

1.2.1

The goal is to minimize the following objective:

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

Taking the gradient with respect to $\hat{\mathbf{w}}$, we get:

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

Solving for $\hat{\mathbf{w}}$, we get:

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

Since XX^\top is invertible, we can multiply both sides by $(X^\top X)^{-1}$ to get:

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

Thus, the solution obtained by gradient descent is $\hat{w} = (X^\top X)^{-1} X^\top \mathbf{t}$, assuming training converges.

1J

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

\$\$

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

\$\$

We assume X is full rank: $X^\top X$ is invertible when $n > d$, and $X^\top X$ 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 $\hat{w} = (X^\top X)^{-1} X^\top \mathbf{t}$, assuming training converges. Show your work.



The goal is to minimize the following objective:

$$\frac{1}{n} \|X\hat{\mathbf{w}} - \mathbf{t}\|_2^2$$

Taking the gradient with respect to $\hat{\mathbf{w}}$, we get:

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

Solving for $\hat{\mathbf{w}}$, we get:

$$X^\top X\hat{\mathbf{w}} = X^\top \mathbf{t}$$

Since $X^\top X$ is invertible, we can multiply both sides by $(X^\top X)^{-1}$ to get:

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

Thus, the solution obtained by gradient descent is $\hat{\mathbf{w}} = (X^\top X)^{-1} X^\top \mathbf{t}$, assuming training converges.

1.2.2

First, let's substitute $\hat{\mathbf{w}} = (X^\top X)^{-1} X^\top \mathbf{t}$ into $\frac{1}{n} \|X\hat{\mathbf{w}} - \mathbf{t}\|_2^2$:

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

Expanding the right-hand side, we get:

$$\frac{1}{n} \|X(X^\top X)^{-1} X^\top \mathbf{t} - \mathbf{t}\|_2^2 = \frac{1}{n} \|X(X^\top X)^{-1} X^\top \mathbf{t} - X\mathbf{w}^* + X\mathbf{w}^* - \mathbf{t}\|_2^2 \quad (6)$$

Using the definition of $\mathbf{t} = X\mathbf{w}^* + \epsilon$, the above equation can be simplified as:

$$\frac{1}{n} \|X(X^\top X)^{-1} X^\top \mathbf{t} - \mathbf{t}\|_2^2 = \frac{1}{n} \|X(X^\top X)^{-1} X^\top (X\mathbf{w}^* + \epsilon) - X\mathbf{w}^* + X\mathbf{w}^* - (X\mathbf{w}^* + \epsilon)\|_2^2 \quad (7)$$

Expanding further, we get:

$$\frac{1}{n} \|X(X^\top X)^{-1} X^\top \mathbf{t} - \mathbf{t}\|_2^2 = \frac{1}{n} \|(I - X(X^\top X)^{-1} X^\top) \epsilon\|_2^2 \quad (8)$$

Thus, the training error can be derived as:

$$\text{Error} = \frac{1}{n} \|(I - X(X^\top X)^{-1} X^\top) \epsilon\|_2^2 \quad (9)$$

The expectation of the training error can be found by taking the expected value of the expression for the training error derived above. We know that the expectation of the noise term ϵ_i is zero, so we have:

$$E[\text{Error}] = E\left[\frac{1}{n}\left\|\left(X(X^\top X)^{-1}X^\top - I\right)\epsilon\right\|_2^2\right] = \frac{1}{n}E\left[\left\|\left(X(X^\top X)^{-1}X^\top - I\right)\epsilon\right\|_2^2\right] \quad (10)$$

Since ϵ is independent and identically distributed with zero mean and variance σ^2 , we have:

$$E\left[\left\|\left(X(X^\top X)^{-1}X^\top - I\right)\epsilon\right\|_2^2\right] = \sigma^2 E\left[\left\|\left(X(X^\top X)^{-1}X^\top - I\right)\right\|_2^2\right]. \quad (11)$$

Using the cyclic property of trace and the fact that $(A + B)^2 = A^2 + 2AB + B^2$, we have:

$$E\left[\left\|\left(X(X^\top X)^{-1}X^\top - I\right)\right\|_2^2\right] = \text{Trace}\left(\left(X(X^\top X)^{-1}X^\top - I\right)^2\right) \quad (12)$$

Therefore, the expectation of the training error is given by:

$$E[\text{Error}] = \frac{\sigma^2}{n} \text{Trace}\left(\left(X(X^\top X)^{-1}X^\top - I\right)^2\right) \quad (13)$$

The expectation of the training error is a function of n, d and σ , but it depends on the specific form of the data matrix X .



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

\$\$

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

\$\$

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

Hints: you might find the cyclic property $\text{Tr}(A^2) = \text{Tr}(A)$ useful.



First, let's substitute $\hat{\mathbf{w}} = \left(X^{\top} X\right)^{-1} X^{\top} \mathbf{t}$ into $\frac{1}{n}\left\|\left(X\left(X^{\top} X\right)^{-1} X^{\top}-I\right) \epsilon\right\|_2^2$:

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

Expanding the right-hand side, we get:

$$\frac{1}{n}\left\|X\left(X^{\top} X\right)^{-1} X^{\top} \mathbf{t}-\mathbf{t}\right\|_2^2=\frac{1}{n}\left\|X\left(X^{\top} X\right)^{-1} X^{\top} \mathbf{t}-X \mathbf{w}^{*}+X \mathbf{w}^{*}-\mathbf{t}\right\|_2^2$$

Using the definition of $\mathbf{t} = \mathbf{X} \mathbf{w}^* + \epsilon$, the above equation can be simplified as:

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

Expanding further, we get:

$$\frac{1}{n} \|\mathbf{X} (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{t} - \mathbf{t}\|_2^2 = \frac{1}{n} \|(\mathbf{I} - \mathbf{X} (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top) \epsilon\|_2^2$$

Thus, the training error can be derived as:

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

Now, let's find the expectation of the training error in terms of n , d and σ .

Since ϵ_i is generated independently with zero mean and variance σ^2 , the expectation of the training error is:

11

Now, let's find the expectation of the training error in terms of n , d and σ .



The expectation of the training error can be found by taking the expected value of the expression for the training error derived above. We know that the expectation of the noise term ϵ_i is zero, so we have:

$$E[\text{Error}] = E\left[\frac{1}{n} \left\| (X(X^T X)^{-1} X^T - I) \epsilon \right\|_2^2\right] = \frac{1}{n} E\left[\left\| (X(X^T X)^{-1} X^T - I) \epsilon \right\|_2^2\right].$$

Since ϵ is independent and identically distributed with zero mean and variance σ^2 , we have:

$$E\left[\left\| (X(X^T X)^{-1} X^T - I) \epsilon \right\|_2^2\right] = \sigma^2 E\left[\left\| (X(X^T X)^{-1} X^T - I) \right\|_2^2\right].$$

Using the cyclic property of trace and the fact that $(A + B)^2 = A^2 + 2AB + B^2$, we have:

$$E\left[\left\| (X(X^T X)^{-1} X^T - I) \right\|_2^2\right] = \text{Trace}\left((X(X^T X)^{-1} X^T - I)^2\right).$$

Therefore, the expectation of the training error is given by:

$$E[\text{Error}] = \frac{\sigma^2}{n} \text{Trace}\left((X(X^T X)^{-1} X^T - I)^2\right).$$

The expectation of the training error is a function of n , d and σ , but it depends on the specific form of the data matrix X .

1.3.1

For the given data point $(x_1, t_1) = ([1; 1], 3)$, the equation for the line that passes through this point and all possible \hat{w} is given by:

$$\hat{w}^\top x_1 = t_1 = 3 \tag{14}$$

Expanding the dot product, we have:

$$\hat{w}_1 + \hat{w}_2 = 3 \tag{15}$$

So the equation of the line is:

$$\hat{\mathbf{w}}_2 = 3 - \hat{\mathbf{w}}_1 \quad (16)$$

This line is a straight line in the $\hat{\mathbf{w}}_1$ - $\hat{\mathbf{w}}_2$ plane, and every point on this line represents a possible solution $\hat{\mathbf{w}}$ that satisfies the equation $\hat{\mathbf{w}}^\top \mathbf{x}_1 = t_1$. As there are infinitely many points on the line, there are infinitely many solutions to the equation, demonstrating that multiple empirical risk minimizers exist for the overparameterized case.

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



For the given data point $(\mathbf{x}_1, t_1) = ([1; 1], 3)$, the equation for the line that passes through this point and all possible $\hat{\mathbf{w}}$ is given by:

$$\hat{\mathbf{w}}^\top \mathbf{x}_1 = t_1 = 3$$

Expanding the dot product, we have:

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

So the equation of the line is:

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

This line is a straight line in the $\hat{\mathbf{w}}_1$ - $\hat{\mathbf{w}}_2$ plane, and every point on this line represents a possible solution $\hat{\mathbf{w}}$ that satisfies the equation $\hat{\mathbf{w}}^\top \mathbf{x}_1 = t_1$. As there are infinitely many points on the line, there are infinitely many solutions to the equation, demonstrating that multiple empirical risk minimizers exist for the overparameterized case.

1.3.2

We can rewrite $\hat{\mathbf{w}}$ as $\hat{\mathbf{w}} = X^\top \mathbf{a}$ for some $\mathbf{a} \in \mathbb{R}^n$. Using this expression, we can rewrite the MSE objective function as:

$$J(\mathbf{a}) = \frac{1}{2} \sum_{i=1}^n (\mathbf{a}^\top X \mathbf{x}_i - t_i)^2 \quad (17)$$

Taking the gradient with respect to \mathbf{a} and setting it to zero, we have:

$$\nabla_{\mathbf{a}} J(\mathbf{a}) = \sum_{i=1}^n (\mathbf{a}^\top X \mathbf{x}_i - t_i) X \mathbf{x}_i = 0 \quad (18)$$

Expanding, we get:

$$\sum_{i=1}^n \mathbf{a}^\top X \mathbf{x}_i X \mathbf{x}_i * i^\top - \sum_{i=1}^n t_i X \mathbf{x}_i = 0 \quad (19)$$

Rearranging, we get:

$$\mathbf{a}^\top X X^\top X = X^\top \mathbf{t} \quad (20)$$

Since $d > n$, $X X^\top$ is an invertible $d \times d$ matrix. Multiplying both sides by $(X X^\top)^{-1}$, we get:

$$\mathbf{a} = (X X^\top)^{-1} \mathbf{t} \quad (21)$$

And

$$\hat{\mathbf{w}} = X^\top \mathbf{a} = X^\top (X X^\top)^{-1} \mathbf{t} \quad (22)$$

This is the closed-form solution for linear regression.

Gradient descent is an iterative optimization algorithm that updates the parameters based on the gradient of the objective function. Starting from an initial point, gradient descent iteratively moves in the direction of the negative gradient until convergence.

