Assignment 3

1 Concepts, intuitions and big picture

1. Suppose you have built a neural network. You decide to initialize the weights and biases to be zero. Which of the following statements are True? (Check all that apply)

   A. Each neuron in the first hidden layer will perform the same computation. So even after multiple iterations of gradient descent each neuron will be computing the same thing as other neurons in the same layer.

   B. Each neuron in the first hidden layer will perform the same computation in the first iteration. But after one iteration of gradient descent they will learn to compute different things because we have "broken symmetry".

   C. Each neuron in the first hidden layer will compute the same thing, but neurons in different layers will compute different things, thus we have accomplished "symmetry breaking" as described in lecture.

   D. The first hidden layer's neurons will perform different computations from each other even in the first iteration; their parameters will thus keep evolving in their own way.

   Answer: A

2. Vectorization allows you to compute forward propagation in an L-layer neural network without an explicit for-loop (or any other explicit iterative loop) over the layers $l = 1, 2, \ldots, L$. True/False?

   A. True
   B. False

   Answer: B (False)

3. The tanh activation usually works better than sigmoid activation function for hidden units because the mean of its output is closer to zero, and so it centers the data better for the next layer. True/False?

   A. True
   B. False

   Answer: A (True)

4. Which of the following techniques does NOT prevent a model from overfitting?

A. Data augmentation
B. Dropout
C. Early stopping
D. None of the above

Answer: D

5. Why should dropout be applied during training? Why should dropout NOT be applied during evaluation?

   Answer: During neural network training, dropout is used as a technique to prevent overfitting. It randomly deactivates neurons, prevents neurons from becoming co-dependent, and pushes the network to leanr more robust features that are not dependent on a single neuron's output. The effects of dropout is like training an ensemble of mulitple networks, or training multiple neural paths in a single network. This makes the nwetwork generalize better.

   During evaluation, the intention is evaluate a deterministic or reliable prediction of the trained model. As such, using dropout for this would make the predictions random, since different neurons would be dropped out each time. This will make the evaluation results inconsistent. Therefore, all neurons are kept active during evaluation. However, their weights are scaled down by the dropout factor to account for the fact that more neurons are active during evaluation than during training. This is referred to as inverse dropout.

6. Explain why initializing the parameters of a neural net with a constant is a bad idea.

   Answer: Initializing all the weights of a neural network to the sam constant is a bad idea because it introduces problems with symmetry breaking. All the neurons or nodes in the network becaome identical and perform the same computations for any given input. This makes them redundant to each other and prevents them from learning the full spectrum of patterns or features in the training data.

7. You design a fully connected neural network architecture where all activations are sigmoids. You initialize the weights with large positive numbers. Is this a good idea? Explain your answer.

   Answer: NO, it's a terrible idea. For large inputs (positive or negative), the gradient of the sigmoid function function is a very small value, close to zero. This is refered to as a squashing effect. As such, when a neural network's weights are initialized to large positive values, the weighted sum of these inputs is also likely to a large postive value, thus pushing the sigmoid outputs to the range where their gradients are near zero. During backpropagation, these small gradients are multiplied down through the earlier layers, and will likely cause the layers in those layers to become even smaller. This is refered to as the vanishing gradient problem, and it can

significnatly slowdown or even halt the networks learning process.

8. Explain what is the importance of “residual connections”.

   Answer: Residual connections (or skip connections) are very important in the training of deep neural networks. These are used to create information flow paths between earlier and later layers of the network. In particular, they enable the direct flow of gradients from the later layers to the earlier ones during backpropagation. This allows the gradients to skip some intermediate layers and avoid being reduced in between. In deep neural networks that have many hidden layers, avoiding gradient reduction or diminishment by skipping some hidden layers during backpropagation is critical in avoiding the vanishing gradient problem.

9. What is cached (“memoized”) in the implementation of forward propagation and backward propagation?

   A. Variables computed during forward propagation are cached and passed on to the corresponding backward propagation step to compute derivatives.

   B. Caching is used to keep track of the hyperparameters that we are searching over, to speed up computation.

   C. Caching is used to pass variables computed during backward propagation to the corresponding forward propagation step. It contains useful values for forward propagation to compute activations.

   Answer: A

10. Which of the following statements is true?

    A. The deeper layers of a neural network are typically computing more complex features of the input than the earlier layers.

    B. The earlier layers of a neural network are typically computing more complex features of the input than the deeper layers.

    Answer: A

2 Revisiting Jacobians

Recall that Jacobians are generalizations of multi-variate derivatives and are extremely useful in denoting the gra- dient computations in computation graph and Backpropagation. A potentially confusing aspect of using Jacobains is their dimensions and so, here we’re going focus on understanding Jacobian dimensions.

