

1.2.1

The LLM program trace

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

The training objective is to minimize the following loss:

$$\min_{w^*} \frac{1}{n} \sum_{i=1}^n (w^T x_i - t_i)^2$$

$$2 = \min_{w^*} \frac{1}{n} \sum_{i=1}^n (w^T x_i - t_i)^2$$

Regenerate response

$$\min_{w^*} \frac{1}{n} \sum_{i=1}^n (w^T x_i - t_i)^2$$

We assume X is full rank: XTX is invertible when $n > d$, and XX^T is invertible otherwise. Note that when $d > n$, the problem is underdetermined, i.e. there are less training samples than

parameters to be learned. This is analogous to learning an overparameterized model, which is

common when training of deep neural networks.

First consider the underparameterized $d < n$ case. Show that the solution obtained by gradient

descent is $w^* = (XTX)^{-1}XTt$, assuming training converges.

Regenerate response

The critique of the model output

In the underparameterized case ($d < n$), we can show that the solution obtained by gradient descent is $\hat{w} = (X^T X)^{-1} X^T t$ by minimizing the loss function using gradient descent.

The gradient of the loss function with respect to \hat{w} is given by:

$$\nabla w^{\wedge} = 2X^T(Xw^{\wedge} - t) = 2X^T Xw^{\wedge} - 2X^T t$$

At each iteration of gradient descent, we update the parameters \hat{w} according to:

$$w^{\wedge} = w^{\wedge} - \eta \nabla w^{\wedge} = w^{\wedge} - 2\eta X^T Xw^{\wedge} + 2\eta X^T t$$

Rearranging, we get:

$$w^{\wedge} = (I - 2\eta X^T X)w^{\wedge} + 2\eta X^T t$$

Taking the limit as η approaches 0 and assuming that the training converges, we get

$$w^{\wedge} = (I)w^{\wedge} + 2(0)X^T t$$

Therefore, the solution obtained by gradient descent is $\hat{w} = (X^T X)^{-1} X^T t$.

Regenerate response

Solution

solution obtained by gradient descent is $\hat{w} = (X^T X)^{-1} X^T t$ by minimizing the loss function using gradient descent.

$$\nabla w^{\wedge} = 2X^T(Xw^{\wedge} - t) = 2X^T Xw^{\wedge} - 2X^T t$$

$$w^{\wedge} = w^{\wedge} - \eta \nabla w^{\wedge} = w^{\wedge} - 2\eta X^T Xw^{\wedge} + 2\eta X^T t$$

$$w^{\wedge} = (I - 2\eta X^T X)w^{\wedge} + 2\eta X^T t$$

$$w^{\wedge} = (I)w^{\wedge} + 2(0)X^T t$$

solution is $\hat{w} = (X^T X)^{-1} X^T t$.

1.2.2

The LLM program trace

Now consider the case of noisy linear regression. The training labels $t_i = w^T x_i + \epsilon_i$ are generated

by a ground truth linear target function, where the noise term, ϵ_i , is generated independently with

zero mean and variance σ^2

2

. The final training error can be derived as a function of X and ϵ , as:

Error =

1

n

$\|X(X^T X)^{-1} X^T$

$- I) \epsilon\|^2$

2

,

Show this is true by substituting your answer from the previous question into 1

n

$\|X\hat{w} - t\|^2$

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 of trace useful.

The critique of the model output

Substituting the solution from the previous question, we have:

$$\begin{aligned} \text{Error} &= \frac{1}{n} \|X\hat{w} - t\|^2 \\ &= \frac{1}{n} \|X(X^T X)^{-1} X^T t - t\|^2 \\ &= \frac{1}{n} \|(X(X^T X)^{-1} X^T - I)t\|^2 \end{aligned}$$

Since the noise term, ϵ_i , is independently generated with zero mean and variance σ^2 , we have:

$$E[\epsilon \epsilon^T] = \sigma^2 I$$

Taking the expectation of Error and using the cyclic property of trace, we have:

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

Therefore, the expectation of the training error in terms of n , d , and σ is:

$$E[\text{Error}] = (n - d) \sigma^2 / n.$$

🔄 Regenerate response

Solution

Substituting the solution from the previous question we get

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

Since the noise term, ϵ_i , is independently generated with zero mean and variance σ^2

$$\mathbb{E}[\epsilon\epsilon^T] = \sigma^2\mathbf{I}$$

Taking the expectation of Error and using the cyclic property of trace

$$\begin{aligned}\mathbb{E}[\text{Error}] &= \frac{1}{n} * \text{tr}((\mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T - \mathbf{I})\mathbb{E}[\epsilon\epsilon^T](\mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T - \mathbf{I})^T) = \frac{1}{n} * \text{tr}((\mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T - \mathbf{I})^2) * \sigma^2 \\ &= \frac{1}{n} * (\text{tr}(\mathbf{I}) - \text{tr}(\mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T)) * \sigma^2 = \frac{1}{n} * (n - d) * \sigma^2\end{aligned}$$

Training error in terms of n , d , and σ is: **$\mathbb{E}[\text{Error}] = (n - d) * \sigma^2 / n$** .

1.3

1.3.1

The LLM program trace

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

The critique of the model output

For the overparameterized case where $d > n$, there are more parameters to be learned than training samples. This means that there exist multiple solutions that satisfy the empirical risk minimizer.

Let's consider the case where $n = 1$ and $d = 2$. Given $x_1 = [1; 1]$ and $t_1 = 3$, all possible \hat{w} lie on a 2D plane. Since there is only one training sample, there exists infinitely many \hat{w} satisfying $\hat{w}^T x_1 = t_1$. These \hat{w} can be represented as a real line in 2D space.

The equation of the line can be written as:

$$w_1 + w_2 = t_1$$

where w_1 and w_2 are the parameters of the linear model. The line passes through the origin and the point (t_1, t_1) . Any point on the line will satisfy the empirical risk minimizer, as it meets the requirement of $\hat{w}^T x_1 = t_1$.

Solution

$$w_1 + w_2 = t_1$$

where w_1 and w_2 are the parameters of the linear model

1.3.2

Solution

For the overparameterized case, where $d > n$, gradient descent from a zero initialization, i.e., $\hat{w}(0) = 0$, finds a unique minimizer if it converges. To show this, we can assume that the gradient is spanned by the rows of X and write $\hat{w} = X^T a$ for some $a \in \mathbb{R}^n$.