In this case, the objective function is the MSE, and its gradient is $\nabla_{\mathbf{a}} J(\mathbf{a})$. The gradient descent update rule is given by:

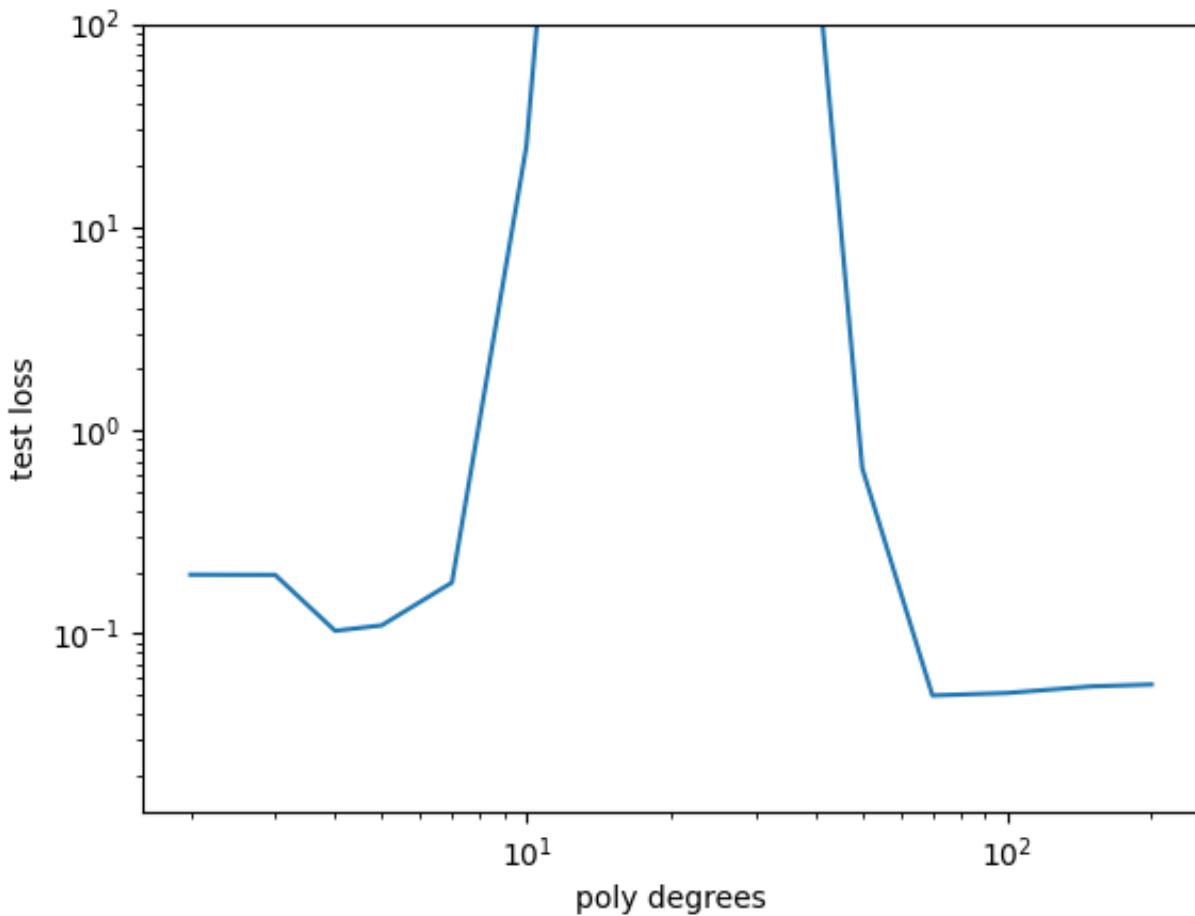
$$\mathbf{a}_{k+1} = \mathbf{a}_k - \eta \nabla_{\mathbf{a}} J(\mathbf{a}_k) \quad (23)$$

where η is the learning rate and k is the iteration number.

If we start from $\mathbf{a}(0) = 0$, gradient descent will converge to the unique solution $\mathbf{a} = (X X^\top)^{-1} \mathbf{t}$. And the final solution $\hat{\mathbf{w}} = X^\top \mathbf{a} = X^\top (X X^\top)^{-1} \mathbf{t}$.

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



Based on the test loss figure above, higher poly degrees doesn't always lead to overfitting, using a large poly degree eventually produces a lower test loss than using small poly degrees.

2.1.2

$$\mathbf{y} = \mathbf{W}^{(3)}\mathbf{g} + \mathbf{W}^{(4)}\mathbf{x} \implies \mathbf{y}' = \text{softmax}(\mathbf{y}) \implies \bar{\mathbf{y}} = \text{softmax}(\mathbf{y}) \odot (\mathbb{I}(t=k) - \mathbf{y}')$$

$$\mathbf{g} = \mathbf{h}_1 \circ \mathbf{h}_2 \implies \bar{\mathbf{g}} = \bar{\mathbf{h}}_1 \odot \mathbf{h}_2 + \mathbf{h}_1 \odot \bar{\mathbf{h}}_2$$

$$\mathbf{h}_2 = \sigma(\mathbf{z}_2) \implies \bar{\mathbf{h}}_2 = \sigma'(\mathbf{z}_2)\bar{\mathbf{z}}_2$$

$$\mathbf{z}_2 = \mathbf{W}^{(2)}\mathbf{x} + \mathbf{b}^{(2)} \implies \bar{\mathbf{z}}_2 = \mathbf{W}^{(2)\top}\bar{\mathbf{h}}_2$$

$$\mathbf{h}_1 = \text{ReLU}(\mathbf{z}_1) \implies \bar{\mathbf{h}}_1 = \mathbb{I}(\mathbf{z}_1 > 0)\bar{\mathbf{z}}_1$$

$$\mathbf{z}_1 = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \implies \bar{\mathbf{z}}_1 = \mathbf{W}^{(1)\top}\bar{\mathbf{h}}_1$$

Finally, $\bar{\mathbf{x}} = \mathbf{W}^{(2)\top}\bar{\mathbf{h}}_2 + \mathbf{W}^{(1)\top}\bar{\mathbf{h}}_1$

2.2.1

First, let's compute the Jacobian matrices for the first and second layers:

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

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} \rightarrow \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}} = \mathbf{x}^\top$$

$$\mathbf{h} = \text{ReLU}(\mathbf{z}) \rightarrow \frac{\partial \mathbf{h}}{\partial \mathbf{z}} = [z > 0] = [h > 0]$$

$$\mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} \rightarrow \frac{\partial \mathbf{y}}{\partial \mathbf{h}} = \mathbf{W}^{(2)\top}$$

$$\frac{\partial \mathcal{J}}{\partial \mathbf{y}} = \bar{\mathbf{y}}$$

$$\frac{\partial \mathcal{J}}{\partial \mathbf{h}} = \bar{\mathbf{y}}^\top \frac{\partial \mathbf{y}}{\partial \mathbf{h}} = \bar{\mathbf{y}}^\top \mathbf{W}^{(2)\top}$$

$$\frac{\partial \mathcal{J}}{\partial \mathbf{z}} = \frac{\partial \mathcal{J}}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{z}} = \bar{\mathbf{y}}^\top \mathbf{W}^{(2)\top} \cdot [h > 0]$$

$$\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}} = \bar{\mathbf{y}}^\top \mathbf{W}^{(2)\top} \cdot [h > 0] \cdot \mathbf{x}^\top$$

$$\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(2)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{W}^{(2)}} = \bar{\mathbf{y}}^\top \cdot \mathbf{h}^\top$$

Then, we have

$$\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}} = \begin{pmatrix} 3 & 9 & 3 \\ -4 & -12 & -4 \\ 0 & 0 & 0 \end{pmatrix} \quad (24)$$

$$\left\| \frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}} \right\|_{\text{F}}^2 = 275 \quad (25)$$

And

$$\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(2)}} = \begin{pmatrix} 8 & 1 & 0 \\ 8 & 1 & 0 \\ 8 & 1 & 0 \end{pmatrix} \quad (26)$$

$$\left\| \frac{\partial \mathcal{J}}{\partial \mathbf{W}(2)} \right\|_{\text{F}}^2 = 195 \quad (27)$$

2.2.2

```

import numpy as np

W1 = np.array([[1, 2, 1], [-2, 1, 0], [1, -2, -1]])
W2 = np.array([[2, 4, 1], [1, -2, -3], [-3, 4, 6]])

x = np.array([1, 3, 1])
overline_y = np.array([1, 1, 1])

z = np.dot(W1, x)
h = np.maximum(z, 0)
y = np.dot(W2, h)

frobenius_norm_partial_J_partial_W1 = np.sum(x ** 2) * np.sum((overline_y.dot(W2.T) *
(h > 0)) ** 2)
frobenius_norm_partial_J_partial_W2 = np.sum(h ** 2) * np.sum(overline_y ** 2)

print(frobenius_norm_partial_J_partial_W1)
print(frobenius_norm_partial_J_partial_W2)

```

Based on the efficient computation trick, we have

$$\left\| \frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}} \right\|_{\text{F}}^2 = \|\mathbf{x}\|_2^2 \|\bar{\mathbf{z}}\|_2^2 = 275 \quad (28)$$

$$\left\| \frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(2)}} \right\|_{\text{F}}^2 = \|\mathbf{h}\|_2^2 \|\bar{\mathbf{y}}\|_2^2 = 195 \quad (29)$$

which is the same as the naive computation.

2.2.3

Table:

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

Explanations:

- Forward Pass:
 - For each input vector, we need to compute the activations in all layers, which requires ND^2 scalar multiplications in total.
 - The memory cost of storing the model parameters and activations is KD^2 , where K is the number of weight matrices.
 - Backward Pass:
 - In the backward pass, we need to compute the error vectors in all layers. It requires KND^2 scalar multiplications in total.
 - In the naive computation, we need to store all the error vectors, which has a memory cost of KD^2 . However, in the efficient computation, we can reuse the memory used by activations in the forward pass, which reduces the memory cost to D^2 .
 - Gradient Norm Computation:
 - In the naive computation, we need to square the gradient before aggregation, which requires KD^2N scalar multiplications.
 - In the efficient computation, we can reuse the memory used by the error vectors in the backward pass, which reduces the memory cost to KD . The number of scalar multiplications required is KD .

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

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

```

\begin{tabular}{|c||l||l||l||}
\hline & $\mathrm{T}$ (Naive) & $\mathrm{T}$ (Efficient) & $\mathrm{M}$ (Naive) & $\mathrm{M}$ (Efficient) \\
\hline Forward Pass & & & \\
\hline Backward Pass & & & \\
\hline Gradient Norm Computation & & & \\
\hline
\end{tabular}

```

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



Table:



	& Naive	& Efficient	& Naive	& Efficient
---	---	---	---	---
Forward Pass	ND^2	ND^2	KD^2	KD^2
Backward Pass	KND^2	KND^2	KD^2	D^2
Gradient Norm Computation	KD^{2N}	KD	KD^{2N}	KD

Explanations:

- Forward Pass:
 - For each input vector, we need to compute the activations in all layers, which requires ND^2 scalar multiplications in total.
 - The memory cost of storing the model parameters and activations is KD^2 , where K is the number of weight matrices.
- Backward Pass:
 - In the backward pass, we need to compute the error vectors in all layers. It requires KND^2 scalar multiplications in total.
 - In the naive computation, we need to store all the error vectors, which has a memory cost of KD^2 . However, in the efficient computation, we can reuse the memory used by activations in the forward pass, which reduces the memory cost to D^2 .
- Gradient Norm Computation:
 - In the naive computation, we need to square the gradient before aggregation, which requires KD^{2N} scalar multiplications.
 - In the efficient computation, we can reuse the memory used by the error vectors in the backward pass, which Regenerate response cost to KD . The number of

3.1

Weights:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad (30)$$

$$\mathbf{W}^{(2)} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \end{bmatrix} \quad (31)$$

Bias:

$$\mathbf{b}^{(1)} = [0 \quad 0] \quad (32)$$

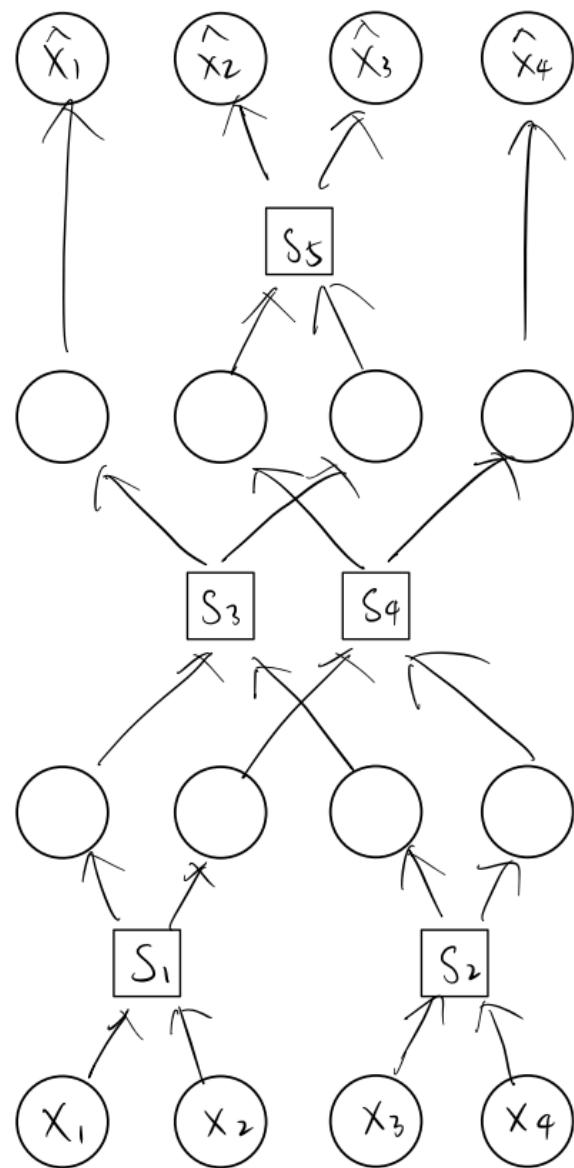
$$\mathbf{b}^{(2)} = [0 \quad 0] \quad (33)$$

Activation functions:

$$\phi^{(1)}(z) = \text{ReLU}(z) \quad (34)$$

$$\phi^{(2)}(z) = \text{ReLU}(z) \quad (35)$$

3.2



11

draw a graph to illustrate how merge sort can be used to sort 4 numbers (the sort_2 will be used for 5 times)



I'm sorry, I cannot draw a graph here as this is a text-based interface. However, I can explain how merge sort can sort 4 numbers using the sort_2 function.