Recap: Let’s first recap the formal definition of Jacobian. Suppose $f : \mathbb{R}^n \to \mathbb{R}^m$ is a

function that takes a point $x \in \mathbb{R}^n$ as input and produces the vector $f(x) \in \mathbb{R}^m$ as output. Then the Jacobian matrix of f is defined to be an $m \times n$ matrix, denoted by $J_f(x)$, whose (i,j)-th entry is $J_{ij} = \frac{\partial f_i}{\partial x_j}$, or:

$$J = \begin{pmatrix} \frac{\partial f}{\partial x_1} & \cdots & \frac{\partial f}{\partial x_n} \end{pmatrix} = \begin{bmatrix} \nabla^\top f_1 \\ \vdots \\ \nabla^\top f_m \end{bmatrix} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}$$

Examples: The shape of a Jacobian is an important notion to note. A Jacobian can be a vector, a matrix, or a tensor of arbitrary ranks. Consider the following special cases:

- If f is a scalar and w is a $d \times 1$ column vector, the Jacobian of f with respect to w is a row vector with $1 \times d$ dimensions.
- If y is an $n \times 1$ column vector and z is a $d \times 1$ column vector, the Jacobian of z with respect to y, or $J_z(y)$ is a $d \times n$ matrix.
- Suppose $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{l \times p \times q}$. Then the Jacobian $J_A(B)$ is a tensor of shape $(m \times n) \times (l \times p \times q)$. More broadly, the shape of the Jacobian is determined as (shape of the output) $\times$ (shape of the input).

Problem setup: Suppose we have:

- X, an $n \times d$ matrix, $x_i \in \mathbb{R}^{d \times 1}$ correspond to the rows of $X = [x_1, \ldots, x_n]^\top$
- Y, an $n \times k$ matrix
- W, a $k \times d$ matrix and w, a $d \times 1$ vector

For the following items, compute (1) the shape of each Jacobian, and (2) an expression for each Jacobian:

1. $f(w) = c$ (constant)

   Answer:

   The shape of the Jacobian $J_f(w)$ is $1 \times d$.

   Since $f(w)$ is a constant, its partial derivative with respect to any element of w is 0.

   $J_f(w) = \begin{pmatrix} 0 & 0 & \cdots & 0 \end{pmatrix}$

2. $f(w) = \|w\|^2$ (squared L2-norm)

   Answer:

   The shape of the Jacobian $J_f(w)$ is $1 \times d$.

   The partial derivative of $f(w)$ with respect to each element w_j of the vector w is

   $J_f(w) = \begin{pmatrix} 2w_1 & 2w_2 & \cdots & 2w_d \end{pmatrix} = 2w^\top$

3. $f(w) = w^\top x_i$ (vector dot product)

Answer:

The shape of the Jacobian $J_f(w)$ is $1 \times d$.

Here, $f(w)$ is the dot product of the vector w and the vector x_i. w is a $d \times 1$ column vector, and x_i is also a $d \times 1$ column vector. $f(w)$ is a scalar output.

$w^\top x_i = \sum_{j=1}^{d} w_j x_{ij}$, where x_{ij} is the j-th element of the vector x_i.

The partial derivative of $f(w)$ with respect to each element w_j of the vector w is

$\frac{\partial f(w)}{\partial w_j} = \frac{\partial}{\partial w_j}\left(\sum_{k=1}^{d} w_k x_{ik}\right) = x_{ij}$

The Jacobian matrix is a row vector where the j-th element is $\frac{\partial f(w)}{\partial w_j}$:

$$J_f(w) = \begin{pmatrix} x_{i1} & x_{i2} & \cdots & x_{id} \end{pmatrix} = x_i^\top$$

4. $f(w) = Xw$ (matrix-vector product)

Answer:

The shape of the Jacobian $J_f(w)$ is $n \times d$.

Here, $f(w)$ is the product of an $n \times d$ matrix X and a $d \times 1$ column vector w. The result $f(w)$ is an $n \times 1$ column vector.

The (i, j)-th entry of the Jacobian matrix $J_f(w)$ is $\frac{\partial f_i(w)}{\partial w_j}$.

$\frac{\partial f_i(w)}{\partial w_j} = x_{ij}$

The Jacobian matrix is an $n \times d$ matrix where the (i, j)-th entry is x_{ij}. This is essentially the matrix X.

$$J_f(w) = X$$

5. $f(w) = w$ (vector identity function)$

Answer:

The shape of the Jacobian $J_f(w)$ is $d \times d$.

Here, the function $f(w)$ is an identity function that simply returns the input vector w. The input w is a $d \times 1$ column vector, and the output $f(w)$ is also a $d \times 1$ column vector.

The (i, j)-th entry of the Jacobian matrix $J_f(w)$ is:

$\frac{\partial f_i(w)}{\partial w_j} = \frac{\partial w_i}{\partial w_j}$

This is 1 if $i = j$ and 0 if $i \neq j$.

$$J_f(w) = I_d$$

where I_d is the $d \times d$ identity matrix.