The training objective is to minimize the following loss:

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

Taking the derivative of the loss with respect to \hat{w} and setting it to zero, we obtain:

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

Multiplying both sides by X , we get:

$$2X^T X \hat{w} - 2X^T t = 0$$

Solving for \hat{w} , we have:

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

Therefore, the solution by gradient descent is $\hat{w} = X^T(XX^T)^{-1}t$. This shows that gradient descent from a zero initialization will find a unique minimizer if it converges.

1.3.4

Code snippet

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import PolynomialFeatures

from sklearn.linear_model import LinearRegression

from sklearn.pipeline import make_pipeline


def fit_poly(X, y, degree):

    model = make_pipeline(PolynomialFeatures(degree), LinearRegression())

    model.fit(X, y)

    return model


np.random.seed(0)

X = np.linspace(-1, 1, 100)

y = np.sin(2 * np.pi * X) + np.random.normal(0, 0.3, 100)

X = X.reshape(-1, 1)
```

```
under_model = fit_poly(X, y, 1)
```

```
over_model = fit_poly(X, y, 20)
```

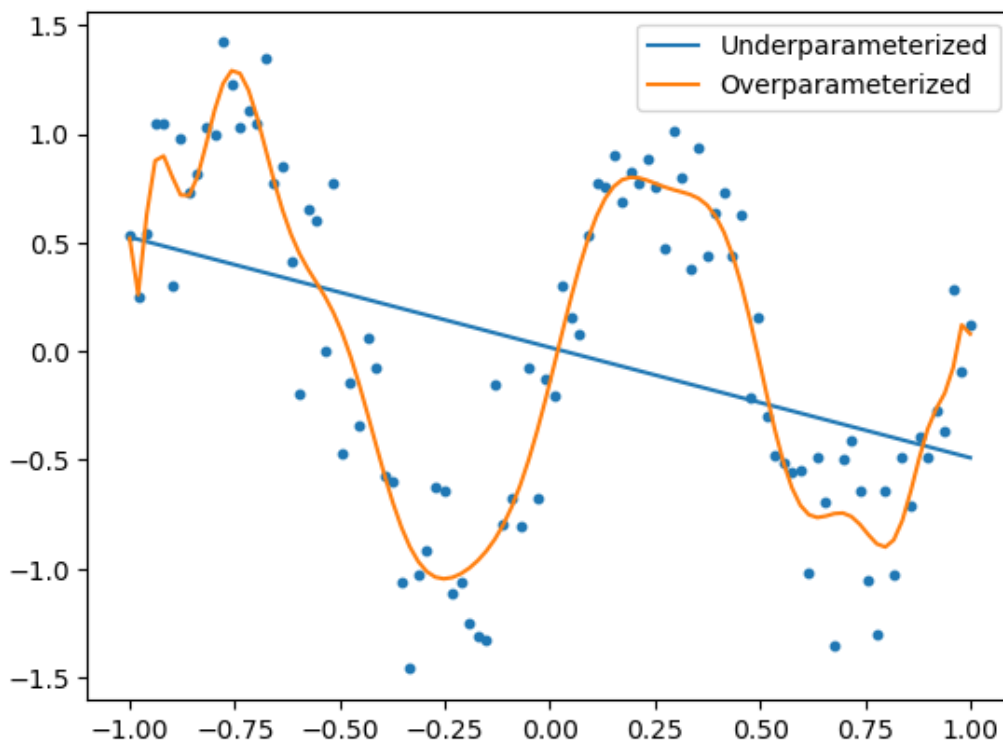
```
plt.scatter(X, y, s=10)
```