Merge sort uses a divide-and-conquer approach to sort numbers. To sort 4 numbers, the 4 numbers can be divided into two pairs of two numbers each. The sort_2 function is then used to sort each pair of numbers, resulting in two sorted pairs. Finally, the two sorted pairs are merged to get a sorted array of 4 numbers.

The steps for sorting 4 numbers using sort_2 and merge sort can be illustrated as follows:

1. Divide the 4 numbers into two pairs: (x1, x2) and (x3, x4).
2. Use the sort_2 function to sort each pair of numbers: (y1, y2) and (y3, y4).
3. Merge the two sorted pairs to get a sorted array of 4 numbers: (y1, y2, y3, y4).

12

how to merge the pairs?



To merge the two sorted pairs, you can compare the first element of each pair and take the smaller one as the first element of the final sorted array. Then, you can compare the second element of each pair and take the smaller one as the second element of the final sorted array. Repeat this process until both pairs are completely merged into one sorted array.



Here's an example of merging two sorted pairs of numbers:

Given two sorted pairs of numbers: (y1, y2) and (y3, y4)

1. Compare y1 and y3, take the smaller one as the first element of the final sorted array, let's say y1 is smaller, so the first element of the final sorted array is y1.
2. Compare y2 and y3, take the smaller one as the second element of the final sorted array, let's say y3 is smaller, so the second element of the final sorted array is y3.
3. The final sorted array is (y1, y3, y2, y4).

a1_code

February 5, 2023

1 Starter code and data

First, perform the required imports for your code:

```
[1]: import collections
import pickle
import numpy as np
import os
from tqdm import tqdm
from six.moves.urllib.request import urlretrieve
import tarfile
import sys
import pylab
import itertools

TINY = 1e-30
EPS = 1e-4
nax = np.newaxis
```

If you're using colaboratory, this following script creates a folder - here we used 'CSC413/A1' - in order to download and store the data. If you're not using colaboratory, then set the path to wherever you want the contents to be stored at locally.

You can also manually download and unzip the data from [http://www.cs.toronto.edu/~jba/a1_data.tar.gz] and put them in the same folder as where you store this notebook.

Feel free to use a different way to access the files *data.pk* , *partially_trained.pk*, and *raw_sentences.txt*.

The file *raw_sentences.txt* contains the sentences that we will be using for this assignment. These sentences are fairly simple ones and cover a vocabulary of only 250 words (+ 1 special [MASK] token word).

```
[2]: drive_location = 'data'
PARTIALLY_TRAINED_MODEL = './data/partially_trained.pk'
data_location = './data/data.pk'
```

We have already extracted the 4-grams from this dataset and divided them into training, validation, and test sets. To inspect this data, run the following:

```
[3]: data = pickle.load(open(data_location, 'rb'))
print(data['vocab'][0]) # First word in vocab is [MASK]
print(data['vocab'][1])
print(len(data['vocab'])) # Number of words in vocab
print(data['vocab']) # All the words in vocab
print(data['train_inputs'][:10]) # 10 example training instances
```

[MASK]
all
251
['[MASK]', 'all', 'set', 'just', 'show', 'being', 'money', 'over', 'both',
'years', 'four', 'through', 'during', 'go', 'still', 'children', 'before',
'police', 'office', 'million', 'also', 'less', 'had', ',', 'including',
'should', 'to', 'only', 'going', 'under', 'has', 'might', 'do', 'them', 'good',
'around', 'get', 'very', 'big', 'dr.', 'game', 'every', 'know', 'they', 'not',
'world', 'now', 'him', 'school', 'several', 'like', 'did', 'university',
'companies', 'these', 'she', 'team', 'found', 'where', 'right', 'says',
'people', 'house', 'national', 'some', 'back', 'see', 'street', 'are', 'year',
'home', 'best', 'out', 'even', 'what', 'said', 'for', 'federal', 'since', 'its',
'may', 'state', 'does', 'john', 'between', 'new', ';', 'three', 'public', '?',
'be', 'we', 'after', 'business', 'never', 'use', 'here', 'york', 'members',
'percent', 'put', 'group', 'come', 'by', '\$', 'on', 'about', 'last', 'her',
'of', 'could', 'days', 'against', 'times', 'women', 'place', 'think', 'first',
'among', 'own', 'family', 'into', 'each', 'one', 'down', 'because', 'long',
'another', 'such', 'old', 'next', 'your', 'market', 'second', 'city', 'little',
'from', 'would', 'few', 'west', 'there', 'political', 'two', 'been', '.',
'their', 'much', 'music', 'too', 'way', 'white', ':', 'was', 'war', 'today',
'more', 'ago', 'life', 'that', 'season', 'company', '-', 'but', 'part', 'court',
'former', 'general', 'with', 'than', 'those', 'he', 'me', 'high', 'made',
'this', 'work', 'up', 'us', 'until', 'will', 'ms.', 'while', 'officials', 'can',
'were', 'country', 'my', 'called', 'and', 'program', 'have', 'then', 'is', 'it',
'an', 'states', 'case', 'say', 'his', 'at', 'want', 'in', 'any', 'as', 'if',
'united', 'end', 'no', ')', 'make', 'government', 'when', 'american', 'same',
'how', 'mr.', 'other', 'take', 'which', 'department', '--', 'you', 'many', 'nt',
'day', 'week', 'play', 'used', "'s", 'though', 'our', 'who', 'yesterday',
'director', 'most', 'president', 'law', 'man', 'a', 'night', 'off', 'center',
'i', 'well', 'or', 'without', 'so', 'time', 'five', 'the', 'left']
[[28 26 90 144]
[184 44 249 117]
[183 32 76 122]
[117 247 201 186]
[223 190 249 6]
[42 74 26 32]
[242 32 223 32]
[223 32 158 144]
[74 32 221 32]
[42 192 91 68]]

Now `data` is a Python dict which contains the vocabulary, as well as the inputs and targets for all three splits of the data. `data['vocab']` is a list of the 251 words in the dictionary; `data['vocab'][0]` is the word with index 0, and so on. `data['train_inputs']` is a 372,500 x 4 matrix where each row gives the indices of the 4 consecutive context words for one of the 372,500 training cases. The validation and test sets are handled analogously.

Even though you only have to modify two specific locations in the code, you may want to read through this code before starting the assignment.

2 Part 4: Linear Embedding – GloVe (3pts)

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 $X \in \mathbb{N}^{V \times V}$ 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 $\mathbf{w}_i, \tilde{\mathbf{w}}_i \in \mathbb{R}^d$, as well as two scalar biases $b_i, \tilde{b}_i \in \mathbb{R}$. Typically we have the dimension of the embedding d much smaller than the number of words V . This objective can be written as:

$$L(\{\mathbf{w}_i, \tilde{\mathbf{w}}_i, b_i, \tilde{b}_i\}_{i=1}^V) = \sum_{i,j=1}^V (\mathbf{w}_i^\top \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij})^2$$

Note that each word is represented by two d -dimensional embedding vectors $\mathbf{w}_i, \tilde{\mathbf{w}}_i$ and two scalar biases b_i, \tilde{b}_i . When the bias terms are omitted and we tie the two embedding vectors $\mathbf{w}_i = \tilde{\mathbf{w}}_i$, then GloVe corresponds to finding a rank- d symmetric factorization of the co-occurrence matrix.

2.1 4.2 Expression for the Vectorized Loss function [0.5pt] [Type 1]

In practice, we concatenate the V embedding vectors into matrices $\mathbf{W}, \tilde{\mathbf{W}} \in \mathbb{R}^{V \times d}$ and bias (column) vectors $\mathbf{b}, \tilde{\mathbf{b}} \in \mathbb{R}^V$, 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 $\mathbf{W}, \tilde{\mathbf{W}}, \mathbf{b}, \tilde{\mathbf{b}}, X$. You are allowed to use elementwise operations such as addition and subtraction as well as matrix operations such as the Frobenius norm and/or trace operator in your answer.

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

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

Answer:

The vectorized form of the loss function is given by:

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

where $\mathbf{1} \in \mathbb{R}^V$ is an all-ones column vector, and $\|\cdot\|_F$ is the Frobenius norm.

ChatGPT trace:

1
$$\$ \$L(\{\mathbf{w}\}_{-i}, \tilde{\{\mathbf{w}\}}_{-i}, \mathbf{b}_{-i}, \tilde{\{\mathbf{b}\}}_{-i})_{-i=1}^V = \sum_{i,j=1}^V (\mathbf{w}_{-i}^T \tilde{\{\mathbf{w}\}}_{-i} + \mathbf{b}_{-i} + \tilde{\{\mathbf{b}\}}_{-i} - \log X_{ij})^2 \$\$ ## 4.2$$

Expression for the Vectorized Loss function [0.5pt] [Type 1]

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

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

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



The vectorized form of the loss function is given by:

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

where $\mathbf{1}$ in \mathbb{R}^V is an all-ones column vector, and $\|\cdot\|_F$ is the Frobenius norm.

2.2 4.3. Expression for gradient $\frac{\partial L}{\partial \mathbf{W}}$ [0.5pt] [Type 2]

Write the vectorized expression for $\frac{\partial L}{\partial \mathbf{W}}$, the gradient of the loss function L with respect to the embedding matrix \mathbf{W} . The gradient should be a function of \mathbf{W} , $\tilde{\mathbf{W}}$, \mathbf{b} , $\tilde{\mathbf{b}}$, 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.

Answer:

The gradient of the loss function L with respect to the embedding matrix \mathbf{W} is given by:

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

ChatGPT trace:



Write the vectorized expression for $\frac{\partial L}{\partial \mathbf{W}}$, the gradient of the loss function L with respect to the embedding matrix \mathbf{W} . The gradient should be a function of \mathbf{W} , $\tilde{\mathbf{W}}$, \mathbf{b} , $\tilde{\mathbf{b}}$, 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 \mathbf{W} is given by:

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

2.3 4.4 Implement Vectorized Loss Function [1pt] [Type 2]

Implement the `loss_GloVe()` function of GloVe.

See YOUR CODE HERE Comment below for where to complete the code

Note that you need to implement both the loss for an *asymmetric* model (from your answer in question 1.2) and the loss for a *symmetric* model which uses the same embedding matrix \mathbf{W} and bias vector \mathbf{b} for the first and second word in the co-occurrence, i.e. $\tilde{\mathbf{W}} = \mathbf{W}$ and $\tilde{\mathbf{b}} = \mathbf{b}$ in the original loss.

Hint: You may take advantage of NumPy's broadcasting feature for the bias vectors: <https://numpy.org/doc/stable/user/basics.broadcasting.html>

We have provided a few functions for training the embedding:

- `calculate_log_co_occurrence` computes the log co-occurrence matrix of a given corpus
- `train_GloVe` runs momentum gradient descent to optimize the embedding
- **loss_GloVe: TO BE IMPLEMENTED.**
- INPUT
 - $V \times d$ matrix \mathbf{W} (collection of V embedding vectors, each d -dimensional)
 - $V \times d$ matrix $\mathbf{W}_{\text{tilde}}$
 - $V \times 1$ vector \mathbf{b} (collection of V bias terms)
 - $V \times 1$ vector $\mathbf{b}_{\text{tilde}}$
 - $V \times V$ log co-occurrence matrix.
- OUTPUT

- loss of the GloVe objective
- **grad_GloVe: TO BE IMPLEMENTED.**
- INPUT:
 - $V \times d$ matrix W (collection of V embedding vectors, each d -dimensional), embedding for first word;
 - $V \times d$ matrix W_{tilde} , embedding for second word;
 - $V \times 1$ vector b (collection of V bias terms);
 - $V \times 1$ vector b_{tilde} , bias for second word;
 - $V \times V$ log co-occurrence matrix.
- OUTPUT:
 - $V \times d$ matrix grad_W containing the gradient of the loss function w.r.t. W ;
 - $V \times d$ matrix $\text{grad}_W_{\text{tilde}}$ containing the gradient of the loss function w.r.t. W_{tilde} ;
 - $V \times 1$ vector grad_b which is the gradient of the loss function w.r.t. b .
 - $V \times 1$ vector $\text{grad}_b_{\text{tilde}}$ which is the gradient of the loss function w.r.t. b_{tilde} .