6. $f(w) = w^2$ (element-wise power)$

Answer:

The shape of the Jacobian $J_f(w)$ is $d \times d$.

Here, the function $f(w)$ takes a $d \times 1$ column vector w and squares each element individually. The result is a $d \times 1$ column vector where the i-th element is w_i^2.

The (i, j)-th entry of the Jacobian matrix $J_f(w)$ is:

$\frac{\partial f_i(w)}{\partial w_j} = \frac{\partial (w_i^2)}{\partial w_j}$

This is $2w_j$ if $i = j$ and 0 if $i \neq j$

The Jacobian matrix is a diagonal matrix with $2w_j$ as the j-th diagonal entry:

$$J_f(w) = \begin{pmatrix} 2w_1 & 0 & \cdots & 0 \\ 0 & 2w_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 2w_d \end{pmatrix}$$

7. Extra Credit: $f(W) = XW^\top$ (matrix multiplication)$

Answer:

The input W has shape (k, d) and the output $f(W)$ has shape (n, k). The shape of the Jacobian $J_f(W)$ is the concatenation of the output shape and the input shape, which is (n, k, k, d).

Here, the function $f(W)$ takes an $n \times d$ matrix X and a $k \times d$ matrix W, and the output is an $n \times k$ matrix $f(W)$.

The (i, j)-th element of the output matrix $f(W)$ is:

$$f(W)_{ij} = \sum_{p=1}^{d} X_{ip}(W^\top)_{pj} = \sum_{p=1}^{d} X_{ip}W_{jp}$$

The (i, j, k, l)-th entry of the Jacobian tensor $J_f(W)$ is the partial derivative of the (i, j)-th element of the output matrix with respect to the (k, l)-th element of the input

matrix W:

$$J_{ijkl} = \frac{\partial f(W)_{ij}}{\partial W_{kl}} = \frac{\partial}{\partial W_{kl}} \left(\sum_{p=1}^{d} X_{ip} W_{jp} \right)$$

The partial derivative is non-zero only when $j = k$ and $p = l$.

$$J_{ijkl} = X_{il} \cdot \mathbf{1}_{j=k}$$

3 Activations Per Layer, Keeps Linearity Away!

Based on the content we saw at the class lectures, answer the following:

1. Why are activation functions used in neural networks?

   Answer:

   Activation functions are used to introduce non-linearity in neural networks. They apply a non-linear transformation to the output of each neuron, thereby allowing the network to learn and approximate arbitrary non-linear functions and decision boundaries. This is essential in order for neural networks to be able to learn the complex patterns and relationships that exist in real-world datasets, and to solve complex tasks like image recognition or natural language processing.

2. Write down the formula for three common activation functions (sigmoid, ReLU, Tanh) and their derivatives (assume scalar input/output). Plot these activation functions and their derivatives on $(-\infty, +\infty)$.

   Answer:

Sigmoid Formula:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Derivative:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

ReLU (Rectified Linear Unit) Formula:

$$\mathrm{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

Derivative:

$$\mathrm{ReLU}'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z < 0 \end{cases}$$

(The derivative is undefined at $z = 0$, but it is commonly taken to be 0 or 1 in practice).

Tanh (Hyperbolic Tangent) Formula:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

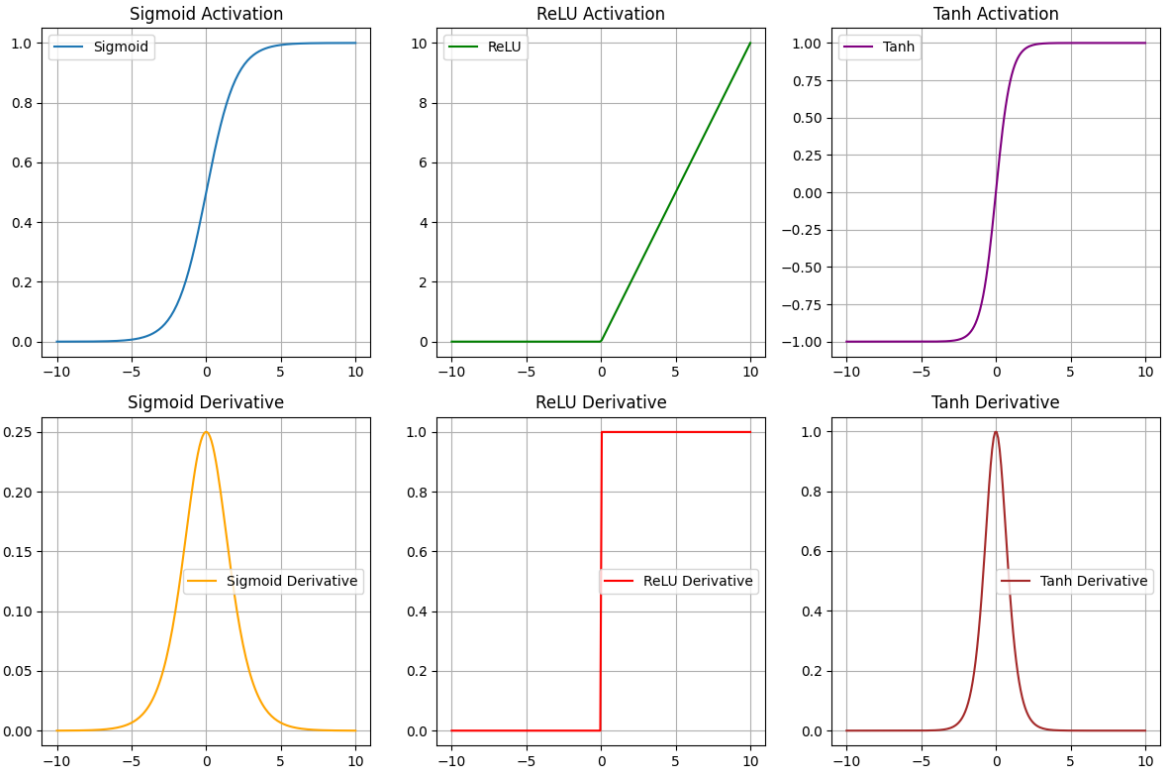
Derivative:

$$\tanh'(z) = 1 - (\tanh(z))^2$$

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt

# Define the activation functions
def sigmoid(z):
  return 1 / (1 + np.exp(-z))

def sigmoid_derivative(z):
  s = sigmoid(z)
  return s * (1 - s)

def relu(z):
  return np.maximum(0, z)

def relu_derivative(z):
  return np.where(z > 0, 1, 0) # Derivative is 1 for z > 0, 0 otherwise

def tanh(z):
  return np.tanh(z)

def tanh_derivative(z):
  return 1 - np.tanh(z)**2

# Generate a range of z values
z = np.linspace(-10, 10, 200)

# Plotting
plt.figure(figsize=(12, 8))

# Sigmoid and its derivative
plt.subplot(2, 3, 1)
plt.plot(z, sigmoid(z), label='Sigmoid')
plt.title('Sigmoid Activation')
plt.grid(True)
plt.legend()

plt.subplot(2, 3, 4)
plt.plot(z, sigmoid_derivative(z), label='Sigmoid Derivative', color='ora
plt.title('Sigmoid Derivative')
plt.grid(True)
plt.legend()

# ReLU and its derivative
plt.subplot(2, 3, 2)
plt.plot(z, relu(z), label='ReLU', color='green')
plt.title('ReLU Activation')
plt.grid(True)
plt.legend()
```

```
plt.subplot(2, 3, 5)
plt.plot(z, relu_derivative(z), label='ReLU Derivative', color='red')
plt.title('ReLU Derivative')
plt.grid(True)
plt.legend()

# Tanh and its derivative
plt.subplot(2, 3, 3)
plt.plot(z, tanh(z), label='Tanh', color='purple')
plt.title('Tanh Activation')
plt.grid(True)
plt.legend()

plt.subplot(2, 3, 6)
plt.plot(z, tanh_derivative(z), label='Tanh Derivative', color='brown')
plt.title('Tanh Derivative')
plt.grid(True)
plt.legend()