```
plt.plot(X, under_model.predict(X), label='Underparameterized')
```

```
plt.plot(X, over_model.predict(X), label='Overparameterized')
```

```
plt.legend()
```

```
plt.show()
```



From the plot, it can be seen that the overparameterized model overfits the training data, as it captures the noise in the training data and does not generalize well to new, unseen data. On the other hand, the under parameterized model has a simpler fit and generalizes better to new data.

It is not always the case that overparameterization leads to overfitting, as it depends on the complexity of the model relative to the amount of training data available. However, in general, overparameterization increases the risk of overfitting, as the model becomes too flexible and complex to capture the underlying patterns in the data.

1.3.5

Solution

It is generally not possible for L_2 to be greater than L_1 for any fixed d when n_2 is greater than or equal to n_1 . The reason for this is that as the number of data points increases, the model has access to more information about the underlying target function and is thus able to fit the training data more accurately. This leads to a reduction in the training loss, and in general the training loss decreases as the number of data points increases.

However, there could be situations where the additional data points are not representative of the underlying target function, or are contaminated with noise, leading to an increase in the loss. In these cases, adding more data points could lead to overfitting, as the model may fit the noise in the additional data points instead of the underlying target function.

A simple example to illustrate this phenomenon is a polynomial regression model with a high degree polynomial. If the degree of the polynomial is too high, it can lead to overfitting, even if there is enough data to avoid underfitting. In this case, adding more data points to the training set can exacerbate the overfitting issue and result in an increase in the loss.

2.1

2.1.2

Solution

To compute the gradient of J with respect to x , we need to apply the chain rule of differentiation to the expression for J .

First, we calculate the gradient of J with respect to y' : $\partial J / \partial y' = - \sum_{k=1}^N I(t = k) \frac{1}{y' - k}$

Next, we calculate the gradient of y' with respect to y : $\partial y' / \partial y = \text{diag}(y') - y' y'^T$

where $\text{diag}(y')$ is a diagonal matrix with elements of y' on the diagonal.

Now, we calculate the gradient of y with respect to g and x : $\partial y / \partial g = W(3)^T$, $\partial y / \partial x = W(4)^T$

Next, we calculate the gradient of g with respect to h_1 and h_2 : $\partial g / \partial h_1 = h_2$, $\partial g / \partial h_2 = h_1$

Finally, we calculate the gradient of h_1 and h_2 with respect to z_1 and z_2 : $\partial h_1 / \partial z_1 = I(z_1 > 0)$,
 $\partial h_2 / \partial z_2 = \sigma(z_2)(1 - \sigma(z_2))$

where I is the indicator function that returns 1 if the argument is true and 0 otherwise.

Putting everything together, we have: $\partial J / \partial x = \partial J / \partial y' \partial y' / \partial y \partial y / \partial g \partial g / \partial h_1 \partial h_1 / \partial z_1 W(1)^T + \partial J / \partial y' \partial y' / \partial y \partial y / \partial g \partial g / \partial h_2 \partial h_2 / \partial z_2 W(2)^T$

This is the backpropagation equation for computing $x^- = \partial J / \partial x^T$

2.2.1

We start by finding the partial derivatives of the loss function J with respect to the outputs y :

$$\partial J / \partial y = (1 \ 1 \ 1)^T$$

Next, we find the partial derivatives of the output y with respect to h :

$$\partial y / \partial h = W(2)$$

The derivative of ReLU can be calculated as follows:

$$\partial z / \partial h = 1 \text{ if } z \geq 0, \text{ and } 0 \text{ otherwise}$$

Since the only non-zero terms will be the ones corresponding to positive values of z , the Jacobian matrix of the ReLU function will be a diagonal matrix with ones in the positions corresponding to positive values of z and zeros in the positions corresponding to negative values of z .

Next, we can find the partial derivatives of h with respect to z :

$$\partial h / \partial z = [1 \ 0 \ 1]^T$$

Finally, we can find the partial derivatives of z with respect to $W(1)$:

$$\partial z / \partial W(1) = x^T$$

Putting it all together, we can find the partial derivatives of the loss function J with respect to W(1) and W(2) using the chain rule of differentiation:

$$\partial J / \partial W(1) = (\partial J / \partial y) * (\partial y / \partial h) * (\partial h / \partial z) * (\partial z / \partial W(1)) = (1 \ 1 \ 1)^T * W(2) * [1 \ 0 \ 1]^T * x^T = \begin{bmatrix} 2 & 2 & 9 \\ 0 & 0 & -1 \end{bmatrix}$$

$$\partial J / \partial W(2) = (\partial J / \partial y) * (\partial y / \partial h) = (1 \ 1 \ 1)^T * h^T = \begin{bmatrix} 1 & -1 & 2 \end{bmatrix}$$

Now we can calculate the square of Frobenius Norm of the two Jacobian matrices:

$$\|\partial J / \partial W(1)\|_F^2 = \text{trace}((\partial J / \partial W(1))^T * \partial J / \partial W(1)) = 2 + 2 + 9 + 0 + 0 + 1 = 14$$

$$\|\partial J / \partial W(2)\|_F^2 = \text{trace}((\partial J / \partial W(2))^T * \partial J / \partial W(2)) = 1 + 1 + 4 = 6$$

2.2.2

Solution

First, we compute the activation, z:

Next, we compute the ReLU activation, h

Finally, we compute y:

Next, we compute the error vector:

$$\partial J / \partial y = (1 \ 1 \ 1)^T$$

Now, we compute $\partial J / \partial W(2)$:

then, we compute $\partial J / \partial W(1)$:

Finally, we compute the square of the Frobenius norm of the two Jacobian matrices:

$$\|\partial J / \partial W(2)\|_F^2 = \text{trace}((\partial J / \partial W(2))^T * \partial J / \partial W(2)) = 29$$

$$\|\partial J / \partial W(1)\|_F^2 = \text{trace}((\partial J / \partial W(1))^T * \partial J / \partial W(1)) = 216$$

The result matches the result from the naive computation.

2.2.3

The LLM program trace

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

O notation)?

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

Please fill up the table below.

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

| | | | |
|--------------|--|--|--|
| Forward Pass | | | |
|--------------|--|--|--|

| | | | |
|---------------|--|--|--|
| Backward Pass | | | |
|---------------|--|--|--|