Run the code to compute the co-occurrence matrix.

```
[4]: vocab_size = len(data['vocab']) # Number of vocabs

def calculate_log_co_occurrence(word_data, symmetric=False):
    "Compute the log-co-occurrence matrix for our data."
    log_co_occurrence = np.zeros((vocab_size, vocab_size))
    for input in word_data:
        # Note: the co-occurrence matrix may not be symmetric
        log_co_occurrence[input[0], input[1]] += 1
        log_co_occurrence[input[1], input[2]] += 1
        log_co_occurrence[input[2], input[3]] += 1
        # Diagonal entries are just the frequency of the word
        log_co_occurrence[input[0], input[0]] += 1
        log_co_occurrence[input[1], input[1]] += 1
        log_co_occurrence[input[2], input[2]] += 1
        log_co_occurrence[input[3], input[3]] += 1
        # If we want symmetric co-occurrence can also increment for these.
        if symmetric:
            log_co_occurrence[input[1], input[0]] += 1
            log_co_occurrence[input[2], input[1]] += 1
            log_co_occurrence[input[3], input[2]] += 1
    delta_smoothing = 0.5 # A hyperparameter. You can play with this if you want.
    log_co_occurrence += delta_smoothing # Add delta so log doesn't break on 0's.
    log_co_occurrence = np.log(log_co_occurrence)
    return log_co_occurrence

asym_log_co_occurrence_train = calculate_log_co_occurrence(data['train_inputs'], symmetric=False)
asym_log_co_occurrence_valid = calculate_log_co_occurrence(data['valid_inputs'], symmetric=False)
```

- **TO BE IMPLEMENTED:** Implement the loss function. You should vectorize the computation, i.e. not loop over every word.

Hint: To prompt a GPT-like model, you may include in your prompt the vectorized loss formula derived in 4.2.

```
[67]: 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

    $$
    L(\mathbf{W}, \tilde{\mathbf{W}}, \mathbf{b}, \tilde{\mathbf{b}}) = \left( \mathbf{W}^\top \mathbf{W} + \mathbf{b}^\top \mathbf{b} - \log \mathbf{X} \right)^2
    $$
    """
    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 #####
        loss = np.sum((np.matmul(W, W.T) + b + b.T - log_co_occurrence)**2)
    else:
        # Asymmetric model
        ##### YOUR CODE HERE #####
        loss = np.sum((np.matmul(W, W_tilde.T) + b + b_tilde.T - log_co_occurrence)**2)
```

```

    return loss

# testing the loss function
W = np.random.randn(vocab_size, 16)
W_tilde = np.random.randn(vocab_size, 16)
b = np.random.randn(vocab_size, 1)
b_tilde = np.random.randn(vocab_size, 1)

print(loss_GloVe(W, W_tilde, b, b_tilde, asym_log_co_occurrence_train))
print(loss_GloVe(W, None, b, None, asym_log_co_occurrence_train))

```

1328584.8540725093

1262403.2244261734

ChatGPT trace:

```

u 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 #####
        loss =
    #####
    else:
        # Asymmetric model
        ##### YOUR CODE HERE #####
        pass
    loss =
    #####
    return loss

```



Here is the implementation of the loss function for both symmetric and asymmetric GloVe models:



```
python                                Copy code

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
        loss = np.sum((np.matmul(W,W.T) + b + b.T - log_co_occurrence)**2)
    else:
        # Asymmetric model
        loss = np.sum((np.matmul(W,W_tilde.T) + b + b_tilde.T - log_co_occurrence)**2)
    return loss
```

2.4 4.5. Implement the gradient update of GloVe. [1pt] [Type 2]

Implement the `grad_GloVe()` function which computes the gradient of GloVe.

See YOUR CODE HERE Comment below for where to complete the code

Again, note that you need to implement the gradient for both the symmetric and asymmetric models.

- TO BE IMPLEMENTED:** Calculate the gradient of the loss function w.r.t. the parameters W , \tilde{W} , \mathbf{b} , and \mathbf{b}_t . You should vectorize the computation, i.e. not loop over every word.

Hint: To prompt a GPT-like model, you may include the GloVe loss implementation in 4.4.

```
[75]: 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 $V \times d$ 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:

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

```
$$
\begin{aligned}
& \frac{\partial L}{\partial W} = 2 \left( \mathbf{W}^T \mathbf{W} + \mathbf{b} \mathbf{b}^T + \mathbf{W}^T \mathbf{b} - \log X \right) \mathbf{W}^T \\
& \quad \frac{\partial L}{\partial \tilde{W}} = 2 \left( \mathbf{W}^T \mathbf{W} + \mathbf{b} \mathbf{b}^T + \mathbf{W}^T \mathbf{b} - \log X \right) \mathbf{W}^T \\
& \quad \frac{\partial L}{\partial b} = 2 \left( \mathbf{W}^T \mathbf{W} + \mathbf{b} \mathbf{b}^T + \mathbf{W}^T \mathbf{b} - \log X \right) \mathbf{b}^T \\
& \quad \frac{\partial L}{\partial \tilde{b}} = 2 \left( \mathbf{W}^T \mathbf{W} + \mathbf{b} \mathbf{b}^T + \mathbf{W}^T \mathbf{b} - \log X \right) \mathbf{b}^T
\end{aligned}
$$
"""
n,_ = log_co_occurrence.shape

# print(W.shape)
# print(W_tilde.shape)
# print(b.shape)
# print(b_tilde.shape)
# print(log_co_occurrence.shape)

# (251, 16)
# (251, 16)
# (251, 1)
# (251, 1)
# (251, 251)
```

```

if W_tilde is None and b_tilde is None:
    # Symmetric case
    error = np.matmul(W,W.T) + b + b.T - log_co_occurrence
    grad_W = 2 * np.matmul(error, W)
    grad_W_tilde = None
    grad_b = 2 * np.matmul(error, np.ones((n,1)))
    grad_b_tilde = None
else:
    # Asymmetric case
    error = np.matmul(W,W_tilde.T) + b + b_tilde.T - log_co_occurrence
    grad_W = 2 * np.matmul(error, W_tilde)
    grad_W_tilde = 2 * np.matmul(error.T, W)
    grad_b = 2 * np.matmul(error, np.ones((n,1)))
    grad_b_tilde = 2 * np.matmul(error, np.ones((n,1)))

return grad_W, grad_W_tilde, grad_b, grad_b_tilde

# np.random.seed(0)

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

# grad_GloVe(W, W_tilde, b, b_tilde, asym_log_co_occurrence_train)

```

ChatGPT trace

```

1 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.
$$
\begin{aligned}
&\frac{\partial L}{\partial \mathbf{W}} = 2 \left( (\mathbf{W}^T \tilde{\mathbf{W}} + \mathbf{b} \mathbf{b}^T) - \log \mathbf{X} \right) \mathbf{W}^T \\
&\frac{\partial L}{\partial \tilde{\mathbf{W}}} = 2 \left( (\mathbf{W}^T \tilde{\mathbf{W}} + \mathbf{b} \mathbf{b}^T) - \log \mathbf{X} \right) \tilde{\mathbf{W}}^T \\
&\frac{\partial L}{\partial \mathbf{b}} = 2 \left( (\mathbf{W}^T \tilde{\mathbf{W}} + \mathbf{b} \mathbf{b}^T) - \log \mathbf{X} \right) \mathbf{b}^T \\
&\frac{\partial L}{\partial \tilde{\mathbf{b}}} = 2 \left( (\mathbf{W}^T \tilde{\mathbf{W}} + \mathbf{b} \mathbf{b}^T) - \log \mathbf{X} \right) \tilde{\mathbf{b}}^T
\end{aligned}
$$
"""
n,_ = log_co_occurrence.shape

if W_tilde is None and b_tilde is None:
    # Symmetric case
    ##### YOUR CODE HERE #####
    grad_W = ...
    grad_b = ...
    grad_W_tilde = None
    grad_b_tilde = None

#####
else:
    # Asymmetric case
    ##### YOUR CODE HERE #####
    grad_W = ...
    grad_W_tilde = ...
    grad_b = ...
    grad_b_tilde = ...

#####
return grad_W, grad_W_tilde, grad_b, grad_b_tilde

```



Here is the implementation for the gradients of GloVe objective function w.r.t its parameters:

```
python Copy code

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
        error = W.T @ W_tilde + b @ np.ones((n, 1)).T + np.ones((n, 1)) @ b_tilde
        grad_W = 2 * error @ W_tilde.T
        grad_b = 2 * error @ np.ones((n, 1))
        grad_W_tilde = None
        grad_b_tilde = None
    else:
        # Asymmetric case
        error = W.T @ W_tilde + b @ np.ones((n, 1)).T + np.ones((n, 1)) @ b_tilde
        grad_W = 2 * error @ W_tilde.T
        grad_W_tilde = 2 * error @ W.T
        grad_b = 2 * error @ np.ones((n, 1))
        grad_b_tilde = 2 * error @ np.ones((n, 1))

    return grad_W, grad_W_tilde, grad_b, grad_b_tilde
```

To help you debug your GloVe gradient computation, we have included a finite-difference gradient checker function defined below:

```
[68]: def relative_error(a, b):
        return np.abs(a - b) / (np.abs(a) + np.abs(b))

def check_GloVe_gradients(W, W_tilde, b, b_tilde, log_co_occurrence):
    """Check the computed gradients using finite differences."""
    np.random.seed(0)
    np.seterr(all='ignore') # suppress a warning which is harmless

    # Obtain the analytical gradient
    grad_W, grad_W_tilde, grad_b, grad_b_tilde = grad_GloVe(W, W_tilde, b, b_tilde, log_co_occurrence)
    grads_dict = {"W":grad_W, "W_tilde": grad_W_tilde,
                  "b": grad_b, "b_tilde": grad_b_tilde}

    params_dict = {"W":W, "W_tilde":W_tilde, "b":b, "b_tilde":b_tilde}
```

```

# Check that the shapes of the parameters and gradients match
for name in params_dict:
    if params_dict[name] is None:
        continue
    dims = params_dict[name].shape
    is_matrix = (len(dims) == 2)
    if not is_matrix:
        print()

    if params_dict[name].shape != grads_dict[name].shape:
        print('The gradient for {} should be size {} but is actually {}.'.format(
            name, params_dict[name].shape, grads_dict[name].shape))
return

# Run finite difference for that param
for count in range(1000):
    if is_matrix:
        slc = np.random.randint(0, dims[0]), np.random.randint(0, dims[1])
    else:
        slc = np.random.randint(dims[0])

    params_dict_plus = params_dict.copy()
    params_dict_plus[name] = params_dict[name].copy()
    params_dict_plus[name][slc] += EPS
    obj_plus = loss_GloVe(params_dict_plus["W"],
                          params_dict_plus["W_tilde"],
                          params_dict_plus["b"],
                          params_dict_plus["b_tilde"],
                          log_co_occurrence)

    params_dict_minus = params_dict.copy()
    params_dict_minus[name] = params_dict[name].copy()
    params_dict_minus[name][slc] -= EPS
    obj_minus = loss_GloVe(params_dict_minus["W"],
                          params_dict_minus["W_tilde"],
                          params_dict_minus["b"],
                          params_dict_minus["b_tilde"],
                          log_co_occurrence)

    empirical = (obj_plus - obj_minus) / (2. * EPS)
    exact = grads_dict[name][slc]
    rel = relative_error(empirical, exact)
    if rel > 5e-4:
        print('The loss derivative has a relative error of {}, which is too large for param {}.'.format(rel, name))
print(empirical)

```

```

    print(exact)
    return False
print('The gradient for {} looks OK.'.format(name))

```

Run the cell below to check if your `grad_GloVe` function passes the checker. The function will check for both the symmetric and asymmetric loss, for each of the parameter variables whether its gradient computation looks ok. The expected output is:

```

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

```

```

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

```

Note: If you update the `grad_GloVe` cell while debugging, make sure to run the `grad_GloVe` cell again before re-running the cell below to check the gradient.

TODO: Run this cell below to check the gradient implementation

```
[76]: np.random.seed(0)

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

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

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

```

Checking asymmetric loss gradient...
The gradient for W looks OK.
The gradient for W_tilde looks OK.
The gradient for b looks OK.
The loss derivative has a relative error of 1.0, which is too large for param
b_tilde.
-18.510645750211552
87.45658387528688

