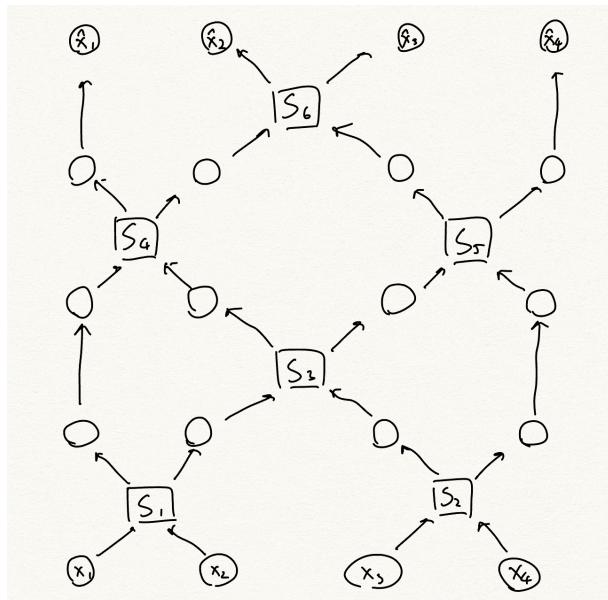
3 Hard-Coding Networks

3.1 Sort two numbers

$$\mathbf{W}^{(1)} = \begin{pmatrix} 0.5 & 0.5 \\ 0.5 & -0.5 \end{pmatrix} \quad \mathbf{b}^{(1)} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad \phi^{(1)}(z) = |z|$$

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

3.2 Perform Sort



3.3 Find the k^{th} smallest number

3.4 Counting Network

4 Linear Embedding - GloVe

4.1 GloVe Parameter Count

4.2 Expression for the vectorized loss function

From ChatGPT (Jan 30 ver.):

R Remember this loss function:
 $L(\{\{ \text{bw}_{-i}, \tilde{\text{bw}}_{-i}, b_{-i}, \tilde{b}_{-i} \}_{i=1}^V) = \sum_{i,j=1}^V (\text{bw}_{-i}^\top \tilde{\text{bw}}_{-i} + b_{-i} + \tilde{b}_{-i} - \log X_{ij})^2$
 Refer to this loss function when I mention "loss function" later on.

G Understood. The loss function mentioned refers to the sum of squared differences between the dot product of the embeddings, the biases and the log of the co-occurrence matrix elements.

R In practice, we concatenate the $\$\$$ embedding vectors into matrices $\$weightMat$, $\tilde{\text{weightMat}} \in \mathcal{M}(R)^{V \times d}$ and bias (column) vectors $\$mathbf{b}$, $\tilde{\text{b}} \in \mathcal{M}(R)^{V \times 1}$, where $\$S$ denotes the number of distinct words in the vocabulary. Rewrite the loss function L in a vectorized format in terms of $\$weightMat$, $\tilde{\text{weightMat}}$, \mathbf{b} , $\tilde{\text{b}}$, \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 $\text{textbf}\{1\} = [1 \dots 1]^\top \in \mathcal{M}(R)^{V \times 1}$. You can assume the bias vectors are column vectors, i.e. implicitly a matrix with V rows and 1 column: $\$mathbf{b}, \tilde{\text{b}} \in \mathcal{M}(R)^{V \times 1}$.
 Please use the align* environment in LateX to write the computation.

