Session 8

Question 1 of 3

Basic Prep (2): Math exercise: Sigmoid example

So far, you've asked LLMs questions to prepare for class. Now it's time for you to be in the hot seat. Give the following prompt to your favorite LLM:

I'm an undergrad taking introductory machine learning. I have some background in statistics, linear algebra, and calculus. I'd like you to act as my professor and write a quiz to test me on gradient methods in neural networks. Use the sigmoid example found at this page: https://cs231n.github.io/optimization-2/ as the basis for the quiz. Please format your response as short-answer quiz with 3 open-ended questions about gradient descent and back-propagation. At least one of these questions should specifically ask about the relationship between Jacobians and neural network gradients.

For each question the LLM creates, answer to the best of your ability using the CS231n reading. Copy your answers into the chat with the LLM one at a time. Add the following prompt after each one

"Here are my responses to question #X. Can you give me feedback, including a model answer for what you think would be a good response to your prompt?"

Repeat for all of your responses and ask the LLM some follow-ups. Some things I've noticed interviewing these systems is that their easily captured by high-level discussions of vanishing gradients without actually defining simpler terms like what a gradient actually is. Don't let them get away with it! Ask pointed follow ups. Here's a good example:

In a neural network we often compute the gradient on the activation of a single unit, and the Jacobian with respect to an entire layer of weights. Why?

Machine Learning Quiz: Gradient Methods in Neural Networks

1. Understanding Gradient Descent

Question:
Explain the key steps of the gradient descent algorithm when optimizing a neural network with a sigmoid activation function. How does the learning rate influence the convergence of the model?

My answer:
The first step of the gradient descent algorithm is initializing parameters, randomly setting weights and biases. Then, it's forward propagation, where you compute network output using the sigmoid function. Then you need to calculate the loss using MSE. Next is backward

propagation, where you compute gradients using the chain rule and sigmoid derivative $\sigma'(z)=\sigma(z)(1-\sigma(z))$. Update the weights and biases using the gradients computed in the previous step using the update rule $\theta=\theta-\eta\cdot\nabla L(\theta)$, where η is the learning rate and $\nabla L(\theta)$ is the gradient of the loss with respect to θ. Lastly, you need to repeat steps 2-5 for several iterations (epochs) or until the loss converges.

When the learning rate is too small, convergence is slow because it would require the model to take very small steps toward the optimal solution. On the other hand, when the learning rate is too large, the model might overshoot the optimal solution, leading to large fluctuations in the loss function, which could potentially cause the algorithm to diverge. Thus, we need to find an optimal learning rate that allows the model to converge quickly and stably.

Feedback:
You provided a detailed explanation of the steps involved, from initialization to forward propagation, loss calculation, and backpropagation. The description of how the learning rate affects convergence is on point. However, it could be slightly more concise and focused.

Model answer:
The gradient descent algorithm starts by initializing the weights and biases randomly. In each iteration, forward propagation computes the network's output using the sigmoid function, and the loss (mean squared error) is calculated. During backpropagation, gradients of the loss with respect to the weights are calculated using the chain rule and the derivative of the sigmoid function $\sigma'(z)=\sigma(z)(1-\sigma(z))$. These gradients are used to update the parameters using the rule $\theta=\theta-\eta\nabla L(\theta)$, where η is the learning rate. A small η leads to slow convergence, while a large η risks overshooting the minimum. An optimal learning rate ensures stable, fast convergence.

---

2. Backpropagation and Chain Rule

Question:
In the context of a neural network using a sigmoid activation function, describe how backpropagation computes gradients. Specifically, how does the chain rule apply when calculating gradients for the weights in the network?

My answer:
Backpropagation calculates gradients by propagating the error from the output layer backward by updating the weights and biases. To calculate gradients for the weights in the network, we start with the forward pass, where the output for each neuron is computed as $a=\sigma(z)$, $z=w\cdot x+b$, where w is the weight, x is the input, b is the bias, and $\sigma(z)$ is the sigmoid activation. Then, the loss function L compares the predicted output with the true label. To compute the gradient of the loss L with respect to a weight w, we use the chain rule to break the process into steps.

$$\frac{\partial L}{\partial w}=\frac{\partial L}{\partial a}\cdot\frac{\partial a}{\partial z}\cdot\frac{\partial z}{\partial w}$$

where $\frac{\partial L}{\partial a}$ is the gradient of the loss with respect to the neuron's output, $\frac{\partial a}{\partial z}$ is the derivative of the sigmoid activation function, and $\frac{\partial z}{\partial w}$ is the derivative of the weighted sum with respect to the weight. The chain rule allows the model to propagate these gradients backward efficiently as error is passed back through the layers, where each weight update in a layer depends on the gradients of the next layer.

Feedback:
Your explanation is clear and detailed, providing a solid step-by-step breakdown of backpropagation. You correctly described the forward pass and how the chain rule is applied. It's thorough, but simplifying the step-by-step flow slightly could make it even clearer and easier to digest.