```

```
Checking symmetric loss gradient...
```

```
The loss derivative has a relative error of 1.0, which is too large for param W.  
-0.4959737998433411  
0.47961455453046875
```

[76]: False

Now that you have checked that the gradient is correct, we define the training function for the model given the initial weights and ground truth log co-occurrence matrix:

```
[35]: def train_GloVe(W, W_tilde, b, b_tilde, log_co_occurrence_train, log_co_occurrence_valid, n_epochs, do_print=False):  
    """Training W and b according to GloVe objective."  
    n,_ = log_co_occurrence_train.shape  
    learning_rate = 0.05 / n # A hyperparameter. You can play with this if you want.  
  
    train_loss_list = np.zeros(n_epochs)  
    valid_loss_list = np.zeros(n_epochs)  
    vocab_size = log_co_occurrence_train.shape[0]  
  
    for epoch in range(n_epochs):  
        grad_W, grad_W_tilde, grad_b, grad_b_tilde = grad_GloVe(W, W_tilde, b, b_tilde, log_co_occurrence_train)  
        W = W - learning_rate * grad_W  
        b = b - learning_rate * grad_b  
        if not grad_W_tilde is None and not grad_b_tilde is None:  
            W_tilde = W_tilde - learning_rate * grad_W_tilde  
            b_tilde = b_tilde - learning_rate * grad_b_tilde  
            train_loss, valid_loss = loss_GloVe(W, W_tilde, b, b_tilde, log_co_occurrence_train), loss_GloVe(W, W_tilde, b, b_tilde, log_co_occurrence_valid)  
            if do_print:  
                print(f"Average Train Loss: {train_loss / vocab_size}, Average valid loss: {valid_loss / vocab_size}, grad_norm: {np.sum(grad_W**2)}")  
            train_loss_list[epoch] = train_loss / vocab_size  
            valid_loss_list[epoch] = valid_loss / vocab_size  
  
    return W, W_tilde, b, b_tilde, train_loss_list, valid_loss_list
```

□ **TODO:** Run this cell below to run an experiment training GloVe model

```
[70]: ### TODO: Run this cell ###  
np.random.seed(1)  
n_epochs = 500 # A hyperparameter. You can play with this if you want.  
  
# Store the final losses for graphing  
do_print = False # If you want to see diagnostic information during training  
init_variance = 0.1 # A hyperparameter. You can play with this if you want.  
embedding_dim = 16
```

```

W = init_variance * np.random.normal(size=(vocab_size, embedding_dim))
W_tilde = init_variance * np.random.normal(size=(vocab_size, embedding_dim))
b = init_variance * np.random.normal(size=(vocab_size, 1))
b_tilde = init_variance * np.random.normal(size=(vocab_size, 1))

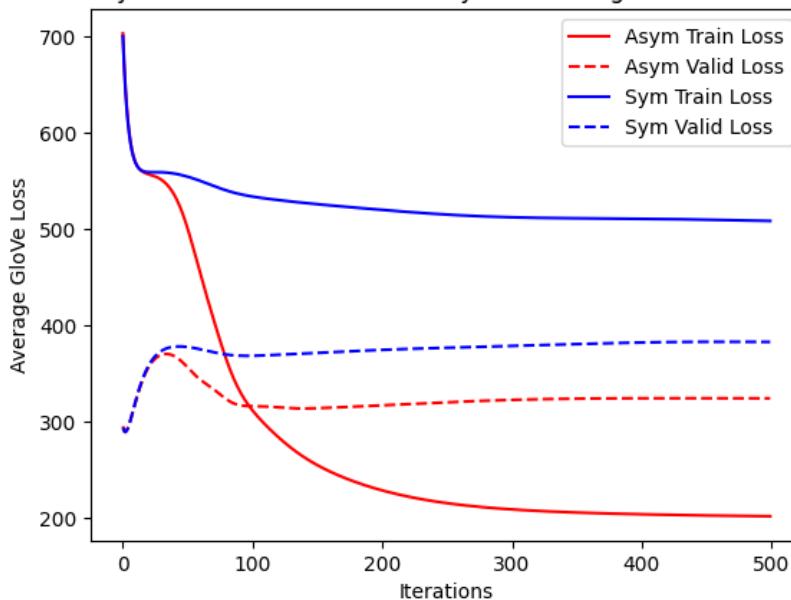
# Run the training for the asymmetric and symmetric GloVe model
Asym_W_final, Asym_W_tilde_final, Asym_b_final, Asym_b_tilde_final, □
↳ Asym_train_loss_list, Asym_valid_loss_list = train_GloVe(W, W_tilde, b, □
↳ b_tilde, asym_log_co_occurrence_train, asym_log_co_occurrence_valid, n_epochs, □
↳ do_print=do_print)

Sym_W_final, Sym_W_tilde_final, Sym_b_final, Sym_b_tilde_final, □
↳ Sym_train_loss_list, Sym_valid_loss_list = train_GloVe(W, None, b, None, □
↳ asym_log_co_occurrence_train, asym_log_co_occurrence_valid, n_epochs, □
↳ do_print=do_print)

# Plot the resulting training curve
pylab.plot(Asym_train_loss_list, label="Asym Train Loss", color='red')
pylab.plot(Asym_valid_loss_list, label="Asym Valid Loss", color='red', □
↳ linestyle='--')
pylab.plot(Sym_train_loss_list, label="Sym Train Loss", color='blue')
pylab.plot(Sym_valid_loss_list, label="Sym Valid Loss", color='blue', □
↳ linestyle='--')
pylab.xlabel("Iterations")
pylab.ylabel("Average GloVe Loss")
pylab.title("Asymmetric and Symmetric GloVe Model on Asymmetric Log Co-Occurrence (Emb Dim={})".format(embedding_dim))
pylab.legend()
pylab.show()

```

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



3 Part 5: Neural Language Model Network architecture (2pt)

See the handout for the written questions in this part.

3.1 Answer the following questions

3.2 5.1. Number of parameters in neural network model [1pt] [Type 2]

The trainable parameters of the model consist of 3 weight matrices and 2 sets of biases. What is the total number of trainable parameters in the model, as a function of V, N, D, H ?

In the diagram given, 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 $V \gg H > D > N$? Note: The symbol \gg means “much greater than” Explain your reasoning.

Answer

The total number of trainable parameters in the model is given by the sum of the number of parameters in each of the 3 weight matrices and 2 sets of biases:

Word Embedding Weights matrix: $V \times D$ parameters
Embedding to Hidden Weights matrix: $H \times (N \times D)$ parameters
Hidden to Output Weights matrix: $V \times H \times N$ parameters
Hidden biases: H parameters
Output biases: V parameters

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

If we have the constraint that $V \gg H > D > N$, then the part of the model with the largest number of parameters is the Hidden to Output Weights matrix, which has $V \times H \times N$ parameters.

ChatGPT Trace

In this assignment, we will train a neural language model like the one we covered in lecture and as in Bengio et al. [2003]. However, we will modify the architecture slightly, inspired by the Masked Language Modeling (MLM) objective introduced in BERT [Devlin et al., 2018]. The network takes in N consecutive words, where one of the words is replaced with a [MASK] token.¹² The aim of the network is to predict the masked word in the corresponding output location. See Figure 1 for the diagram of this architecture.

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).¹³

5.1 Number of parameters in neural network model [1pt] [Type 2]

The trainable parameters of the model consist of 3 weight matrices and 2 sets of biases. What is the total number of trainable parameters in the model, as a function of V , N , D , H ? In the diagram given above, which part of the model (i.e., `word_embedding_weights`, `embed_to_hid_weights`, `hid_to_output_weights`, `hid_bias`, or `output_bias`) has the largest number of trainable parameters if we have the constraint that $V \gg H > D > N$?

14 Explain your reasoning.



The total number of trainable parameters in the model is given by the sum of the number of parameters in each of the 3 weight matrices and 2 sets of biases:



1. Word Embedding Weights matrix: $V \times D$ parameters
2. Embedding to Hidden Weights matrix: $H \times (N \times D)$ parameters
3. Hidden to Output Weights matrix: $V \times H$ parameters
4. Hidden biases: H parameters
5. Output biases: V parameters

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

If we have the constraint that $V \gg H > D > N$, then the part of the model with the largest number of parameters is the Embedding to Hidden Weights matrix, which has $H \times (N \times D)$ parameters. This is because, in this case, the product of the number of hidden units and the number of dimensions in the embedding layer is larger than the number of parameters in any of t

Regenerate response del.

3.3 5.2 Number of parameters in n -gram model [1pt] [Type 1]

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

Answer

In an n -gram model with context length $N - 1$, we need to store the counts of all possible N -grams. The number of possible N -grams is given by V^N . This is because for each of the N words in the N -gram, there are V possible words that could be chosen from the dictionary.

Therefore, the table for storing the counts of all possible N -grams would have V^N entries.

ChatGPT Trace

u

5.2 Number of parameters in \$n\$-gram model {[1pt]} {[Type 1]}

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



In an n -gram model with context length $N-1$, we need to store the counts of all possible N -grams. The number of possible N -grams is given by V^N . This is because for each of the N words in the N -gram, there are V possible words that could be chosen from the dictionary.

Therefore, the table for storing the counts of all possible N -grams would have V^N entries.

4 Part 6: Training the Neural Network (2pts)

In this part, you will learn to implement and train the neural language model from Figure 1. As described in the previous section, during training, 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. In practice, this [MASK] token is assigned the index 0 in our dictionary. The weights $W^{(2)} = \text{hid_to_output_weights}$ now has the shape $NV \times H$, 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 $(v + nV)$ -th column is for the word v in 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:

$$C = - \sum_i^B \sum_n^N \sum_v^V m_n^{(i)} (t_{v+nV}^{(i)} \log y_{v+nV}^{(i)})$$

Where: $* y_{v+nV}^{(i)}$ denotes the output probability prediction from the neural network for the i -th training example for the word v in the n -th output word. Denoting z as the logits output, we define the output probability y as a softmax on z over contiguous chunks of V units (see also Figure 1):

$$y_{v+nV}^{(i)} = \frac{e^{z_{v+nV}^{(i)}}}{\sum_l^V e^{z_{l+nV}^{(i)}}}$$

$* t_{v+nV}^{(i)} \in \{0, 1\}$ is 1 if for the i -th training example, the word v is the n -th word in context
 $m_n^{(i)} \in \{0, 1\}$ is a mask that is set to 1 if we are predicting the n -th word position for the i -th example (because we had masked that word in the input), and 0 otherwise

There are three classes defined in this part: `Params`, `Activations`, `Model`. You will make changes to `Model`, but it may help to read through `Params` and `Activations` first.

```
[77]: class Params(object):
    """A class representing the trainable parameters of the model. This class
    has five fields:

        word_embedding_weights, a matrix of size V x D, where V is the
        number of words in the vocabulary
        and D is the embedding dimension.

        embed_to_hid_weights, a matrix of size H x ND, where H is the number
        of hidden units. The first D
        columns represent connections from the embedding of the
        first context word, the next D columns
        for the second context word, and so on. There are N context
        words.

        hid_bias, a vector of length H
        hid_to_output_weights, a matrix of size NV x H
        output_bias, a vector of length NV"""

    def __init__(self, word_embedding_weights, embed_to_hid_weights,
                 hid_to_output_weights,
                 hid_bias, output_bias):
        self.word_embedding_weights = word_embedding_weights
        self.embed_to_hid_weights = embed_to_hid_weights
        self.hid_to_output_weights = hid_to_output_weights
        self.hid_bias = hid_bias
```

```

        self.output_bias = output_bias

    def copy(self):
        return self.__class__(self.word_embedding_weights.copy(), self.
embed_to_hid_weights.copy(),
                             self.hid_to_output_weights.copy(), self.hid_bias.
copy(), self.output_bias.copy())

    @classmethod
    def zeros(cls, vocab_size, context_len, embedding_dim, num_hid):
        """A constructor which initializes all weights and biases to 0."""
        word_embedding_weights = np.zeros((vocab_size, embedding_dim))
        embed_to_hid_weights = np.zeros((num_hid, context_len * embedding_dim))
        hid_to_output_weights = np.zeros((vocab_size * context_len, num_hid))
        hid_bias = np.zeros(num_hid)
        output_bias = np.zeros(vocab_size * context_len)
        return cls(word_embedding_weights, embed_to_hid_weights,
hid_to_output_weights,
               hid_bias, output_bias)

    @classmethod
    def random_init(cls, init_wt, vocab_size, context_len, embedding_dim,
num_hid):
        """A constructor which initializes weights to small random values and
biases to 0."""
        word_embedding_weights = np.random.normal(0., init_wt, size=(vocab_size, embedding_dim))
        embed_to_hid_weights = np.random.normal(0., init_wt, size=(num_hid, context_len * embedding_dim))
        hid_to_output_weights = np.random.normal(0., init_wt, size=(vocab_size *
context_len, num_hid))
        hid_bias = np.zeros(num_hid)
        output_bias = np.zeros(vocab_size * context_len)
        return cls(word_embedding_weights, embed_to_hid_weights,
hid_to_output_weights,
               hid_bias, output_bias)

##### The functions below are Python's somewhat oddball way of overloading
operators, so that
##### we can do arithmetic on Params instances. You don't need to
understand this to do the assignment.

    def __mul__(self, a):
        return self.__class__(a * self.word_embedding_weights,
                            a * self.embed_to_hid_weights,
                            a * self.hid_to_output_weights,