plt.tight_layout()
plt.show()
```



3. What is the “vanishing gradient” problem? (respond in no more than 3 sentences) Which activation functions are subject to this issue and why? (respond in no more than 3 sentences).

   Answer: The vanishing gradient problem occurs during neural network trainig when gradients from the later layers thata are already small, get increasingly smaller as they flow back through the earlier layers of the network. As teh gradients get too small, approaching zero, their effects on the neurons become negligible, and the network's learning slows down or halts. Activation functions that are more susceptible to this are Sigmoid and Tanh, as their gradients are close to zero for very high positive and negative number inputs.

4. Why zero-centered activation functions impact the results of Backprop?

   Answer: Zero-centered activation functions (e.g. Tanh) are those whose output can be both positive and negative, thus resulting in means that are around zero. Thanks to this, zero-centered activations allow for more more diverse gradient directions, which can result in more efficient and stable updates to the weights during backpropagation, and possibly accelerate training.

5. Remember the Softmax function $\sigma(\mathbf{z})$ and how it extends sigmoid to multiple dimensions? Let’s compute the derivative of Softmax for each dimension. Prove that:

$$\frac{d\sigma(\mathbf{z})_i}{dz_j} = \sigma(\mathbf{z})_i(\delta_{ij} - \sigma(\mathbf{z})_j)$$

   where δ_{ij} is the Kronecker delta function.

   Answer:

The Softmax function for a vector $\mathbf{z} = (z_1, z_2, \ldots, z_K)$:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$$

The partial derivative of $\sigma(\mathbf{z})_i$ with respect to z_j when $i = j$ and when $i \neq j$.

When $i = j$

$$\frac{\partial\sigma(\mathbf{z})_i}{\partial z_i} = \frac{\partial}{\partial z_i}\left(\frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}\right)$$

Using quotient rule with $u = e^{z_i}$ and $v = \sum_{k=1}^{K} e^{z_k}$:

$$\frac{\partial \sigma(\mathbf{z})_i}{\partial z_i} = \frac{e^{z_i}\left(\sum_{k=1}^{K} e^{z_k}\right) - e^{z_i}(e^{z_i})}{\left(\sum_{k=1}^{K} e^{z_k}\right)^2} = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}} - \left(\frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}\right)^2 = \sigma(\mathbf{z})_i - \sigma(\mathbf{z})_i^2$$

When $i \neq j$

$$\frac{\partial \sigma(\mathbf{z})_i}{\partial z_j} = \frac{\partial}{\partial z_j}\left(\frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}\right)$$

Using quotient rule with $u = e^{z_i}$ and $v = \sum_{k=1}^{K} e^{z_k}$:

$$\frac{\partial \sigma(\mathbf{z})_i}{\partial z_j} = \frac{0 \cdot \left(\sum_{k=1}^{K} e^{z_k}\right) - e^{z_i}(e^{z_j})}{\left(\sum_{k=1}^{K} e^{z_k}\right)^2} = -\frac{e^{z_i}e^{z_j}}{\left(\sum_{k=1}^{K} e^{z_k}\right)^2} = -\frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}} \cdot \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} =$$

Conclusion

The Kronecker delta function δ_{ij} is 1 if $i = j$ and 0 if $i \neq j$.

For $i = j$: $\sigma(\mathbf{z})_i(\delta_{ii} - \sigma(\mathbf{z})_i) = \sigma(\mathbf{z})_i(1 - \sigma(\mathbf{z})_i)$, which matches the answer shown above for $i = j$.

For $i \neq j$: $\sigma(\mathbf{z})_i(\delta_{ij} - \sigma(\mathbf{z})_j) = \sigma(\mathbf{z})_i(0 - \sigma(\mathbf{z})_j) = -\sigma(\mathbf{z})_i\sigma(\mathbf{z})_j$, which matches that for $i \neq j$.

Thus, $\frac{d\sigma(\mathbf{z})_i}{dz_j} = \sigma(\mathbf{z})_i(\delta_{ij} - \sigma(\mathbf{z})_j)$.

6. Use the above point to prove that the Jacobian of the Softmax function is the following:

   $$J_\sigma(\mathbf{z}) = \text{diag}(\sigma(\mathbf{z})) - \sigma(\mathbf{z})\sigma(\mathbf{z})^\top$$

   where $\text{diag}(\cdot)$ turns a vector into a diagonal matrix. Also, note that $J_\sigma(\mathbf{z}) \in \mathbb{R}^{K \times K}$.

   Answer:

From question 3.5, the (i, j)-th entry of the Jacobian matrix $J_\sigma(\mathbf{z})$ is given by:

$$(J_\sigma(\mathbf{z}))_{ij} = \frac{d\sigma(\mathbf{z})_i}{dz_j} = \sigma(\mathbf{z})_i(\delta_{ij} - \sigma(\mathbf{z})_j)$$

This can be expanded to:

$$(J_\sigma(\mathbf{z}))_{ij} = \sigma(\mathbf{z})_i\delta_{ij} - \sigma(\mathbf{z})_i\sigma(\mathbf{z})_j$$

The first term, $\sigma(\mathbf{z})_i\delta_{ij}$ is non-zero only when $i = j$. When $i = j$, the term is $\sigma(\mathbf{z})_i$, which is a diagonal matrix with $\sigma(\mathbf{z})$ on the diagonal, i.e. $\text{diag}(\sigma(\mathbf{z}))$.

The second term, $\sigma(\mathbf{z})_i\sigma(\mathbf{z})_j$, is the (i, j)-th entry of the outer product of the vector

$\sigma(\mathbf{z})$ with itself, i.e. $\sigma(\mathbf{z})\sigma(\mathbf{z})^\top$.

Since the (i, j)-th entry of $J_\sigma(\mathbf{z})$ is the difference of the (i, j)-th entries of these two matrices, the Jacobian matrix is:

$$J_\sigma(\mathbf{z}) = \mathrm{diag}(\sigma(\mathbf{z})) - \sigma(\mathbf{z})\sigma(\mathbf{z})^\top$$

4 Simulating XOR

1. Can a single-layer network simulate (represent) an XOR function on $x = [x_1, x_2]$?

   $$y = \mathrm{XOR}(x) = \begin{cases} 1, & \text{if } x = (0,1) \text{ or } x = (1,0) \\ 0, & \text{if } x = (1,1) \text{ or } x = (0,0) \end{cases}$$

   Explain your reasoning using the following single-layer network definition:
   $\hat{y} = \mathrm{ReLU}(W \cdot x + b)$

   Answer:

   No, this cannot be done with a single-layer network with a ReLU activation function.

   A single straight line cannot be drawn to perfectly deliniate the points of the XOR function into two separate classes, for outputs 0 and 1.

   $\hat{y} = \mathrm{ReLU}(W \cdot x + b)$ computes a linear combination of the inputs $z = w_1 x_1 + w_2 x_2 + b$, and then applies the ReLU function $\hat{y} = \max(0, z)$.

   This will result in a single linear decision boundary defined by $w_1 x_1 + w_2 x_2 + b = 0$. While ReLU introduces non-linearity in the output value, the resulting decision boundary still remains linear. This is easier to see from a plot

2. Repeat (1) with a two-layer network:

   $$h = \mathrm{ReLU}(W_1 \cdot x + b_1)$$
   $$\hat{y} = W_2 \cdot h + b_2$$

   Note that this model has an additional layer compared to the earlier question: an input layer $x \in \mathbb{R}^2$, a hidden layer h with ReLU activation functions that are applied component-wise, and a linear output layer, resulting in scalar prediction $\hat{y}$. Provide a set of weights W_1 and W_2 and biases b_1 and b_2 such that this model can accurately model the XOR problem.

   Answer:

Yes, a two-layer network with a non-linear activation function in the hidden layer can simulate the XOR function.

See the illustration below:

Layer 1:

$$W_1 = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$
$$b_1 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

Layer 2:

$$W_2 = \begin{pmatrix} 1 & -2 \end{pmatrix}$$
$$b_2 = \begin{pmatrix} 0 \end{pmatrix}$$

Demonstration:

Layer 1 computes:
$$z_1 = W_1 x + b_1 = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix} = \begin{pmatrix} x_1 + x_2 \\ x_1 + x_2 - 1 \end{pmatrix}$$

Layer 1 activations are: $h = \mathrm{ReLU}(z_1) = \begin{pmatrix} \max(0, x_1 + x_2) \\ \max(0, x_1 + x_2 - 1) \end{pmatrix}$

Layer 2 computes: $\hat{y} = W_2 h + b_2 = \begin{pmatrix} 1 & -2 \end{pmatrix} h + 0 = h_1 - 2h_2$

Network outputs for the XOR inputs:

- (0, 0): $h = \begin{pmatrix} \max(0, 0) \\ \max(0, -1) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \hat{y} = 0 - 2(0) = 0$
- (0, 1): $h = \begin{pmatrix} \max(0, 1) \\ \max(0, 0) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \hat{y} = 1 - 2(0) = 1$
- (1, 0): $h = \begin{pmatrix} \max(0, 1) \\ \max(0, 0) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \hat{y} = 1 - 2(0) = 1$