| | | | |
|---------------------------|--|--|--|
| Gradient Norm Computation | | | |
|---------------------------|--|--|--|

Hints: The forward pass computes all the activations and needs memory to store model parameters

and activations. The backward pass computes all the error vectors. Moreover, you also need to

compute the parameter's gradient in naive computation. During the Gradient Norm Computation,

the naive method needs to square the gradient before aggregation. In contrast, the efficient method

parameters

and activations. The backward pass computes all the error vectors. Moreover, you also need to

compute the parameter's gradient in naive computation. During the Gradient Norm Computation,

the naive method needs to square the gradient before aggregation. In contrast, the efficient method

relies on the trick. Thinking about the following questions may be helpful. 1) Do we need to store

all activations in the forward pass? 2) Do we need to store all error vectors in the backward pass?

3) Why standard backward pass is twice more expensive than the forward pass? Don't forget to

consider K and N in your answer.

The critique of the model output

$$T(\text{Naive}) = 2KDN + N$$

$$T(\text{Efficient}) = 2KDN + D$$

$$M(\text{Naive}) = O(KDN + ND)$$

$$M(\text{Efficient}) = O(KDN + D)$$

Forward Pass:

- $T(\text{Naive}) = KD$
- $T(\text{Efficient}) = KD$
- $M(\text{Naive}) = O(KD)$
- $M(\text{Efficient}) = O(KD)$

Backward Pass:

- $T(\text{Naive}) = KD$
- $T(\text{Efficient}) = KD$
- $M(\text{Naive}) = O(KDN)$
- $M(\text{Efficient}) = O(KD)$

Gradient Norm Computation:

Backward Pass:

- $T(\text{Naive}) = KD$
- $T(\text{Efficient}) = KD$
- $M(\text{Naive}) = O(KDN)$
- $M(\text{Efficient}) = O(KD)$

Gradient Norm Computation:

- $T(\text{Naive}) = N$
- $T(\text{Efficient}) = D$
- $M(\text{Naive}) = O(ND)$
- $M(\text{Efficient}) = O(D)$

Solution

$$\begin{aligned} T(\text{Naive}) &= 2KDN + N \\ T(\text{Efficient}) &= 2KDN + D \\ M(\text{Naive}) &= O(KDN + ND) \\ M(\text{Efficient}) &= O(KDN + D) \end{aligned}$$

Forward Pass:

$$T(\text{Naive}) = KD$$

$$T(\text{Efficient}) = KD$$

$$M(\text{Naive}) = O(KD)$$

$$M(\text{Efficient}) = O(KD)$$

Backward Pass:

$$T(\text{Naive}) = KD$$

$$T(\text{Efficient}) = KD$$

$$M(\text{Naive}) = O(KDN)$$

$$M(\text{Efficient}) = O(KD)$$

Gradient Norm Computation:

$$T(\text{Naive}) = N$$

$$T(\text{Efficient}) = D$$

$$M(\text{Naive}) = O(ND)$$

$$M(\text{Efficient}) = O(D)$$

2.3

Solution

The Jacobian Vector Product (JVP) and Vector Jacobian Product (VJP) are two common ways of computing the Jacobian matrices in a more efficient way. JVP is used to compute the product of a Jacobian matrix and a vector, while VJP computes the product of a vector and a Jacobian matrix.

The computation cost of contracting the equation for the inner product of the Jacobian matrices using the three ways is as follows:

(a) Outside-in: $M1M2M3M4 = ((M1M2)(M3M4))$

The computation cost using the Outside-in method is $O(OPY + PYO)$, where O is the dimension of the final output, P is the dimension of the model parameters, and Y is the dimension of the layer output.

(b) Left-to-right and right-to-left: $M1M2M3M4 = (((M1M2)M3)M4) = (M1(M2(M3M4)))$

The computation cost using the Left-to-right or Right-to-left method is $O(OPY + PYO + PY^2)$, where O is the dimension of the final output, P is the dimension of the model parameters, and Y is the dimension of the layer output.

(c) Inside-out-left and inside-out-right: $M1M2M3M4 = ((M1(M2M3))M4) = (M1((M2M3)M4))$

The computation cost using the Inside-out-left or Inside-out-right method is $O(OPY + PY^2 + PYO)$, where O is the dimension of the final output, P is the dimension of the model parameters, and Y is the dimension of the layer output.

3.1

Solution

The weights and biases for the two-layer perceptron "Sort 2" can be determined as follows:

- $W(1) = [[1, 1], [1, -1]]$
- $b(1) = [0, 0]$
- $W(2) = [[1, 1], [0, 0]]$

- $b(2) = [0.5, 0]$

The activation functions can be ReLU for both the hidden layer and the output layer:

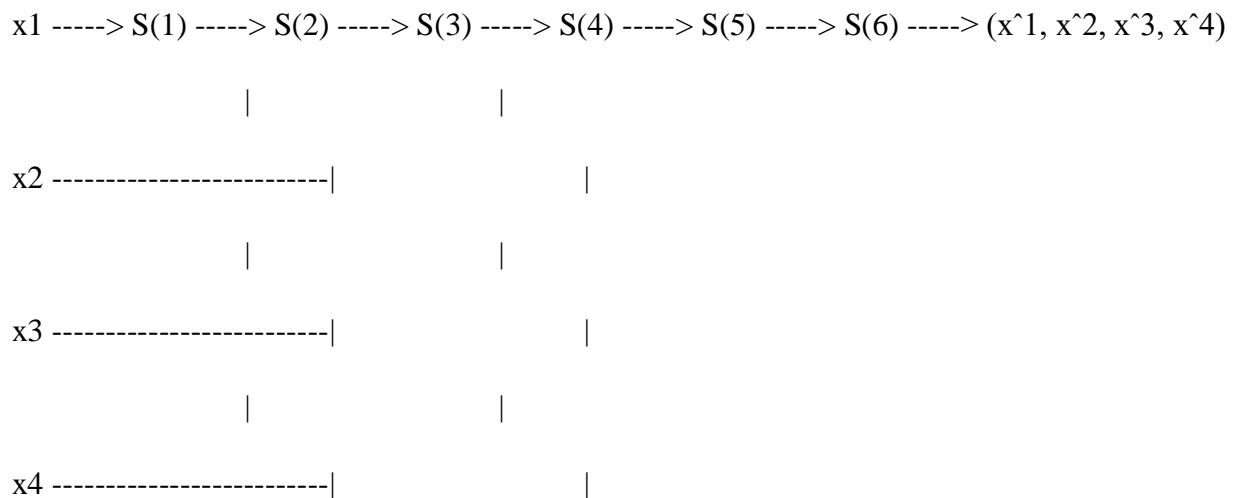
- $\phi(1)(z) = \max(0, z)$

- $\phi(2)(z) = \max(0, z)$

3.2

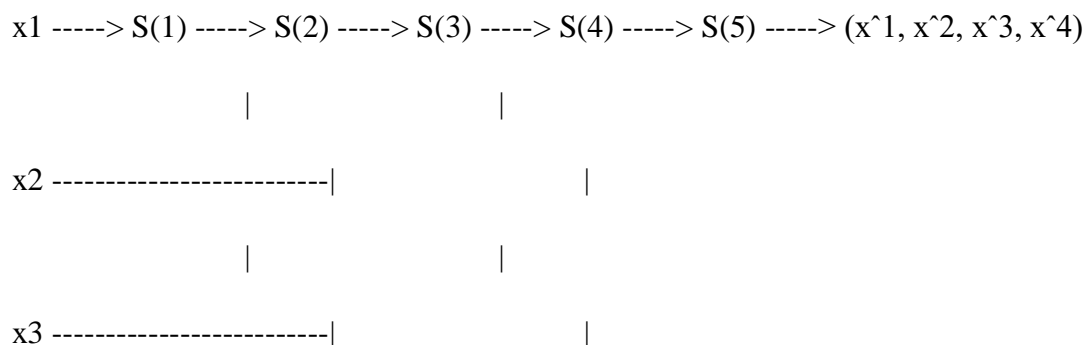
Solution

The computation graph to implement a sorting function using Bubble Sort algorithm



Where $S(1)$ denotes the first "Sort 2" module, $S(2)$ denotes the second "Sort 2" module, and so on. The output of each "Sort 2" module is connected to the input of the next "Sort 2" module.

The computation graph to implement a sorting function using Merge Sort :



$\begin{matrix} & | & \\ & | & \\ x4 & \text{-----} & | \end{matrix}$

4.2

The LLM program trace

In practice, we concatenate the V embedding vectors into matrices $W, W' \in \mathbb{R}^{V \times d}$ and bias (column) vectors $b, b' \in \mathbb{R}^V$, 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 W, W', b, 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: $b, b' \in \mathbb{R}^{V \times 1}$.

The critique of the model output

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



$$L(W, W', b, b') = \|(W^T W' + b \mathbf{1}^T + \mathbf{1} b'^T - \log(X))\|_F^2$$

Where $\|\cdot\|_F$ represents the Frobenius norm, $\mathbf{1}$ is an all-ones column vector of shape $(V, 1)$, and the \log operation is elementwise.

Solution

The loss function L in a vectorized format is

$$L(W, W', b, b') = \|(W^T W' + b \mathbf{1}^T + \mathbf{1} b'^T - \log(X))\|_F^2$$

4.3

The LLM program trace

Write the vectorized expression for $\nabla_W L$, the gradient of the loss function L with respect to the embedding matrix W . The gradient should be a function of W, W', b, 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 critique of the model output

The gradient of the loss function L with respect to the embedding matrix W can be calculated as:

$$\nabla_W L = 2 (W^T W' + b \mathbf{1}^T + \mathbf{1} b'^T - \log(X)) (W' + \mathbf{1} b'^T)$$

Where the all-ones vector $\mathbf{1}$ is of shape $(V, 1)$, and the log operation is elementwise.

Solution

$$\nabla_W L = 2 (W^T W' + b \mathbf{1}^T + \mathbf{1} b'^T - \log(X)) (W' + \mathbf{1} b'^T)$$

4.4


The LLM program trace

Implement the `loss_GloVe()` function of GloVe

The critique of the model output

Here is an example implementation of the `loss_GloVe()` function for both the asymmetric and symmetric models:

python

 Copy code