```

```

        a * self.hid_bias,
        a * self.output_bias)

def __rmul__(self, a):
    return self * a

def __add__(self, other):
    return self.__class__(self.word_embedding_weights + other.
word_embedding_weights,
                         self.embed_to_hid_weights + other.
embed_to_hid_weights,
                         self.hid_to_output_weights + other.
hid_to_output_weights,
                         self.hid_bias + other.hid_bias,
                         self.output_bias + other.output_bias)

def __sub__(self, other):
    return self + -1. * other

```

[78]:

```

class Activations(object):
    """A class representing the activations of the units in the network. This
    class has three fields:

        embedding_layer, a matrix of  $B \times ND$  matrix (where  $B$  is the batch size,
    ↵ $D$  is the embedding dimension,
        and  $N$  is the number of input context words), representing the
    ↵activations for the embedding
        layer on all the cases in a batch. The first  $D$  columns
    ↵represent the embeddings for the
        first context word, and so on.
        hidden_layer, a  $B \times H$  matrix representing the hidden layer activations
    ↵for a batch
        output_layer, a  $B \times V$  matrix representing the output layer activations
    ↵for a batch"""

    def __init__(self, embedding_layer, hidden_layer, output_layer):
        self.embedding_layer = embedding_layer
        self.hidden_layer = hidden_layer
        self.output_layer = output_layer

    def get_batches(inputs, batch_size, shuffle=True):
        """Divide a dataset (usually the training set) into mini-batches of a given
    ↵size. This is a
        'generator', i.e. something you can use in a for loop. You don't need to
    ↵understand how it
        works to do the assignment."""

```

```

if inputs.shape[0] % batch_size != 0:
    raise RuntimeError('The number of data points must be a multiple of the
batch size.')
num_batches = inputs.shape[0] // batch_size

if shuffle:
    idxs = np.random.permutation(inputs.shape[0])
    inputs = inputs[idxs, :]

for m in range(num_batches):
    yield inputs[m * batch_size:(m + 1) * batch_size, :]

```

In this part of the assignment, you implement a method which computes the gradient using back-propagation. To start you out, the *Model* class contains several important methods used in training:

- `compute_activations` computes the activations of all units on a given input batch
- `compute_loss_derivative` computes the gradient with respect to the output logits $\frac{\partial C}{\partial z}$
- `evaluate` computes the average cross-entropy loss for a given set of inputs and targets

You will need to complete the implementation of two additional methods to complete the training, and print the outputs of the gradients.

4.1 6.1 Implement gradient with respect to output layer inputs [0.5pt] [Type 2]

Implement a vectorized `compute_loss` function, which computes the total cross-entropy loss on a mini-batch according to Eq. 2. Look for the `## YOUR CODE HERE ##` comment for where to complete the code. The docstring provides a description of the inputs to the function.

Answer

```
loss = -((expanded_target_batch * np.log(output_activations)).reshape(-1, self.context_len, len
```

ChatGPT Trace

```

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

```

In this part, you will learn to implement and train the neural language model from Figure 1. As described in the previous section, during training, 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. In practice, this '[MASK]' token is assigned the index 0 in our dictionary. The weights $W^{(2)}$ = 'hid_to_output_weights' now has the shape $N \times V$, as the output layer has V 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 $(v + nV)$ -th column is for the word v in 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:

$$\text{loss} = -\sum_i \sum_n \sum_v m^{(i)}_n t^{(i)}_{(v+nV)} \log y^{(i)}_{(v+nV)}$$

Where:

- * $y^{(i)}_{(v+nV)}$ denotes the output probability prediction from the neural network for the i -th training example for the word v in the n -th output word. Denoting z as the logits output, we define the output probability y as a softmax on z over contiguous chunks of V units (see also Figure 1):

$$y^{(i)}_{(v+nV)} = \frac{\exp(z^{(i)}_{(v+nV)})}{\sum_l \exp(z^{(i)}_{(l+nV)})}$$

$t^{(i)}_{(v+nV)}$ is 1 if for the i -th training example, the word v is the n -th word in context

$m^{(i)}_n$ is a mask that is set to 1 if we are predicting the n -th word position for the i -th example (because we had masked that word in the input), and 0 otherwise

YOUR CODE HERE

return loss

Please help me to build the function without comments or documents

```

scss
Copy code
def compute_loss(output_activations, expanded_target_batch, target_mask):
    loss = -(expanded_target_batch * torch.log(output_activations)).sum() /
    return loss

```

Regenerate response

4.2 6.2 Implement gradient with respect to parameters [1pt] [Type 2]

`back_propagate` is the function which computes the gradient of the loss with respect to model parameters using backpropagation. It uses the derivatives computed by `compute_loss_derivative`. Some parts are already filled in for you, but you need to compute the matrices of derivatives for `embed_to_hid_weights`, `hid_bias`, `hid_to_output_weights`, and `output_bias`. These matrices have the same sizes as the parameter matrices (see previous section). These matrices have the same sizes as the parameter matrices. Look for the `## YOUR CODE HERE ##` comment for where to complete the code.

In order to implement backpropagation efficiently, you need to express the computations in terms of matrix operations, rather than *for* loops. You should first work through the derivatives on pencil and paper. First, apply the chain rule to compute the derivatives with respect to individual units, weights, and biases. Next, take the formulas you've derived, and express them in matrix form. You should be able to express all of the required computations using only matrix multiplication, matrix transpose, and elementwise operations — no *for* loops! If you want inspiration, read through the code for `Model.compute_activations` and try to understand how the matrix operations correspond to the computations performed by all the units in the network.

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

Hint: To prompt a GPT-like model, you may only include functions that are relevant to the implementation in your prompt.

```
[93]: class Model(object):
    """A class representing the language model itself. This class contains
    ↪various methods used in training
    the model and visualizing the learned representations. It has two fields:

        params, a Params instance which contains the model parameters
        vocab, a list containing all the words in the dictionary; vocab[0] is
    ↪the word with index
        0, and so on."""

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

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

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

@classmethod
```

```

def random_init(cls, init_wt, vocab, context_len, embedding_dim, num_hid):
    """Constructor which randomly initializes the weights to Gaussians with
    standard deviation init_wt
    and initializes the biases to all zeros."""
    params = Params.random_init(init_wt, len(vocab), context_len, embedding_dim, num_hid)
    return Model(params, vocab)

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

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

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

def compute_loss_derivative(self, output_activations, expanded_target_batch, target_mask):
    """Compute the gradient of cross-entropy loss wrt output logits z

    For example:

    
$$[y_{0} \dots y_{V-1}] \ [y_V, \dots, y_{2V-1}] \ [y_{2V} \dots y_{3V-1}] \ [y_{3V} \dots y_{4V-1}]$$


    Where for column  $v + n*V$ ,
    
$$y_{v+n*V} = e^{z_{v+n*V}} / \sum_{m=0}^{V-1} e^{z_{m+n*V}},$$

    for  $n=0, \dots, N-1$ 

```

This function should return a dC / dz matrix of size $[batch_size \times (vocab_size * context_len)]$,
 where each row i in dC / dz has columns 0 to $V-1$ containing the gradient the 1st output
 context word from i -th training example, then columns $vocab_size$ to $2*vocab_size - 1$ for the 2nd
 output context word of the i -th training example, etc.

C is the loss function summed across all examples as well:

$C = -\sum_{i,j,n} mask_{i,n} (t_{i,j+n*V} \log y_{i,j+n*V})$,
 for $j=0, \dots, V$, and $n=0, \dots, N$

where $mask_{i,n} = 1$ if the i -th training example has n -th context word as the target,
 otherwise $mask_{i,n} = 0$.

Args:

`output_activations`: A $[batch_size \times (context_len * vocab_size)]$ matrix,
 for the activations of the output layer, i.e. the y_j 's.
`expanded_target_batch`: A $[batch_size \times (context_len * vocab_size)]$ 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.
`target_mask`: A $[batch_size \times context_len \times 1]$ tensor, where
 $target_mask[i,n] = 1$ if for the i 'th example the n -th context word is a target position, otherwise 0

Outputs:

`loss_derivative`: A $[batch_size \times (context_len * vocab_size)]$ 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.

```
"""
# Reshape output_activations and expanded_target_batch and use broadcasting
output_activations_reshape = output_activations.reshape(-1, self.context_len, len(self.vocab))
```

```

        expanded_target_batch_reshape = expanded_target_batch.reshape(-1, self.
˓→context_len, len(self.vocab))
        gradient_masked_reshape = target_mask * (output_activations_reshape -_
˓→expanded_target_batch_reshape)
        gradient_masked = gradient_masked_reshape.reshape(-1, self.context_len_
˓→* len(self.vocab))
    return gradient_masked

    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
    
```

In this part, you will learn to implement and train the neural language model from Figure 1. As described in the previous section, during training, 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. In practice, this `[MASK]` token is assigned the index 0 in our dictionary. The weights $W^{(2)}$ = `hid_to_output_weights` now has the shape $NV \times H$, 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 $(v + nV)$ -th column is for the word v in 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:

$$\text{C} = -\sum_i \sum_n \sum_v m^{(i)}_n (t^{(i)}_v + nV) \log y^{(i)}_{v+nV}$$

Where:

- * $y^{(i)}_{v+nV}$ denotes the output probability prediction from the neural network for the i -th training example for the word v in the n -th output word. Denoting z as the logits output, we define the output probability y as a softmax on z over contiguous chunks of V units (see also Figure 1):

```


$$y^{(i)}_{v+nV} = \frac{e^{z^{(i)}_{v+nV}}}{\sum_l e^{z^{(i)}_{l+nV}}}$$


*  $t^{(i)}_{v+nV}$  is 1 if for the  $i$ -th training example, the word  $v$  is the  $n$ -th word in context
*  $m^{(i)}_n$  is a mask that is set to 1 if we are predicting the  $n$ -th word position for the  $i$ -th example (because we had masked that word in the input), and 0 otherwise

"""

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

# print(output_activations.shape)
# print(expanded_target_batch.shape)
# print(target_mask.shape)
loss = -((expanded_target_batch * np.log(output_activations)).reshape(-1, self.context_len, len(self.vocab)) * target_mask).sum()

#
#####

return loss

def compute_activations(self, inputs):
    """Compute the activations on a batch given the inputs. Returns an Activations instance.

    You should try to read and understand this function, since this will give you clues for
    how to implement back_propagate."""
    batch_size = inputs.shape[0]
    if inputs.shape[1] != self.context_len:

```

```

        raise RuntimeError('Dimension of the input vectors should be {}, but is instead {}'.format(
            self.context_len, inputs.shape[1]))

    # Embedding layer
    # Look up the input word indices in the word_embedding_weights matrix
    embedding_layer_state = self.params.word_embedding_weights[inputs.
        reshape([-1]), :].reshape([batch_size, self.embedding_layer_dim])

    # Hidden layer
    inputs_to_hid = np.dot(embedding_layer_state, self.params.
        embed_to_hid_weights.T) + \
                    self.params.hid_bias
    # Apply logistic activation function
    hidden_layer_state = 1. / (1. + np.exp(-inputs_to_hid))

    # Output layer
    inputs_to_softmax = np.dot(hidden_layer_state, self.params.
        hid_to_output_weights.T) + \
                        self.params.output_bias

    # Subtract maximum.
    # Remember that adding or subtracting the same constant from each input
    # to a
    # softmax unit does not affect the outputs. So subtract the maximum to
    # make all inputs <= 0. This prevents overflows when computing their
    # exponents.
    inputs_to_softmax -= inputs_to_softmax.max(1).reshape((-1, 1))

    # Take softmax along each V chunks in the output layer
    output_layer_state = np.exp(inputs_to_softmax)
    output_layer_state_shape = output_layer_state.shape
    output_layer_state = output_layer_state.reshape((-1, self.context_len,
        len(self.vocab)))
    output_layer_state /= output_layer_state.sum(axis=-1, keepdims=True) # Softmax along vocab of each target word
    output_layer_state = output_layer_state.
    reshape(output_layer_state_shape) # Flatten back to 2D matrix

    return Activations(embedding_layer_state, hidden_layer_state,
        output_layer_state)

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  $j$ th hidden unit,  
# i.e.  $h_j = 1 / (1 + e^{-z_j})$   
hid_deriv = np.dot(loss_derivative, self.params.hid_to_output_weights) \  
           * activations.hidden_layer * (1. - activations.hidden_layer)
```