G We can rewrite the loss function as follows:

```
\begin{aligned}
& \begin{aligned}
& L(\text{weightMat}, \tilde{\text{weightMat}}, \mathbf{b}, \tilde{\text{b}}) = \sum_{i,j=1}^V \\
& (\text{bw}_{-i}^\top \tilde{\text{bw}}_{-i} + b_{-i} + \tilde{b}_{-i} - \log X_{ij})^2 \\
& & \approx \sum_{i,j=1}^V ((\text{weightMat}_{-i,-j})^\top \tilde{\text{weightMat}}_{-i,-j} + b_{-i} + \tilde{b}_{-i} - \\
& \log X_{ij})^2 \\
& & \approx \sum_{i,j=1}^V (\sum_{k=1}^d \text{weightMat}_{-i,k} \tilde{\text{weightMat}}_{-i,k} + b_{-i} + \\
& \tilde{b}_{-i} - \log X_{ij})^2 \\
& & \approx \sum_{i,j=1}^V (\text{weightMat}_{-i,-j})^\top \text{weightMat}_{-i,-j} + b_{-i} + \tilde{b}_{-i} - \\
& \log X_{ij})^2 \\
& & \approx \sum_{i,j=1}^V (\text{weightMat}^\top \tilde{\text{weightMat}})^\top \text{weightMat}_{-i,-j} + b_{-i} + \tilde{b}_{-i} - \\
& \log X_{ij})^2 \\
& & \approx \left[ \text{tr}((\text{weightMat}^\top \tilde{\text{weightMat}})^\top \text{weightMat}_{-i,-j}) + b_{-i} + \tilde{b}_{-i} - \\
& \log X_{ij} \right]^2
\end{aligned}
\end{aligned}
```

Here, $\text{tr}(\cdot)$ denotes the trace operator, and $\langle \cdot \rangle$ is the Frobenius norm.

The answer is correct, but there are still some minor aesthetic issues in its answer. After some cleaning it can be a good answer.

Answer:

$$\begin{aligned}
 L(\mathbf{W}, \tilde{\mathbf{W}}, \mathbf{b}, \tilde{\mathbf{b}}) &= \sum_{i,j=1}^V (\mathbf{w}_i^\top \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij})^2 \\
 &= \left\| \mathbf{W} \tilde{\mathbf{W}}^\top + \mathbf{b} \mathbf{1}^\top + \mathbf{1} \tilde{\mathbf{b}}^\top - \log \mathbf{X} \right\|_F^2 \\
 &= \text{tr}((\mathbf{W} \tilde{\mathbf{W}}^\top + \mathbf{b} \mathbf{1}^\top + \mathbf{1} \tilde{\mathbf{b}}^\top - \log \mathbf{X})^\top (\mathbf{W} \tilde{\mathbf{W}}^\top + \mathbf{b} \mathbf{1}^\top + \mathbf{1} \tilde{\mathbf{b}}^\top - \log \mathbf{X}))
 \end{aligned}$$

4.3 Expression for the vectorized gradient $\nabla_{\mathbf{W}} L$

From ChatGPT (Jan 30 ver.):

R Write the vectorized expression for $\nabla_{\text{weightMat}} L$, the gradient of the loss function L with respect to the embedding matrix W . The gradient should be a function of weightMat , $\text{Tilde}(\text{weightMat})$, \mathbf{b} , $\text{Tilde}(\mathbf{b})$, $\text{Tilde}(X)$.
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 weightMat can be calculated as follows:

```
\begin{aligned}
\nabla_{\text{weightMat}} L &\leftarrow \frac{\partial}{\partial \text{weightMat}} \text{tr}((\text{weightMat}^\top + \mathbf{b}^\top) (\text{weightMat}^\top + \mathbf{b}^\top)^\top - \log X)^2 \\
&\leftarrow 2 (\text{weightMat}^\top + \mathbf{b}^\top) (\text{weightMat}^\top + \mathbf{b}^\top)^\top - \log X \\
&\leftarrow 2 (\text{weightMat}^\top + \mathbf{b}^\top) (\text{weightMat}^\top + \mathbf{b}^\top)^\top - \log X
\end{aligned}
```

So, the gradient of the loss function with respect to the embedding matrix weightMat is given by $2 (\text{weightMat}^\top + \mathbf{b}^\top) (\text{weightMat}^\top + \mathbf{b}^\top)^\top - \log X$.

Even though it uses the square notation for the loss function (rather than the transposed multiplication version), it still manages to get the correct gradient.