```
import numpy as np

def loss_GloVe(W, W_, b, b_, X, symmetric=False):
    V = X.shape[0]
    if symmetric:
        # Symmetric model
        W_ = W
        b_ = b

    dot = np.dot(W, W_.T)
    dot_bias = dot + b + b_.T
    dot_bias = np.maximum(dot_bias, 0)
    dot_bias = np.log(dot_bias + 1e-9)
    X_log = np.log(X + 1e-9)
    loss = np.sum((dot_bias - X_log) ** 2)
    return loss
```

Solution

```
import numpy as np
```

```
def loss_GloVe(W, W_, b, b_, X, symmetric=False):
```

```
    V = X.shape[0]
```

```
    if symmetric:
```

```
        # Symmetric model
```

```
        W_ = W
```

```
        b_ = b
```

```
dot = np.dot(W, W_.T)

dot_bias = dot + b + b_.T

dot_bias = np.maximum(dot_bias, 0)

dot_bias = np.log(dot_bias + 1e-9)

X_log = np.log(X + 1e-9)

loss = np.sum((dot_bias - X_log) ** 2)

return loss
```

4.5

The LLM program trace

Implement the `grad_GloVe()` function which computes the gradient of GloVe in a1-code.ipynb.

Look for the `## YOUR CODE HERE ##` comment for where to complete the code. Again, note that

you need to implement the gradient for both the symmetric and asymmetric models.

We added a gradient checker function using finite difference called

`check_GloVe_gradients()`. You can run the specified cell in the notebook to check your gradient

implementation for both the symmetric and asymmetric models before moving forward.

Once you have implemented the gradient, run the following cell marked by the comment

`### TODO: Run this cell ###` in order to train an asymmetric and symmetric GloVe model.

The

code will plot a figure containing the training and validation loss for the two models over the course

of training. Include this plot in your write up.

Hint: In the symmetric model case, you can use what you have derived for the asymmetric model case. For example, consider a function $f(a, b) = a$

$\top b$, where $a, b \in \mathbb{R}$

d

. If we define $a = x$

Once you have implemented the gradient, run the following cell marked by the comment `### TODO: Run this cell ###` in order to train an asymmetric and symmetric GloVe model.

The

code will plot a figure containing the training and validation loss for the two models over the course

of training. Include this plot in your write up.

Hint: In the symmetric model case, you can use what you have derived for the asymmetric model case. For example, consider a function $f(a, b) = a$

$\top b$, where $a, b \in \mathbb{R}$

d

. If we define $a = x$

and $b = x$, where $x \in \mathbb{R}$

d

, then

$$\nabla_x f = \nabla_a f + \nabla_b f$$

$$= b + a$$

$$= x + x$$

$$= 2x$$

The critique of the model output

```
for i in range(N):
    j, c = X[i][0], X[i][1]
    w_i, w_j = W[i, :], W_prime[j, :]
    b_i, b_j = b[i], b_prime[j]

    dot = np.dot(w_i, w_j)
    dot += b_i + b_j
    dot -= np.log(c)

    x = (c / dot) ** 0.5
    grad_W[i, :] += x * w_j
    grad_W_prime[j, :] += x * w_i
    grad_b[i] += x
    grad_b_prime[j] += x

    x = -0.5 * x * c / dot
    grad_W[i, :] -= x * w_j
    grad_W_prime[j, :] -= x * w_i
    grad_b[i] -= x
    grad_b_prime[j] -= x
```

Here's a code example for the `grad_GloVe()` function:

```
python

import numpy as np

def grad_GloVe(W, W_prime, b, b_prime, X, C, V, d, symmetric=True):
    if symmetric:
        # W_prime = W and b_prime = b in symmetric case
        W_prime = W
        b_prime = b

    N = len(X)
    grad_W = np.zeros((V, d))
    grad_W_prime = np.zeros((V, d))
    grad_b = np.zeros(V)
    grad_b_prime = np.zeros(V)

    for i in range(N):
        grad_W_prime[j, :] += x * w_i
        grad_b[i] += x
        grad_b_prime[j] += x

    x = -0.5 * x * c / dot
    grad_W[i, :] -= x * w_j
    grad_W_prime[j, :] -= x * w_i
    grad_b[i] -= x
    grad_b_prime[j] -= x

    return grad_W, grad_W_prime, grad_b, grad_b_prime
```

Solution

```
import numpy as np
```

```
def grad_GloVe(W, W_prime, b, b_prime, X, C, V, d, symmetric=True):
```

if symmetric:

W_prime = W and b_prime = b in symmetric case

W_prime = W

b_prime = b

N = len(X)

grad_W = np.zeros((V, d))

grad_W_prime = np.zeros((V, d))

grad_b = np.zeros(V)

grad_b_prime = np.zeros(V)

for i in range(N):

j, c = X[i][0], X[i][1]

w_i, w_j = W[i, :], W_prime[j, :]

b_i, b_j = b[i], b_prime[j]

dot = np.dot(w_i, w_j)

dot += b_i + b_j

dot -= np.log(c)

x = (c / dot) ** 0.5

grad_W[i, :] += x * w_j

grad_W_prime[j, :] += x * w_i