- (1, 1): $h = \begin{pmatrix} \max(0, 2) \\ \max(0, 1) \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \hat{y} = 2 - 2(1) = 0$

The outputs above are identical with the XOR outputs for the given inputs.

3. Consider the same network as above (with ReLU activations for the hidden layer), with an arbitrary differentiable loss function $\ell : \{0, 1\} \times \{0, 1\} \to \mathbb{R}$ which takes as input $\hat{y}$ and y, our prediction and ground truth labels, respectively. Suppose all weights and biases are initialized to zero. Show that a model trained using standard gradient descent will not learn the XOR function given this initialization.

   Answer:

If all weights and biases are initialized to zero, the network will encounter the symmetry problem and not learn the XOR function using standard gradient descent.

With $W_1 = \mathbf{0}$, $b_1 = \mathbf{0}$, $W_2 = \mathbf{0}$, and $b_2 = 0$:

During the forwardpass:

- The hidden layer before activations will be $z_1 = W_1 x + b_1 = \mathbf{0}x + \mathbf{0} = \mathbf{0}$ for any input x.

- The hidden layer activations will be $h = \mathrm{ReLU}(z_1) = \mathrm{ReLU}(\mathbf{0}) = \mathbf{0}$. All hidden units will output zero.
- The output layer before activation will be $z_2 = W_2 h + b_2 = \mathbf{0}\mathbf{0} + 0 = 0$ for any input x.
- The final output $\hat{y}$ will be 0 for any input x.

During backpropagation

- The gradient with respect to W_2 is $\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial z_2} h^\top$. Since $h = \mathbf{0}$, $\frac{\partial L}{\partial W_2} = \mathbf{0}$.
- The gradient with respect to b_2 is $\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial z_2}$. This might be non-zero, but it's insufficient on its own to enable the network to learn XOR.
- The gradient with respect to z_1 is $\frac{\partial L}{\partial z_1} = (W_2^\top \frac{\partial L}{\partial z_2}) \odot \mathrm{ReLU}'(z_1)$. Since $W_2 = \mathbf{0}$, $W_2^\top \frac{\partial L}{\partial z_2} = \mathbf{0}$.
- The gradient with respect to W_1 is $\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial z_1} x^\top$. Since $\frac{\partial L}{\partial z_1} = \mathbf{0}$, $\frac{\partial L}{\partial W_1} = \mathbf{0}$.
- The gradient with respect to b_1 is $\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial z_1}$. Since $\frac{\partial L}{\partial z_1} = \mathbf{0}$, $\frac{\partial L}{\partial b_1} = \mathbf{0}$.

With the exception of b_2 that could be non-zero, all gradients for weights and biases are zero. As such, W_1, b_1, and W_2 will never be updated, and the network will remain stuck outputting 0 and unable to learn the XOR function.

4. Extra Credit: Now let's consider a more general case than the previous question: we have the same network with an arbitrary hidden layer activation function:

$$h = f(W_1 \cdot x + b_1)$$
$$\hat{y} = W_2 \cdot h + b_2$$

Show that if the initial weights are any uniform constant, then gradient descent will not learn the XOR function from this initialization.

Answer:

This points back to the symmetry problem. If all the weights are initialized to the same value, along with their corresponding weights for the layer, the all the elements of the resulting vector that goes on to the subsequent layer will also be identical. Because they are identical, their gradients will also be identical. This will make it impossible for the different nodes in the network to break symmetry and learn different aspects of the input space needed to form a non-linear decision boundary for XOR.

5 Neural Nets and Backpropagation

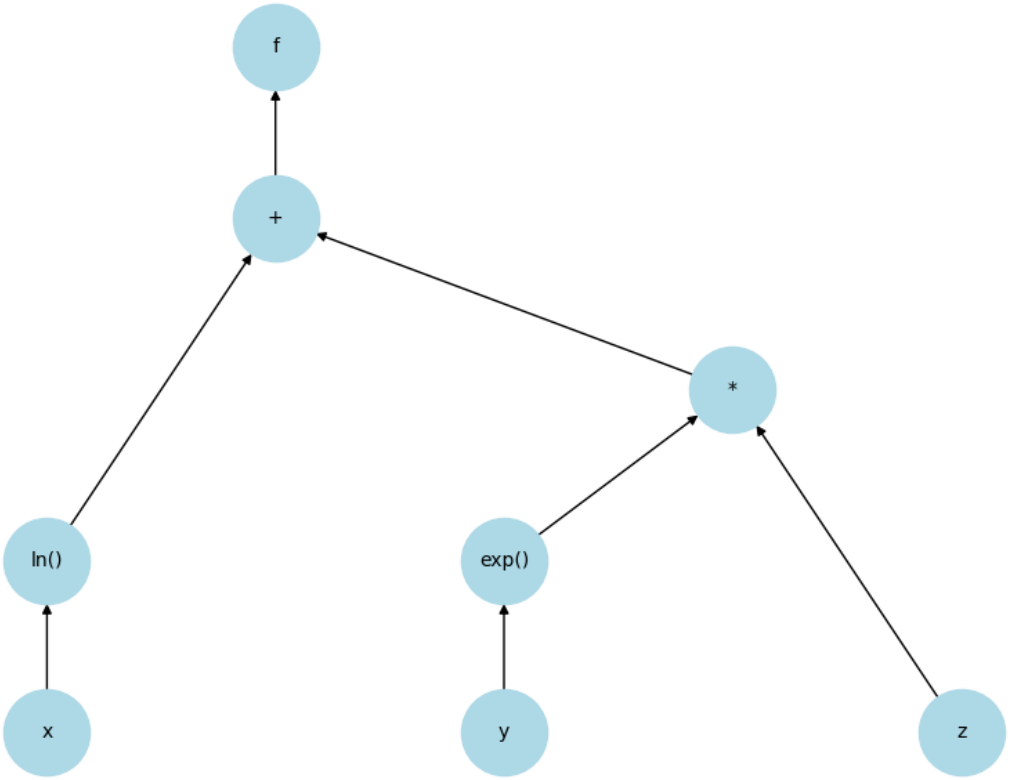
Draw the computation graph for $f(x, y, z) = \ln x + \exp(y) \cdot z$. Each node in the graph should correspond to only one simple operation (addition, multiplication, exponentiation, etc.). Then we will follow the forward and backward propagation described in class to estimate the value of f and partial derivatives $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$ at $[x, y, z] = [1, 3, 2]$. For each step, show your work.