Answer:

$$\begin{aligned} \nabla_{\mathbf{W}} L &= \frac{\partial}{\partial \mathbf{W}} \text{tr}((\mathbf{W}\tilde{\mathbf{W}}^\top + \mathbf{b}\mathbf{1}^\top + \mathbf{1}\tilde{\mathbf{b}}^\top - \log \mathbf{X})^\top (\mathbf{W}\tilde{\mathbf{W}}^\top + \mathbf{b}\mathbf{1}^\top + \mathbf{1}\tilde{\mathbf{b}}^\top - \log \mathbf{X})) \\ &= 2(\mathbf{W}\tilde{\mathbf{W}}^\top + \mathbf{b}\mathbf{1}^\top + \mathbf{1}\tilde{\mathbf{b}}^\top - \log \mathbf{X})\tilde{\mathbf{W}} \end{aligned}$$

4.4 Implement Vectorized Loss Function

R Use this loss function for later on instructions:
 $\$L(\text{weightMat}, \text{Tilde}(\text{weightMat}), \mathbf{b}, \text{Tilde}(\mathbf{b})) = \text{tr}((\text{weightMat}^\top + \mathbf{b}^\top) (\text{weightMat}^\top + \mathbf{b}^\top)^\top - \log \mathbf{X})^2$
 $\text{Tilde}(\text{weightMat})^\top = \text{weightMat}^\top - \log \mathbf{b}$
 $\text{Tilde}(\mathbf{b})^\top = \mathbf{b}^\top - \log \mathbf{X}$

Ok, I'll keep that loss function in mind for later instructions.

R Complete this python function. Only write in sections provided between "##### YOUR CODE HERE #####".
 $\text{def loss_GloVe(W, W_tilde, b, b_tilde, log_co_occurrence):}$
 *** 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 $V \times d$ where V is vocab size and d is the embedding dimension
 W_tilde : for asymmetric GloVe model, a second word embedding matrix, with dimensions $V \times d$
 b : bias vector, dimension $V \times 1$
 b_tilde : for asymmetric GloVe model, a second bias vector, dimension $V \times 1$
 $log_co_occurrence$: $V \times V$ log co-occurrence matrix (log X)
 Returns:
 $loss$: a scalar (float) for GloVe loss

 $n_{..} = \log_co_occurrence.shape$
 $# Symmetric Case, no W_tilde and b_tilde$
 $if W_tilde is None and b_tilde is None:$
 $# Symmetric model$
 $##### YOUR CODE HERE #####$
 $pass$
 $# loss = ...$
 $##### YOUR CODE HERE #####$
 $else:$
 $# Asymmetric model$
 $##### YOUR CODE HERE #####$
 $pass$
 $# loss = ...$
 $##### YOUR CODE HERE #####$
 $return loss$

Here's the completed function for both symmetric and asymmetric GloVe models:

```

python
import numpy as np

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

    Args:
        W: word embedding matrix, dimension V x d where V is vocab size
            is the embedding dimension
        W_tilde: for asymmetric GloVe model, a second word embedding matrix
            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
        loss = np.trace(
            (W @ W.T + b @ np.ones((1,n)) + np.ones((n,1)) @ b.T - 1)
            .T @ (W @ W.T + b @ np.ones((1,n)) + np.ones((n,1)) @ b)
        )
    else:
        # Asymmetric model
        loss = np.trace(
            (W @ W_tilde.T + b @ np.ones((1,n)) + np.ones((n,1)) @ b.T - 1)
            .T @ (W @ W_tilde.T + b @ np.ones((1,n)) + np.ones((n,1)) @ b)
        )
    return loss

```

4.5 Implement the gradient update of GloVe

R The gradient of this loss function with respect to weightMat is $2(\text{weightMat} \text{Tilde}(\text{weightMat})^{\top} + |\mathbf{b}| \text{textbf}(1)^{\top} + \text{textbf}(1)^{\top} \text{Tilde}(|\mathbf{b}|) - \log |\mathbf{b}|X)$. Compute the gradient of this loss function with respect to $\text{Tilde}(\text{weightMat})$, $|\mathbf{b}|$ and $\text{Tilde}(|\mathbf{b}|)$. Remember this 4 gradient for later on instructions.