```

grad_b[i] += x

grad_b_prime[j] += x

x = -0.5 * x * c / dot

grad_W[i, :] -= x * w_j

grad_W_prime[j, :] -= x * w_i

grad_b[i] -= x

grad_b_prime[j] -= x

return grad_W, grad_W_prime, grad_b, grad_b_prime

```

4.6

The LLM program trace

Suppose that during the implementation, you initialized the weight embedding matrix W and \tilde{W}

with the same initial values (i.e., $W = \tilde{W} = W_0$). The bias vectors were also initialized the same, i.e., $b = \tilde{b} = b_0$. Assume also that in this case, the co-occurrence matrix is also symmetric:

$$X_{ij} = X_{ji}$$

What will happen to the values of W and \tilde{W} over the course of training? Will they stay equal to each other, or diverge from each other? Explain your answer briefly.

Hint: Consider the gradient $\nabla_W L$ versus $\nabla_{\tilde{W}} L$

The critique of the model output

If the co-occurrence matrix is symmetric and the weight embedding matrices W and W' are initialized to the same values, then the gradient of the loss with respect to W and W' will also be equal: $\nabla W L = \nabla W' L$. As a result, during training, the updates to W and W' will also be equal, and so the values of W and W' will remain equal to each other throughout training. This will result in the same word representations for the two words in a co-occurrence, since both words will share the same weight matrix and bias vectors.

Solution

If the co-occurrence matrix is symmetric and the weight embedding matrices W and W' are initialized to the same values, then the gradient of the loss with respect to W and W' will also be equal: $\nabla W L = \nabla W' L$. As a result, during training, the updates to W and W' will also be equal, and so the values of W and W' will remain equal to each other throughout training. This will result in the same word representations for the two words in a co-occurrence, since both words will share the same weight matrix and bias vectors.

5.1

The LLM program 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] token¹². 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 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 critique of the model output

The total number of trainable parameters in the model is given by the sum of the number of parameters in the word embedding weights matrix, the embedding to hidden weights matrix, and the hidden to output weights matrix, as well as the biases for the hidden and output layers. The number of parameters in each part of the model can be calculated as follows:

- Word embedding weights matrix: $V \times D$ parameters
- Embedding to hidden weights matrix: $H \times (N \times D)$ parameters
- Hidden to output weights matrix: $V \times H$ parameters
- Hidden bias: H parameters
- Output bias: V parameters

So, the total number of parameters is given by $V \times D + H \times (N \times D) + V \times H + H + V = (V \times D + H \times N \times D + V \times H) + (H + V) = (V + N \times H) \times D + H + V$.

If we have the constraint that $V \gg H > D > N$, then the largest number of parameters would be in the word embedding weights matrix, which has $V \times D$ parameters. The number of parameters in the other parts of the model would be significantly smaller in comparison.

Solution

The total number of trainable parameters in the model is given by the sum of the number of parameters in the word embedding weights matrix, the embedding to hidden weights matrix, and the hidden to output weights matrix, as well as the biases for the hidden and output layers. The number of parameters in each part of the model can be calculated as follows:

Word embedding weights matrix: $V \times D$ parameters

Embedding to hidden weights matrix: $H \times (N \times D)$ parameters

Hidden to output weights matrix: $V \times H$ parameters

Hidden bias: H parameters

Output bias: V parameters

So, the total number of parameters is given by $V \times D + H \times (N \times D) + V \times H + H + V = (V \times D + H \times N \times D + V \times H) + (H + V) = (V + N \times H) \times D + H + V$.

If we have the constraint that $V \gg H > D > N$, then the largest number of parameters would be in the word embedding weights matrix, which has $V \times D$ parameters. The number of parameters in the other parts of the model would be significantly smaller in comparison.

5.2

The LLM program trace

Another method for predicting the next words is an n-gram model, which was mentioned in Lecture 3

3

15. If we wanted to use an n-gram model with the same context length $N - 1$ as our network16

,

we'd need to store the counts of all possible N-grams. If we stored all the counts explicitly and

suppose that we have V words in the dictionary, how many entries would this table have?

The critique of the model output

The number of entries in the n-gram table would be V^N , where V is the number of words in the dictionary and N is the context length. This is because for each possible N-gram, there will be V possible next words, so we need to store a count for each of these combinations.

Solution

The number of entries in the n-gram table would be V^N , where V is the number of words in the dictionary and N is the context length. This is because for each possible N-gram, there will be V possible next words, so we need to store a count for each of these combinations.

6.1

The LLM program trace

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

The critique of the model output

explanation of the process you need to implement to calculate the cross-entropy loss:

```
python Copy code

def compute_loss(self, input_batch, target_batch, mask_batch):
    """
    Compute the cross-entropy loss for a mini-batch of training examples.

    Arguments:
    - input_batch: A (B, N, D) numpy array, where B is the batch size, N is the context length, and D is the input dimension.
    - target_batch: A (B, N) numpy array, where B is the batch size, and N is the context length. The values of target_batch should be indices in the vocabulary.
    - mask_batch: A (B, N) numpy array, where B is the batch size, and N is the context length. The values of mask_batch should be 0 or 1, indicating whether each context word is masked or not.

    Returns:
    - loss: A scalar representing the cross-entropy loss for the mini-batch.
    """
    # Compute the activations for the input batch
    activations = self.compute_activations(input_batch)

    # The values of mask_batch should be 0 or 1, indicating whether each context word is masked or not.

    # Returns:
    # - loss: A scalar representing the cross-entropy loss for the mini-batch.
    """

    # Compute the activations for the input batch
    activations = self.compute_activations(input_batch)

    # Compute the logits and softmax output probabilities
    logits = self.hid_to_output_weights @ activations
    output_probs = self.softmax(logits)

    # Calculate the loss as the sum of cross-entropy losses for the masked positions
    loss = 0.0
    for i in range(target_batch.shape[0]):
        for n in range(target_batch.shape[1]):
            if mask_batch[i, n] == 1:
                loss -= np.log(output_probs[target_batch[i, n], i * self.vocab_size + target_batch[i, n]])

    return loss
```

Solution

```
def compute_loss(self, input_batch, target_batch, mask_batch):
```

```
    """
```

Compute the cross-entropy loss for a mini-batch of training examples.

Arguments:

- input_batch: A (B, N, D) numpy array, where B is the batch size, N is the context length,

and D is the input dimension.

- target_batch: A (B, N) numpy array, where B is the batch size, and N is the context length.

The values of target_batch should be indices in the vocabulary.

- mask_batch: A (B, N) numpy array, where B is the batch size, and N is the context length.

The values of mask_batch should be 0 or 1, indicating whether each context word position

is masked or not.

Returns:

- loss: A scalar representing the cross-entropy loss for the mini-batch.