1. Draw the computation graph for $f(x, y, z) = \ln x + \exp(y) \cdot z$. The graph should have three input nodes for x, y, z and one output node f. Label each intermediate node h_i.

   Answer:

The operations on the input nodes are as follows:

- **Input nodes:** x, y, z
- **Operation 1:** $\ln x$
  - Node: $\ln(\cdot)$
  - Input: x
  - Output: $h_1 = \ln x$
- **Operation 2:** $\exp(y)$
  - Node: $\exp(\cdot)$
  - Input: y
  - Output: $h_2 = \exp(y)$
- **Operation 3:** $\exp(y) \cdot z$
  - Node: $\cdot$ (multiplication)
  - Inputs: $h_2 = \exp(y)$ and z
  - Output: $h_3 = h_2 \cdot z = \exp(y) \cdot z$
- **Operation 4:** $\ln x + (\exp(y) \cdot z)$
  - Node: $+$ (addition)
  - Inputs: $h_1 = \ln x$ and $h_3 = \exp(y) \cdot z$
  - Output: $f = h_1 + h_3 = \ln x + \exp(y) \cdot z$



2. Run the forward propagation and evaluate f and h_i ($i = 1, 2, \ldots$) at $[x, y, z] = [1, 3, 2]$.

   Answer:

   For inputs: $x = 1$, $y = 3$, and $z = 2$

   $h_1 = \ln x$
   $h_1 = \ln(1) = 0$

   $h_2 = \exp(y)$
   $h_2 = \exp(3) \approx 20.0855$

   $h_3 = h_2 \cdot z$
   $h_3 = h_2 \cdot 2 \approx 20.0855 \cdot 2 \approx 40.171$

   $f = h_1 + h_3$
   $f = h_1 + h_3 \approx 0 + 40.171 \approx 40.171$

3. Run the backward propagation and give partial derivatives for each intermediate operation, i.e., $\frac{\partial h_i}{\partial x}$, $\frac{\partial h_i}{\partial y}$, and $\frac{\partial f}{\partial h_i}$. Evaluate the partial derivatives at $[x, y, z] = [1, 3, 2]$.

Local Gradients (Partial Derivatives of Operations with respect to inputs):

- Addition node ($f = h_1 + h_3$):
  - $\frac{\partial f}{\partial h_1} = 1$

    - $\frac{\partial f}{\partial h_3} = 1$
- Multiplication node ($h_3 = h_2 \cdot z$):
    - $\frac{\partial h_3}{\partial h_2} = z = 2$
    - $\frac{\partial h_3}{\partial z} = h_2 \approx 20.0855$
- Ln node ($h_1 = \ln x$):
    - $\frac{\partial h_1}{\partial x} = \frac{1}{x} = 1$
- Exp node ($h_2 = \exp(y)$):
    - $\frac{\partial h_2}{\partial y} = \exp(y) \approx 20.0855$

Upstream Gradients (With respect to intermediate nodes):

- $\frac{\partial f}{\partial h_3} = 1$
- $\frac{\partial f}{\partial h_2} = \frac{\partial f}{\partial h_3} \cdot \frac{\partial h_3}{\partial h_2} = 1 \cdot z = 1 \cdot 2 = 2$
- $\frac{\partial f}{\partial h_1} = \frac{\partial f}{\partial h_1} = 1$

4. Aggregate the results in (c) and evaluate the partial derivatives $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$ with the chain rule. Show your work.

    Answer:

Based on values from question 3 at $[x, y, z] = [1, 3, 2]$:

- $\frac{\partial f}{\partial h_1} = 1$
- $\frac{\partial f}{\partial h_2} = 2$
- $\frac{\partial f}{\partial h_3} = 1$
- $\frac{\partial h_1}{\partial x} = 1$
- $\frac{\partial h_2}{\partial y} \approx 20.0855$
- $\frac{\partial h_3}{\partial h_2} = 2$
- $\frac{\partial h_3}{\partial z} \approx 20.0855$

Gradient with respect to x ($\frac{\partial f}{\partial x}$): Path: $x \to h_1 \to f$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial h_1} \cdot \frac{\partial h_1}{\partial x} = 1 \cdot 1 = 1$$

Gradient with respect to y ($\frac{\partial f}{\partial y}$): Path: $y \to h_2 \to h_3 \to f$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial h_3} \cdot \frac{\partial h_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial y} = 1 \cdot z \cdot \exp(y) = 1 \cdot 2 \cdot \exp(3) \approx 40.171$$

Gradient with respect to z ($\frac{\partial f}{\partial z}$): Path: $z \to h_3 \to f$

$$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial h_3} \cdot \frac{\partial h_3}{\partial z} = 1 \cdot h_2 = 1 \cdot \exp(3) \approx 20.0855$$

Final Gradients evaluated at $[1, 3, 2]$: $\frac{\partial f}{\partial x} = 1$, $\frac{\partial f}{\partial y} \approx 40.171$, $\frac{\partial f}{\partial z} \approx 20.0855$