The gradient of the loss function with respect to $\text{Tilde}(\text{weightMat})$ is $2(\text{weightMat} \text{Tilde}(\text{weightMat})^{\top} + |\mathbf{b}| \text{textbf}(1)^{\top} + \text{textbf}(1)^{\top} \text{Tilde}(|\mathbf{b}|) - \log |\mathbf{b}|X) \text{weightMat}$.

The gradient of the loss function with respect to $|\mathbf{b}|$ is $2(\text{weightMat} \text{Tilde}(\text{weightMat})^{\top} + |\mathbf{b}| \text{textbf}(1)^{\top} + \text{textbf}(1)^{\top} \text{Tilde}(|\mathbf{b}|) - \log |\mathbf{b}|X) \mathbf{b}$.

The gradient of the loss function with respect to $\text{Tilde}(|\mathbf{b}|)$ is $2(\text{weightMat} \text{Tilde}(\text{weightMat})^{\top} + |\mathbf{b}| \text{textbf}(1)^{\top} + \text{textbf}(1)^{\top} \text{Tilde}(|\mathbf{b}|) - \log |\mathbf{b}|X) \text{Tilde}(|\mathbf{b}|)$.

R Complete this python function using the four gradient computed previously. Only write in sections provided between ##### YOUR CODE HERE ##### and ##### YOUR CODE HERE #####.

```
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

    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
    ##### YOUR CODE HERE #####
    else:
        # Asymmetric case
        ##### YOUR CODE HERE #####
        pass
        # grad_W = ...
        # grad_W_tilde = ...
        # grad_b = ...
        # grad_b_tilde = ...
    ##### YOUR CODE HERE #####
    return grad_W, grad_W_tilde, grad_b, grad_b_tilde
```

Here's the implementation for the gradient calculation:

```
python
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)

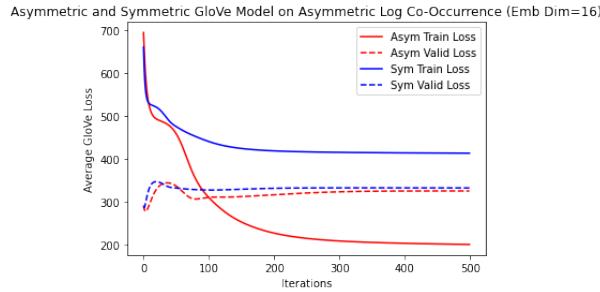
    Returns:
        grad_W: gradient of the loss wrt W, dimension V x d
        grad_W_tilde: gradient of the loss wrt W_tilde, dimension V x d. None if W_tilde is None.
        grad_b: gradient of the loss wrt b, dimension V x 1
        grad_b_tilde: gradient of the loss wrt b, dimension V x 1. None if b_tilde is None.
    """
    n,_ = log_co_occurrence.shape

    if W_tilde is None and b_tilde is None:
        # Symmetric case
        X = np.exp(np.dot(W, W.T) + b + b.T)
        grad_W = 2 * np.dot((X - log_co_occurrence), W)
        grad_b = 2 * np.sum(X - log_co_occurrence, axis=1, keepdims=True)
        grad_W_tilde = None
        grad_b_tilde = None
    else:
        # Asymmetric case
        X = np.exp(np.dot(W, W_tilde.T) + b + b_tilde.T)
        grad_W = 2 * np.dot((X - log_co_occurrence), W)
        grad_W_tilde = 2 * np.dot((X - log_co_occurrence).T, W)
        grad_b = 2 * np.sum(X - log_co_occurrence, axis=1, keepdims=True)
        grad_b_tilde = 2 * np.sum((X - log_co_occurrence).T, axis=1, keepdims=True)

    return grad_W, grad_W_tilde, grad_b, grad_b_tilde
```

Notice that it do put an exponential before doing computation, which I believe is not correct, but other parts are pretty correct to me.

Training and validation loss plot:



4.6 Effects of a buggy implementation

4.7 Effects of embedding dimension

5 Neural Language Model Network architecture

5.1 Number of parameters in neural network model

From ChatGPT (Jan 30 ver.):

R Consider this neural language network:
The network consists of an input layer, embedding layer, hidden layer and output layer.
The input consists of a sequence of N consecutive words.
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.