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

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

```
#####
```

```

hid_bias_grad = hid_deriv.sum(0)

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

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

return Params(word_embedding_weights_grad, embed_to_hid_weights_grad, □
↪hid_to_output_weights_grad,
↪hid_bias_grad, output_bias_grad)

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

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

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

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

    return total / float(ndata)

def display_nearest_words(self, word, k=10):

```

```

"""List the k words nearest to a given word, along with their distances.
"""

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

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

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

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

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

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

```

ChatGPT Trace

```

u # 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]))

```



```

output_bias_grad = loss_derivative.sum(0)

embed_to_hid_weights_grad = np.dot(activations.word_embedding.T, hid_deriv)

```



4.3 6.3 Print the gradients [0.5pt] [Type 4]

To make your life easier, we have provided the routine `check_gradients`, which checks your gradients using finite differences. You should make sure this check passes before continuing with the assignment. Once `check_gradients()` passes, call `print_gradients()` and include its output in your write-up.

```

[94]: def relative_error(a, b):
        return np.abs(a - b) / (np.abs(a) + np.abs(b))

def check_output_derivatives(model, input_batch, target_batch, mask):
    def softmax(z):
        z = z.copy()
        z -= z.max(-1, keepdims=True)
        y = np.exp(z)
        y /= y.sum(-1, keepdims=True)
        return y

```

```

batch_size = input_batch.shape[0]
z = np.random.normal(size=(batch_size, model.context_len, model.vocab_size))
y = softmax(z).reshape((batch_size, model.context_len * model.vocab_size))
z = z.reshape((batch_size, model.context_len * model.vocab_size))

expanded_target_batch = model.indicator_matrix(target_batch)
target_mask = np.expand_dims(mask, axis=2)
loss_derivative = model.compute_loss_derivative(y, expanded_target_batch, target_mask)

if loss_derivative is None:
    print('Loss derivative not implemented yet.')
    return False

if loss_derivative.shape != (batch_size, model.vocab_size * model.context_len):
    print('Loss derivative should be size {} but is actually {}.'.format(
        (batch_size, model.vocab_size), loss_derivative.shape))
    return False

def obj(z):
    z = z.reshape((-1, model.context_len, model.vocab_size))
    y = softmax(z).reshape((batch_size, model.context_len * model.vocab_size))
    return model.compute_loss(y, expanded_target_batch, target_mask)

for count in range(1000):
    i, j = np.random.randint(0, loss_derivative.shape[0]), np.random.randint(0, loss_derivative.shape[1])

    z_plus = z.copy()
    z_plus[i, j] += EPS
    obj_plus = obj(z_plus)

    z_minus = z.copy()
    z_minus[i, j] -= EPS
    obj_minus = obj(z_minus)

    empirical = (obj_plus - obj_minus) / (2. * EPS)
    rel = relative_error(empirical, loss_derivative[i, j])
    if rel > 1e-4:
        print('The loss derivative has a relative error of {}, which is too large.'.format(rel))
        return False

print('The loss derivative looks OK.')
return True

```

```

def check_param_gradient(model, param_name, input_batch, target_batch, mask):
    activations = model.compute_activations(input_batch)
    expanded_target_batch = model.indicator_matrix(target_batch)
    target_mask = np.expand_dims(mask, axis=2)
    loss_derivative = model.compute_loss_derivative(activations.output_layer, □
    ↵expanded_target_batch, target_mask)
    param_gradient = model.back_propagate(input_batch, activations, □
    ↵loss_derivative)

    def obj(model):
        activations = model.compute_activations(input_batch)
        return model.compute_loss(activations.output_layer, □
    ↵expanded_target_batch, target_mask)

    dims = getattr(model.params, param_name).shape
    is_matrix = (len(dims) == 2)

    if getattr(param_gradient, param_name).shape != dims:
        print('The gradient for {} should be size {} but is actually {}.'.
    ↵format(
            param_name, dims, getattr(param_gradient, param_name).shape))
    return

    for count in range(1000):
        if is_matrix:
            slc = np.random.randint(0, dims[0]), np.random.randint(0, dims[1])
        else:
            slc = np.random.randint(dims[0])

        model_plus = model.copy()
        setattr(model_plus.params, param_name)[slc] += EPS
        obj_plus = obj(model_plus)

        model_minus = model.copy()
        setattr(model_minus.params, param_name)[slc] -= EPS
        obj_minus = obj(model_minus)

        empirical = (obj_plus - obj_minus) / (2. * EPS)
        exact = getattr(param_gradient, param_name)[slc]
        rel = relative_error(empirical, exact)
        if rel > 5e-4:
            print('The loss derivative has a relative error of {}, which is too□
    ↵large for param {}.'.format(rel, param_name))
        return False

```

```

print('The gradient for {} looks OK.'.format(param_name))

def load_partially_trained_model():
    obj = pickle.load(open(PARTIALLY_TRAINED_MODEL, 'rb'))
    params = Params(obj['word_embedding_weights'], obj['embed_to_hid_weights'],
                    obj['hid_to_output_weights'], □
    ↵obj['hid_bias'], □
                    obj['output_bias'])
    vocab = obj['vocab']
    return Model(params, vocab)

def check_gradients():
    """Check the computed gradients using finite differences."""
    np.random.seed(0)

    np.seterr(all='ignore') # suppress a warning which is harmless

    model = load_partially_trained_model()
    data_obj = pickle.load(open(data_location, 'rb'))
    train_inputs = data_obj['train_inputs']
    input_batch = train_inputs[:100, :]
    mask = model.sample_input_mask(input_batch.shape[0])
    input_batch_masked = input_batch * (1 - mask)

    if not check_output_derivatives(model, input_batch_masked, input_batch, □
    ↵mask):
        return

    for param_name in ['word_embedding_weights', 'embed_to_hid_weights', □
    ↵'hid_to_output_weights',
                       'hid_bias', 'output_bias']:
        check_param_gradient(model, param_name, input_batch_masked, □
    ↵input_batch, mask)

def print_gradients():
    """Print out certain derivatives for grading."""
    np.random.seed(0)

    model = load_partially_trained_model()
    data_obj = pickle.load(open(data_location, 'rb'))
    train_inputs = data_obj['train_inputs']
    input_batch = train_inputs[:100, :]

    mask = model.sample_input_mask(input_batch.shape[0])

```

```

        input_batch_masked = input_batch * (1 - mask)
        activations = model.compute_activations(input_batch_masked)
        expanded_target_batch = model.indicator_matrix(input_batch)
        target_mask = np.expand_dims(mask, axis=2)
        loss_derivative = model.compute_loss_derivative(activations.output_layer,
        ↪expanded_target_batch, target_mask)
        param_gradient = model.back_propagate(input_batch, activations,
        ↪loss_derivative)

        print('loss_derivative[46, 785]', loss_derivative[46, 785])
        print('loss_derivative[46, 766]', loss_derivative[46, 766])
        print('loss_derivative[5, 42]', loss_derivative[5, 42])
        print('loss_derivative[5, 31]', loss_derivative[5, 31])
        print()
        print('param_gradient.word_embedding_weights[27, 2]', param_gradient.
        ↪word_embedding_weights[27, 2])
        print('param_gradient.word_embedding_weights[43, 3]', param_gradient.
        ↪word_embedding_weights[43, 3])
        print('param_gradient.word_embedding_weights[22, 4]', param_gradient.
        ↪word_embedding_weights[22, 4])
        print('param_gradient.word_embedding_weights[2, 5]', param_gradient.
        ↪word_embedding_weights[2, 5])
        print()
        print('param_gradient.embed_to_hid_weights[10, 2]', param_gradient.
        ↪embed_to_hid_weights[10, 2])
        print('param_gradient.embed_to_hid_weights[15, 3]', param_gradient.
        ↪embed_to_hid_weights[15, 3])
        print('param_gradient.embed_to_hid_weights[30, 9]', param_gradient.
        ↪embed_to_hid_weights[30, 9])
        print('param_gradient.embed_to_hid_weights[35, 21]', param_gradient.
        ↪embed_to_hid_weights[35, 21])
        print()
        print('param_gradient.hid_bias[10]', param_gradient.hid_bias[10])
        print('param_gradient.hid_bias[20]', param_gradient.hid_bias[20])
        print()
        print('param_gradient.output_bias[0]', param_gradient.output_bias[0])
        print('param_gradient.output_bias[1]', param_gradient.output_bias[1])
        print('param_gradient.output_bias[2]', param_gradient.output_bias[2])
        print('param_gradient.output_bias[3]', param_gradient.output_bias[3])

```

[95]: # Run this to check if your implement gradients matches the finite difference
 ↪within tolerance
 # Note: this may take a few minutes to go through all the checks
 check_gradients()

```
/tmp/ipykernel_3247017/4092977173.py:229: DeprecationWarning: `np.int` is a
deprecated alias for the builtin `int`. To silence this warning, use `int` by
itself. Doing this will not modify any behavior and is safe. When replacing
`np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the
precision. If you wish to review your current use, check the release note link
for additional information.

Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
mask = np.zeros((batch_size, self.context_len), dtype=np.int) # Convert to one
hot B x N, B batch size, N context len

The loss derivative looks OK.
The gradient for word_embedding_weights looks OK.
The gradient for embed_to_hid_weights looks OK.
The gradient for hid_to_output_weights looks OK.
The gradient for hid_bias looks OK.
The gradient for output_bias looks OK.
```

```
[96]: # Run this to print out the gradients
print_gradients()
```

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

param_gradient.word_embedding_weights[27, 2] 0.0
param_gradient.word_embedding_weights[43, 3] 0.011596892511489444
param_gradient.word_embedding_weights[22, 4] -0.022267062381729714
param_gradient.word_embedding_weights[2, 5] 0.0

param_gradient.embed_to_hid_weights[10, 2] 0.37932570919301645
param_gradient.embed_to_hid_weights[15, 3] 0.016045161321109152
param_gradient.embed_to_hid_weights[30, 9] -0.4312854367997418
param_gradient.embed_to_hid_weights[35, 21] 0.06679896665436336

param_gradient.hid_bias[10] 0.02342880312334519
param_gradient.hid_bias[20] -0.024370452378874308

param_gradient.output_bias[0] 0.000970106146902794
param_gradient.output_bias[1] 0.1686894627476322
param_gradient.output_bias[2] 0.0051664774143909235
param_gradient.output_bias[3] 0.1509622647181436

/tmp/ipykernel_3247017/4092977173.py:229: DeprecationWarning: `np.int` is a
deprecated alias for the builtin `int`. To silence this warning, use `int` by
itself. Doing this will not modify any behavior and is safe. When replacing
`np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the
precision. If you wish to review your current use, check the release note link
```

```

for additional information.

Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
mask = np.zeros((batch_size, self.context_len), dtype=np.int)# Convert to one
hot B x N, B batch size, N context len

```

To convince us that you have correctly implemented the gradient computations, please include the following with your assignment submission:

- You will submit `a1-code.ipynb` through MarkUs. You do not need to modify any of the code except the parts we asked you to implement.
- In your writeup, include the output of the function `print_gradients`. This prints out part of the gradients for a partially trained network which we have provided, and we will check them against the correct outputs. **Important:** make sure to give the output of `print_gradients`, not `check_gradients`.

5 Part 7: Bias in Word Embeddings (2pts)

Unfortunately, stereotypes and prejudices are often reflected in the outputs of natural language processing algorithms. For example, Google Translate is more likely to translate a non-English sentence to “*He* is a doctor” than “*She* is a doctor when the sentence is ambiguous. In this section, you will explore how bias enters natural language processing algorithms by implementing and analyzing a popular method for measuring bias in word embeddings.

Note: In AI and machine learning, **bias** generally refers to prior information, a necessary prerequisite for intelligent action. However, bias can be problematic when it is derived from aspects of human culture known to lead to harmful behaviour, such as stereotypes and prejudices.

5.1 7.1 WEAT method for detecting bias [1pt] [Type 2]

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 ...).

There is an excellent blog on bias in word embeddings and the WEAT test [here](#).

In the following section, you will run a WEAT test for a given set of target and attribute words. Specifically, you must implement the function `weat_association_score` and then run the remaining cells to compute the p-value and effect size. Before you begin, make sure you understand the formal definition of the WEAT test given in section 4.1 of the handout.

Run the following cell to download pretrained GloVe embeddings.