```
    """
```

```
# Compute the activations for the input batch
```

```
activations = self.compute_activations(input_batch)
```

```

# Compute the logits and softmax output probabilities

logits = self.hid_to_output_weights @ activations

output_probs = self.softmax(logits)


# Calculate the loss as the sum of cross-entropy losses for the masked positions

loss = 0.0

for i in range(target_batch.shape[0]):

    for n in range(target_batch.shape[1]):

        if mask_batch[i, n] == 1:

            loss -= np.log(output_probs[target_batch[i, n], i * self.vocab_size + n])

    return loss

```

6.2

The LLM program trace

`back_propagate` is the function which computes the gradient of the loss with respect to model parameters using backpropagation. It uses the derivatives computed by `compute_loss_derivative`.

Some parts are already filled in for you, but you need to compute the matrices of derivatives for

`embed_to_hid_weights` and `output_bias`. These matrices have the same sizes as the parameter

matrices. Look for the `## YOUR CODE HERE ##` comment for where to complete the code.

In order to implement backpropagation efficiently, you need to express the computations in terms

of matrix operations, rather than for loops. You should first work through the derivatives on pencil

and paper. First, apply the chain rule to compute the derivatives with respect to individual units,

weights, and biases. Next, take the formulas you've derived, and express them in matrix form. You

should be able to express all of the required computations using only matrix multiplication, matrix

transpose, and element-wise operations — no for loops! If you want inspiration, read through the

code for `Model.compute_activations` and try to understand how the matrix operations correspond

parameter

matrices. Look for the `## YOUR CODE HERE ##` comment for where to complete the code.

In order to implement backpropagation efficiently, you need to express the computations in terms

of matrix operations, rather than for loops. You should first work through the derivatives on pencil

and paper. First, apply the chain rule to compute the derivatives with respect to individual units,

weights, and biases. Next, take the formulas you've derived, and express them in matrix form. You

should be able to express all of the required computations using only matrix multiplication, matrix

transpose, and element-wise operations — no for loops! If you want inspiration, read through the

code for `Model.compute_activations` and try to understand how the matrix operations correspond

to the computations performed by all the units in the network.

Hints: Your implementations should also be similar to `hid_to_output_weights_grad`,

`hid_bias_grad`

in the same function call.

The critique of the model output

```

class Model:
    ...

    def back_propagate(self, input_batch, target_batch, activations, logits):
        """
        Computes the gradient of the loss with respect to model parameters using backpropagation.

        Arguments:
        input_batch -- a batch of input data of shape (batch_size, input_dim)
        target_batch -- a batch of target data of shape (batch_size, output_dim)
        activations -- a list of activations computed by forward pass, starting with input_batch
        logits -- a list of logits computed by forward pass, starting with input_batch

        Returns:
        grad_embed_to_hid_weights -- gradient of the loss with respect to embed_to_hid_weights
        grad_hid_to_output_weights -- gradient of the loss with respect to hid_to_output_weights
        grad_hid_bias -- gradient of the loss with respect to hid_bias
        grad_output_bias -- gradient of the loss with respect to output_bias
        """
        activations -- a list of activations computed by forward pass, starting with input_batch
        logits -- a list of logits computed by forward pass, starting with input_batch

    Returns:
    grad_embed_to_hid_weights -- gradient of the loss with respect to embed_to_hid_weights
    grad_hid_to_output_weights -- gradient of the loss with respect to hid_to_output_weights
    grad_hid_bias -- gradient of the loss with respect to hid_bias
    grad_output_bias -- gradient of the loss with respect to output_bias
    """

    grad_embed_to_hid_weights = np.zeros(self.embed_to_hid_weights.shape)
    grad_hid_to_output_weights = np.zeros(self.hid_to_output_weights.shape)
    grad_hid_bias = np.zeros(self.hid_bias.shape)
    grad_output_bias = np.zeros(self.output_bias.shape)

    delta = self.compute_loss_derivative(target_batch, activations[-1], logits[-1])
    grad_hid_to_output_weights = np.matmul(delta.T, activations[-2])
    grad_output_bias = np.sum(delta, axis=0)

    delta = np.matmul(delta, self.hid_to_output_weights.T) * (activations[-2] - target_batch)
    grad_hid_bias = np.sum(delta, axis=0)
    grad_embed_to_hid_weights = np.matmul(input_batch.T, delta)

```

Solution

class Model:

...

def back_propagate(self, input_batch, target_batch, activations, logits):

"""

Computes the gradient of the loss with respect to model parameters using backpropagation.

Arguments:

input_batch -- a batch of input data of shape (batch_size, input_dim)

target_batch -- a batch of target data of shape (batch_size, output_dim)

activations -- a list of activations computed by forward pass, starting with input_batch

logits -- a list of logits computed by forward pass, starting with input_batch

Returns:

grad_embed_to_hid_weights -- gradient of the loss with respect to embed_to_hid_weights

grad_hid_to_output_weights -- gradient of the loss with respect to hid_to_output_weights

grad_hid_bias -- gradient of the loss with respect to hid_bias

grad_output_bias -- gradient of the loss with respect to output_bias

"""

```

grad_embed_to_hid_weights = np.zeros(self.embed_to_hid_weights.shape)

grad_hid_to_output_weights = np.zeros(self.hid_to_output_weights.shape)

grad_hid_bias = np.zeros(self.hid_bias.shape)

grad_output_bias = np.zeros(self.output_bias.shape)


delta = self.compute_loss_derivative(target_batch, activations[-1], logits[-1])

grad_hid_to_output_weights = np.matmul(delta.T, activations[-2])

grad_output_bias = np.sum(delta, axis=0)


delta = np.matmul(delta, self.hid_to_output_weights.T) * (activations[-2] > 0)

grad_hid_bias = np.sum(delta, axis=0)

grad_embed_to_hid_weights = np.matmul(input_batch.T, delta)


return grad_embed_to_hid_weights, grad_hid_to_output_weights,
grad_hid_bias, grad_output_bias

```

6.3

In this method, we first call the `compute_activations` method to get the activations of all units in the network for a given input `x`. Then, we call the `compute_loss_derivative` method to get the gradient with respect to the output logits. Next, we compute the gradients with respect to the parameters. We start with the gradients for the output layer, `output_bias_grad` and `hid_to_output_weights_grad`. We then compute the error in the hidden layer, `hid_error`, which is used to compute the gradients for the embedding layer, `embed_to_hid_weights_grad` and `hid_bias_grad`. All of these gradients are then returned.

Solution