This neural language network is a word-level language model, where the input consists of a sequence of N words, and the output is a probability distribution over the vocabulary of V words. The input words are first mapped to their corresponding vector representations using an embedding layer, which has $N \times D$ units. These embeddings are then fed through a hidden layer with H units that applies a logistic nonlinearity. Finally, the outputs of the hidden layer are transformed into logits using an output layer with $H \times V$ units, and a softmax activation function is applied to generate a probability distribution over the vocabulary.

R The trainable parameters of the model consist of 3 weight matrices and 2 sets of biases.
The biases are applied to the hidden layer and the logits output layer.
What is the total number of trainable parameters in the model, as a function of V, N, D, H?

The number of trainable parameters in the model can be calculated as follows:

- Word embedding layer: $N \times D$ word embedding weights, total of $N \times D$ parameters.
- Hidden layer: $D \times H$ weights for the connections from the embedding layer, plus H biases, total of $(D \times H + H)$ parameters.
- Output layer: $H \times V$ weights for the connections from the hidden layer, plus $N \times V$ biases, total of $(H \times V + N \times V)$ parameters.

Therefore, the total number of trainable parameters in the model is $N \times D + (D \times H + H) + (H \times V + N \times V) = N \times D + D \times H + H \times V + N \times V$.

As a function of the parameters:

total parameters = $N \times D + D \times H + H \times V + N \times V$.

R Assume $V \gg H > D > N$. Which part of the model (i.e. which weight matrix or bias) has the largest number of trainable parameters? The symbol " \gg " means "much greater than".

Given the condition $V \gg H > D > N$, the weight matrix with the largest number of trainable parameters would be the one connecting the hidden layer to the logits output layer. This weight matrix has $H * V$ parameters, which would be much larger than the number of parameters in the other two weight matrices. The number of biases in the logits output layer ($N * V$) would also be much larger than the number of biases in the hidden layer (H).

ChatGPT didn't remember the fact that the weight used from input layer to embedding layer all use the same weight matrix, so that matrix should not be in shape $D \times N$, but only $D \times 1$. Also it seems to forget about there are N embedded words in the embedding layer.

Answer:

Number of parameters of each part is:

word_embedding_weights: D

embed_to_hid_weights: $N \times H \times D$

hid_to_output_weights: $N \times V \times H$

hid_bias: H

output_bias: $N \times V$

So in total there are: $D + NHD + NVH + H + NV = D + (ND+1)H + (H+1)NV$ trainable parameters.

It's still true that the weight from hidden layer to output layer is the largest parameter, because among the two large matrices (the embedding weights are small to consider here), weights to hidden layer is NHD while weights to output layer is NHV , and we know that $V \gg D$, so that weight is the largest parameter in this model.

5.2 Number of parameters in n-gram model

From ChatGPT (Jan 30 ver.):

R 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?

In an n-gram model with context length $N - 1$, each N-gram is comprised of N words, with $N - 1$ of these words serving as the context and the last word representing the target word. If the dictionary contains V words, then the total number of possible N-grams is V^N , as each word in the N-gram can be chosen from the V possible words in the dictionary.

Thus, if we store the counts of all possible N-grams explicitly, the table would have V^N entries.

Correct answer.

Answer:

If the dictionary contains V words, then the total number of possible N-grams is V^N , as each word in the N-gram can be chosen from the V possible words in the dictionary.