```
[5]: import gensim.downloader as api

glove = api.load("glove-wiki-gigaword-50")
num_words, num_dims = glove.vectors.shape
print(f"Downloaded {num_words} word embeddings of dimension {num_dims}.")
```

Downloaded 400000 word embeddings of dimension 50.

Before proceeding, you should familiarize yourself with the `similarity` method, which computes the cosine similarity between two words. You will need this method to implement `weat_association_score`. Some examples are given below.

Can you spot the gender bias between occupations in the examples below?

```
[6]: print(glove.similarity("man", "scientist"))
print(glove.similarity("man", "nurse"))
print(glove.similarity("woman", "scientist"))
print(glove.similarity("woman", "nurse"))
```

0.49226812
0.5718704
0.43883625
0.715502

Below, we define our target words (`occupations`) and attribute words (A and B). Our target words consist of *occupations*, and our attribute words are *gendered*. We will use the WEAT test to determine if the word embeddings contain gender biases for certain occupations.

```
[7]: # 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"]
```

- **TODO:** 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: To prompt a GPT-like model, think about how to provide the information that the glove has an method `similarity` which can be called.

```
[8]: 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 #####
    sim_A = [glove.similarity(w, a) for a in A]
```

```

mean_sim_A = sum(sim_A) / len(A)
sim_B = [glove.similarity(w, b) for b in B]
mean_sim_B = sum(sim_B) / len(B)
return mean_sim_A - mean_sim_B
#####

```

Use the following code to check your implementation:

[9]: np.isclose(weat_association_score("programmer", A, B, glove), 0.019615129)

[9]: True

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

[10]: # TODO: Print out the weat association score for each occupation
YOUR CODE HERE #####
for occupation in occupations:
 score = weat_association_score(occupation, A, B, glove)
 print("Association score for", occupation, "is:", score)
#####

```

Association score for programmer is: 0.019615095923654735
Association score for engineer is: 0.05364736542105675
Association score for scientist is: 0.06795817986130714
Association score for nurse is: -0.09486919268965721
Association score for teacher is: -0.018930237740278244
Association score for librarian is: -0.024141353089362383

```

ChatGPT Trace

```

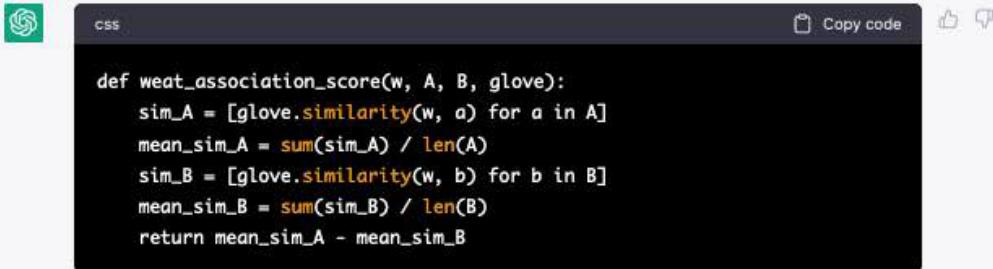
# Target words (occupations)
occupations = ["programmer", "engineer", "scientist", "nurse", "teacher", "librarian"]
# Two sets of gendered attribute words, A and B
A = ["man", "male", "he", "boyish"]
B = ["woman", "female", "she", "girlish"] - [ ] __TODO__: Implement the following function,
`weat_association_score` 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: To prompt a GPT-like model, think about how to provide the information that the `glove` has an
method `similarity` which can be called.* 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 #####

```

```
#####
```



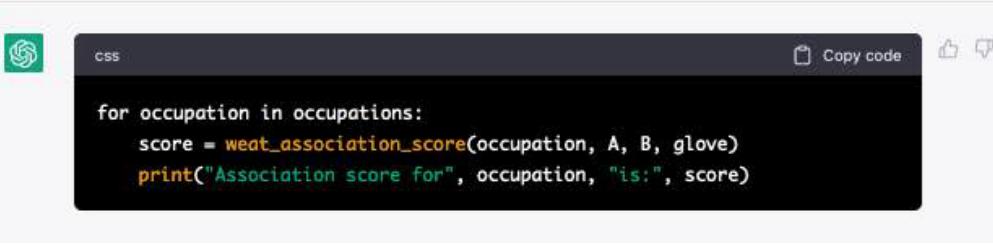
The screenshot shows a code editor window with a dark theme. At the top, there's a toolbar with icons for file operations and a 'Copy code' button. Below the toolbar, the code is displayed:

```

def weat_association_score(w, A, B, glove):
    sim_A = [glove.similarity(w, a) for a in A]
    mean_sim_A = sum(sim_A) / len(A)
    sim_B = [glove.similarity(w, b) for b in B]
    mean_sim_B = sum(sim_B) / len(B)
    return mean_sim_A - mean_sim_B

```

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



The screenshot shows a code editor window with a dark theme. At the top, there's a toolbar with icons for file operations and a 'Copy code' button. Below the toolbar, the code is displayed:

```

for occupation in occupations:
    score = weat_association_score(occupation, A, B, glove)
    print("Association score for", occupation, "is:", score)

```

5.2 7.3 Analyzing WEAT [1pt]

While WEAT makes intuitive sense by asserting that closeness in the embedding space indicates greater similarity, more recent work ([Ethayarajh et al. \[2019\]](#)) has further analyzed the mathematical assertions and found some flaws with this method. Analyzing edge cases is a good way to find logical inconsistencies with any algorithm, and WEAT in particular can behave strangely when A and B contain just one word each.

5.2.1 7.3.1 1-word subsets [0.5 pts] [Type 4]

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

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

# TODO: Print out the weat association score for each word in occupations, with ↴regards to C and D
##### YOUR CODE HERE #####
for i in range(len(C)):
    for j in range(len(D)):
        C_subset = [C[i]]
        D_subset = [D[j]]
        for occupation in occupations:
            score = weat_association_score(occupation, C_subset, D_subset, ↴glove)
            original_score = weat_association_score(occupation, A, B, glove)
            # if the sign of the two scores are different
            if np.sign(score) != np.sign(original_score):
                print("Association score for", occupation, "is:", score)
                print("Original score for", occupation, "is:", original_score)
                print("The 1-word sets are:{} , {} \n".format(C_subset, D_subset))
#####
```

```
Association score for teacher is: 0.058042705059051514
Original score for teacher is: -0.018930237740278244
The 1-word sets are: ['man'] ['female']
Association score for librarian is: 0.06666575372219086
Original score for librarian is: -0.024141353089362383
The 1-word sets are: ['man'] ['female']
Association score for nurse is: 0.35583968460559845
Original score for nurse is: -0.09486919268965721
The 1-word sets are: ['man'] ['girlish']
Association score for teacher is: 0.4468337446451187
Original score for teacher is: -0.018930237740278244
```

```
The 1-word sets are: ['man'] ['girlish']
Association score for librarian is: 0.27873520366847515
Original score for librarian is: -0.024141353089362383
The 1-word sets are: ['man'] ['girlish']
Association score for programmer is: -0.011286124587059021
Original score for programmer is: 0.019615095923654735
The 1-word sets are: ['male'] ['woman']
Association score for engineer is: -0.18472768366336823
Original score for engineer is: 0.05364736542105675
The 1-word sets are: ['male'] ['woman']
Association score for scientist is: -0.16186094284057617
Original score for scientist is: 0.06795817986130714
The 1-word sets are: ['male'] ['woman']
Association score for programmer is: -0.051630899310112
Original score for programmer is: 0.019615095923654735
The 1-word sets are: ['male'] ['female']
Association score for engineer is: -0.08600880205631256
Original score for engineer is: 0.05364736542105675
The 1-word sets are: ['male'] ['female']
Association score for scientist is: -0.06417012214660645
Original score for scientist is: 0.06795817986130714
The 1-word sets are: ['male'] ['female']
Association score for programmer is: -0.02121761441230774
Original score for programmer is: 0.019615095923654735
The 1-word sets are: ['male'] ['she']
Association score for engineer is: -0.21983717381954193
Original score for engineer is: 0.05364736542105675
The 1-word sets are: ['male'] ['she']
Association score for scientist is: -0.13856497406959534
Original score for scientist is: 0.06795817986130714
The 1-word sets are: ['male'] ['she']
Association score for nurse is: 0.3437433987855911
Original score for nurse is: -0.09486919268965721
The 1-word sets are: ['male'] ['girlish']
Association score for teacher is: 0.34907539188861847
Original score for teacher is: -0.018930237740278244
The 1-word sets are: ['male'] ['girlish']
Association score for librarian is: 0.17784558050334454
Original score for librarian is: -0.024141353089362383
The 1-word sets are: ['male'] ['girlish']
Association score for programmer is: -0.00685197114944458
Original score for programmer is: 0.019615095923654735
The 1-word sets are: ['he'] ['female']
Association score for teacher is: 0.08346575498580933
Original score for teacher is: -0.018930237740278244
The 1-word sets are: ['he'] ['female']
Association score for librarian is: 0.15235765278339386
Original score for librarian is: -0.024141353089362383
```

```

The 1-word sets are: ['he'] ['female']
Association score for nurse is: 0.2644778937101364
Original score for nurse is: -0.09486919268965721
The 1-word sets are: ['he'] ['girly']
Association score for teacher is: 0.4722567945718765
Original score for teacher is: -0.018930237740278244
The 1-word sets are: ['he'] ['girly']
Association score for librarian is: 0.36442710272967815
Original score for librarian is: -0.024141353089362383
The 1-word sets are: ['he'] ['girly']
Association score for programmer is: -0.1547197625041008
Original score for programmer is: 0.019615095923654735
The 1-word sets are: ['boyish'] ['woman']
Association score for engineer is: -0.41798314452171326
Original score for engineer is: 0.05364736542105675
The 1-word sets are: ['boyish'] ['woman']
Association score for scientist is: -0.3417776823043823
Original score for scientist is: 0.06795817986130714
The 1-word sets are: ['boyish'] ['woman']
Association score for programmer is: -0.19506453722715378
Original score for programmer is: 0.019615095923654735
The 1-word sets are: ['boyish'] ['female']
Association score for engineer is: -0.3192642629146576
Original score for engineer is: 0.05364736542105675
The 1-word sets are: ['boyish'] ['female']
Association score for scientist is: -0.2440868616104126
Original score for scientist is: 0.06795817986130714
The 1-word sets are: ['boyish'] ['female']
Association score for programmer is: -0.16465125232934952
Original score for programmer is: 0.019615095923654735
The 1-word sets are: ['boyish'] ['she']
Association score for engineer is: -0.45309263467788696
Original score for engineer is: 0.05364736542105675
The 1-word sets are: ['boyish'] ['she']
Association score for scientist is: -0.3184817135334015
Original score for scientist is: 0.06795817986130714
The 1-word sets are: ['boyish'] ['she']
Association score for teacher is: 0.03795386850833893
Original score for teacher is: -0.018930237740278244
The 1-word sets are: ['boyish'] ['girly']
Association score for librarian is: 0.06820434145629406
Original score for librarian is: -0.024141353089362383
The 1-word sets are: ['boyish'] ['girly']

```

5.2.2 7.3.2 How word frequency affects embedding similarity [0.5 pts] [Type 3] [EC]

Consider the fact that the squared norm of a word embedding is linear in the log probability of the word in the training corpus. In other words, the more common a word is in the training corpus,

the larger the norm of its word embedding. (See handout for more thorough description)

Briefly explain how this fact might contribute to the results from the previous section when using different attribute words. Provide your answers in no more than three sentences.

Hint 2: The paper cited above is a great resource if you are stuck.

Answer

This fact shows that the association score in WEAT can be influenced by the frequency of words in the training corpus. The more common a word is in the corpus, the larger its norm in the word embedding, and the larger its impact on the association score. As a result, the results from the previous section could be biased towards more frequent words, leading to inaccuracies in the assessment of gender bias in the embeddings.