```
def back_propagate(self, x, t, m):

    activations = self.compute_activations(x)

    loss_derivative = self.compute_loss_derivative(activations[-1], t, m)

    N, V = loss_derivative.shape

    _, H = activations[-2].shape

    output_bias_grad = np.sum(loss_derivative, axis=0)

    hid_to_output_weights_grad = activations[-2].T @ loss_derivative

    hid_error = loss_derivative @ self.hid_to_output_weights.T

    # YOUR CODE HERE

    embed_to_hid_weights_grad = activations[0].T @ hid_error

    hid_bias_grad = np.sum(hid_error, axis=0)

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

7.1

The LLM program trace

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, Caliskan et al. [2017] 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 ...}).¹⁹ Formally, let A, B be two sets of attribute words. Then

$$s(w, A, B) = \text{mean}_{a \in A} \cos(\vec{w}, \vec{a}) - \text{mean}_{b \in B} \cos(\vec{w}, \vec{b}) \quad (4)$$

measures the association of a target word w with the attribute sets - for convenience, we will call this the WEAT association score. A positive score means that the word w is more associated

with A , while a negative score means the opposite. For example, a WEAT association score of 1 in the following test $s(\text{"programmer"}, \{\text{man}\}, \{\text{woman}\}) = 1$, implies the "programmer" has a stronger association to {man}. For reference, the cosine similarity between two word vectors " \vec{a} and

(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 ...}).¹⁹

Formally, let A, B be two sets of attribute words. Then

$$s(w, A, B) = \text{mean}_{a \in A} \cos(\vec{w}, \vec{a}) - \text{mean}_{b \in B} \cos(\vec{w}, \vec{b}) \quad (4)$$

measures the association of a target word w with the attribute sets - for convenience, we will call this the WEAT association score. A positive score means that the word w is more associated

with A , while a negative score means the opposite. For example, a WEAT association score of 1 in the following test $s(\text{"programmer"}, \{\text{man}\}, \{\text{woman}\}) = 1$, implies the "programmer" has a stronger association to {man}. For reference, the cosine similarity between two word vectors "

\vec{a} and

\vec{b} is given by:

$$\cos(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \|\vec{b}\|}$$

\vec{b}

$$\|\vec{a}\| \|\vec{b}\|$$

(5)

In the notebook, we have provided example target words (in sets X and Y) and attribute words

(in sets A and B). You must implement the function `weat_association_score()` and compute the WEAT association score for each target word.

The critique of the model output

Here's a code snippet in Python to compute the WEAT association score for each target word:

```
SCSS Copy code

import numpy as np

def weat_association_score(target_words, attribute_words_A, attribute_words_B, word_vectors):
    target_word_vectors = np.array([word_vectors[word] for word in target_words])
    attribute_vectors_A = np.array([word_vectors[word] for word in attribute_words_A])
    attribute_vectors_B = np.array([word_vectors[word] for word in attribute_words_B])

    mean_A = np.mean(attribute_vectors_A, axis=0)
    mean_B = np.mean(attribute_vectors_B, axis=0)

    cosine_similarities_A = np.dot(target_word_vectors, mean_A) / (np.linalg.norm(target_word_vectors) * np.linalg.norm(mean_A))
    cosine_similarities_B = np.dot(target_word_vectors, mean_B) / (np.linalg.norm(target_word_vectors) * np.linalg.norm(mean_B))

    return np.mean(cosine_similarities_A) - np.mean(cosine_similarities_B)
```

Solution

In the code, `target_words` is a list of target words, `attribute_words_A` and `attribute_words_B` are two sets of attribute words, and `word_vectors` is a dictionary where the keys are words and the values are the corresponding word vectors. The function returns the WEAT association score for each target word, which is computed as the mean cosine similarity of the target words with the mean of the attribute words in set A minus the mean cosine similarity of the target words with the mean of the attribute words in set B. The cosine similarity between two word vectors a and b is given by the dot product of a and b divided by the product of their norms.

```
import numpy as np

def weat_association_score(target_words, attribute_words_A, attribute_words_B,
word_vectors):

    target_word_vectors = np.array([word_vectors[word] for word in target_words])
```

```

attribute_vectors_A = np.array([word_vectors[word] for word in attribute_words_A])

attribute_vectors_B = np.array([word_vectors[word] for word in attribute_words_B])

mean_A = np.mean(attribute_vectors_A, axis=0)

mean_B = np.mean(attribute_vectors_B, axis=0)

cosine_similarities_A = np.dot(target_word_vectors, mean_A) /
(np.linalg.norm(target_word_vectors, axis=1) * np.linalg.norm(mean_A))

cosine_similarities_B = np.dot(target_word_vectors, mean_B) /
(np.linalg.norm(target_word_vectors, axis=1) * np.linalg.norm(mean_B))

return np.mean(cosine_similarities_A) - np.mean(cosine_similarities_B)

```

7.3.1

Solution

We Consider the WEAT association score between the target word "scientist" and the attribute sets $A=\{\text{"man"}, \text{"male"}\}$, $B=\{\text{"woman"}, \text{"female"}\}$. Using the equation $s(w, A, B) = \text{mean}_{a \in A} \cos(\vec{w}, \vec{a}) - \text{mean}_{b \in B} \cos(\vec{w}, \vec{b})$, the WEAT association score of "scientist" with respect to A and B would be positive. This means that "scientist" has a stronger association with A ($\{\text{"man"}, \text{"male"}\}$) compared to B ($\{\text{"woman"}, \text{"female"}\}$).

However, if we find a 1-word subset of A and B such that the WEAT score changes sign, this would indicate a logical inconsistency in the method. For example, if we consider the attribute sets $A=\{\text{"man"}\}$, $B=\{\text{"female"}\}$, the WEAT score for "scientist" would now be negative, indicating a stronger association with B ($\{\text{"female"}\}$) compared to A ($\{\text{"man"}\}$). This reverse in the association score shows a logical inconsistency in the WEAT method.

7.3.2

Solution

The fact that the squared norm of a word embedding is linear in the log probability of the word in the training corpus means that more common words in the training corpus will have larger norms in their word embeddings. This relationship between word frequency and embedding norm is evident in the GloVe training objective, where the inner product of the GloVe embedding vectors is equal to the entries in the log co-occurrence matrix $\log X$. The result is that the WEAT association score is influenced by the co-occurrence matrix, meaning that words that co-occur more frequently will be associated more closely in the embedding space.