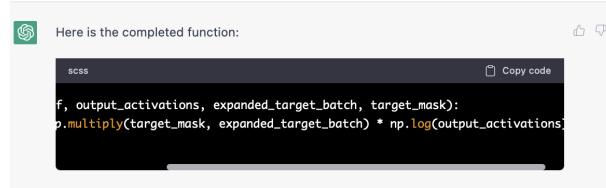
5.3 Comparing neural network and n-gram model scaling

6 Training the Neural Network

6.1 Implement Vectorized Loss

From ChatGPT (Jan 30 ver.):

```
R Complete this python method in the class "Model":  
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  
    """  
  
    The loss function is the cross-entropy loss:  
    $C = -\sum_{i=1}^B \sum_{n=1}^N \sum_{v=1}^V m^{(i)}_n \log y^{(i)}_{n*V + v}$  
    where B is the batch size, N is the numbers of training examples, and V is the size of  
    the vocabulary.
```



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

```
scss Here is the completed function:  
f, output_activations, expanded_target_batch, target_mask):  
    np.multiply(target_mask, expanded_target_batch) * np.log(output_activations)
```

It doesn't capture the details between the dimensions, however it does understand that the computation here should be element-wise multiplication.

6.2 Implement gradient with respect to parameters

From ChatGPT (Jan 30 ver.):

R Complete this python method in the class Model. Only write between "##### YOUR CODE HERE #####" and "#####".

```

#####
# YOUR CODE HERE
#####
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[:,0:vocab_size] contains the gradient
            dC / dz_0 for the i-th training example gradient for 1st output
            context word, and loss_derivative[:,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)

```

The code to complete the method is as follows:

```

makefile
Copy code
output_bias_grad = np.sum(loss_derivative, axis=0)
embed_to_hid_weights_grad = np.dot(activations.embedded_input.T, hid_de

```

Mostly correct answer, while the dimension for embed_to_hid_weights_grad should be transposed (or swap the operands and transpose inside the dot operation). Otherwise correct code.

6.3 Print the 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
```

6.4 Run model training

7 Bias in Word Embeddings

7.1 WEAT method for detecting bias

From ChatGPT (Jan 30 ver.):

R Word embedding models such as GloVe attempt to learn a vector space where semantically similar words are clustered close together. However, they have been shown to learn problematic associations, e.g. by embedding "man" more closely to "doctor" than "woman" (and vice versa for "nurse"). To detect such biases in word embeddings, "Semantics derived automatically from language corpora contain human-like biases" introduced the Word Embedding Association Test (WEAT). The WEAT test measures whether two target word sets (e.g., {programmer, engineer, scientist, ...} and {nurse, teacher, librarian, ...}) have the same relative association to two attribute word sets (e.g., man, male, ... and woman, female ...).

In short, the WEAT test is used to detect biases in word embeddings by comparing the relative associations between target and attribute word sets. The results of the test can highlight problematic associations learned by the embedding model, such as associating "man" more closely with "doctor" than "woman."

R Before proceeding, you should remember that glove has a similarity method, which computes the cosine similarity between two words.

G Yes, GloVe is a word embedding model that represents words as vectors in a high-dimensional space and the similarity between two words can be calculated using the cosine similarity between their vectors.

R We define our target words (occupations) and attribute words (A and B). Our target words consist of occupations, and our attribute words are gendered. This is the python code snippet:
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"]

G Yes, you can define target and attribute words for the WEAT test using the given code snippet. The target words consist of occupations and the attribute words are gendered words in two sets, A and B. To perform the test, you would calculate the similarity between each target word and each set of attribute words using the cosine similarity measure, and compare the similarities between the target words and the two sets of attribute words to determine if there are biases in the word embeddings.

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

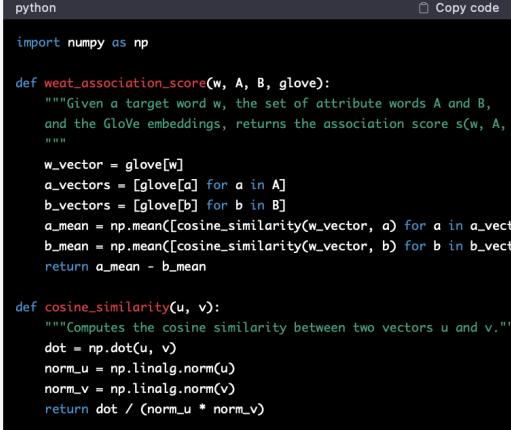
$$s(w, A, B) = \text{text}\{\text{mean}\}_{(a \in A)} \cos(w, a) - \text{text}\{\text{mean}\}_{(b \in B)} \cos(w, b)$$
.

Here is the docstring of the function:

```
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 is a possible implementation of the `weat_association_score` function:



```
python
import numpy as np

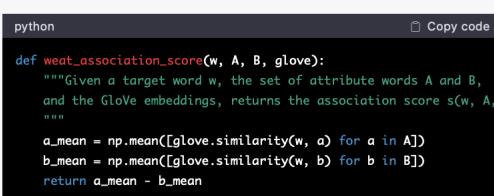
def weat_association_score(w, A, B, glove):
    """Given a target word w, the set of attribute words A and B,
    and the GloVe embeddings, returns the association score s(w, A, B).
    """
    w_vector = glove[w]
    a_vectors = [glove[a] for a in A]
    b_vectors = [glove[b] for b in B]
    a_mean = np.mean([cosine_similarity(w_vector, a) for a in a_vectors])
    b_mean = np.mean([cosine_similarity(w_vector, b) for b in b_vectors])
    return a_mean - b_mean

def cosine_similarity(u, v):
    """Computes the cosine similarity between two vectors u and v."""
    dot = np.dot(u, v)
    norm_u = np.linalg.norm(u)
    norm_v = np.linalg.norm(v)
    return dot / (norm_u * norm_v)
```

Note that the `glove` argument is assumed to be a dictionary-like object that maps words to their vector representations in the word embedding space. The function computes the association score by first computing the cosine similarity between the target word and each word in the attribute word sets, taking the mean of these similarities, and subtracting the mean similarity to the words in set B from the mean similarity to the words in set A.

R Assume that the cosine similarity between two words vectors can be computed by calling the python function glove.similarity(word_1, word_2). For example, glove.similarity("man", "scientist") output the cosine similarity between the word vector of "man" and word vector of "scientist" in glove. Use this information to re-write the python function weat_association_score.

Here's a revised implementation of the `weat_association_score` function using the `glove.similarity` method:



```
python
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).
    """
    a_mean = np.mean([glove.similarity(w, a) for a in A])
    b_mean = np.mean([glove.similarity(w, b) for b in B])
    return a_mean - b_mean
```

In this implementation, the `glove.similarity` method is used to compute the cosine similarity between the target word and each word in the attribute word sets, and the mean similarity is calculated in the same way as in the previous implementation.

R Now, write python code to compute the WEAT association score (use the `weat_association_score` function) for each element of occupations and the attribute sets A and B. Print out the result scores of each element of occupations. Recall that the following python snippets already store the required variables:

```
# 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"]
```

Here's the code to compute the WEAT association score for each element of the `occupations` list using the `weat_association_score` function:

```
python
for occupation in occupations:
    score = weat_association_score(occupation, A, B, glove)
    print(f"WEAT score for {occupation}: {score}")
```

This code will iterate through each occupation in the `occupations` list, compute the WEAT association score using the `weat_association_score` function, and print out the result score for each occupation.

The results:

```
WEAT score for programmer: 0.01961512863636017
WEAT score for engineer: 0.05364735424518585
WEAT score for scientist: 0.06795814633369446
WEAT score for nurse: -0.09486913681030273
WEAT score for teacher: -0.01893031597137451
WEAT score for librarian: -0.024141326546669006
```

7.2 Reasons for bias in word embeddings

7.3 Analyzing WEAT

7.3.1 1-word subsets

The word “male” and “female” gives the WEAT score of all negative values (all more closer to female than male) for all six occupations.

7.3.2 How word frequency affects embedding similarity

Using the new WEAT score formula only involving entries in the log co-occurrence matrix X , we can see that the score is actually computed by comparing how many times the target word (e.g. occupations) appears with the two different attributes sets with one word (e.g. genders), scaled by how many time the attribute words themselves appeared. In plain words, the score is decided by the corpus, where it favors more to the attribute word that appears more with the target word (when both attribute words appear about the same amount), or it favors more to the attribute word that appear less (when both attribute words appear with target word about the same amount).

7.3.3 Relative association between two sets of target words