Model answer:
Backpropagation calculates the gradients by applying the chain rule to propagate the error backward through the network. For each neuron, the output is computed as $a=\sigma(z)$ where $z=w\cdot x+b$. The loss function compares the predicted output with the true label. To compute $\frac{\partial L}{\partial w}$, we apply the chain rule:

$$\frac{\partial L}{\partial w}=\frac{\partial L}{\partial a}\cdot\frac{\partial a}{\partial z}\cdot\frac{\partial z}{\partial w}$$

where $\frac{\partial a}{\partial z}$ is the derivative of the sigmoid function. Backpropagation efficiently calculates these gradients layer by layer, allowing weight updates that minimize the loss function. The key idea is that the gradient of each layer depends on the gradients from subsequent layers.

---

3. Jacobian and Neural Network Gradients

Question:
Explain the role of the Jacobian matrix in computing gradients for neural networks. How does the Jacobian relate to the partial derivatives of the loss function with respect to the weights during backpropagation?

My answer:
The Jacobian matrix plays a crucial role in computing gradients in neural networks, especially when dealing with vector-valued functions like the output of multiple neurons. In a neural network, the Jacobian matrix is the partial derivatives of each output neuron with respect to each input neuron. In backpropagation, we compute the gradient of the loss function L with respect to the weights w. The Jacobian matrix helps represent the transformation of gradients as they propagate through layers of the network. When the chain rule is applied to compute the partial derivative of the loss L with respect to the weights w of layer l as

$$\frac{\partial L}{\partial w^{(l)}}=\frac{\partial L}{\partial a^{(l)}}\cdot\frac{\partial a^{(l)}}{\partial w^{(l)}},$$

the term $\frac{\partial a^{(l)}}{\partial a^{(l-1)}}$ forms the Jacobian matrix, which represents the relationship between the outputs of the current layer and the inputs from the previous layer. That is, the matrix multiplies the gradient coming from the subsequent layers to compute how the parameters at the current layer should be adjusted.

Feedback:
This is your strongest answer—thorough and correct. You clearly articulate the role of the Jacobian in backpropagation and how it transforms gradients across layers. The only minor suggestion would be slightly more brevity in some parts, but the explanation is excellent overall.

Model answer:
In neural networks, the Jacobian matrix represents the partial derivatives of the output with respect to the input. In backpropagation, the Jacobian helps compute the transformation of gradients between layers. Specifically, when we apply the chain rule to compute the gradient $\frac{\partial L}{\partial w^{(l)}}$, the term $\frac{\partial a^{(l)}}{\partial a^{(l-1)}}$ forms the Jacobian matrix. This matrix efficiently propagates gradients from one layer to the next, ensuring that the weight updates reflect the influence of all layers in the network. The Jacobian plays a key role in scaling gradients across multiple dimensions, especially in deeper networks.

In a neural network we often compute the gradient on the activation of a single unit, and the Jacobian with respect to an entire layer of weights. Why?

In a neural network, we compute the gradient of the activation of a single unit because we want to adjust the weight connected to that particular unit based on how much it contributes to the error in the output. The gradient tells us the direction and magnitude of the change needed for that specific unit to minimize the loss function. This fine-grained update ensures that each unit's contribution is optimized individually.

On the other hand, we compute the Jacobian with respect to an entire layer of weights because the Jacobian matrix encapsulates how the output of an entire layer is affected by changes in the input or the weights. The Jacobian represents the partial derivatives of all output units with respect to all input units (or weights). This is critical in backpropagation, where we need to efficiently propagate the gradients across layers, and the Jacobian helps aggregate and propagate the influence of an entire layer on subsequent layers. This layer-wide representation allows for efficient matrix operations, speeding up the computation of gradients for deep networks, especially when multiple neurons and layers are involved.

Q2A: Coding exercise: Sigmoid example (3)

Exercise 3.1: Work out (on paper, or media of your choice) all the derivatives (the red values) in the example. Start at the rightmost value (the 1.00 in red). Your computed derivatives should match the ones in the diagram.

Here's a few hints: What's for any ? (a.k.a. why is the right most value 1.00?) Remember backpropagation is a strictly local process: to compute a gradient you only need immediate parents/children node values. If you remember the material from CS111, recall each step is just the chain rule! For some (most) of the calculations, you'll need some combination of both nearby derivatives (red) and original (green) values. If you're really stuck, there's one completed value (in code form) in the next cell.

Of course, in real life computers do the forward/backward passes for us! Here's an implementation of the forward pass, and an incomplete implementation of the backwards pass.

Backpropagation Explanation

Forward Pass:

1. **Multiplication Nodes (left side of the diagram)**:
   - $w_0 \cdot x_0 = 2.00 \cdot (-1.00) = -2.00$
   - $w_1 \cdot x_1 = -3.00 \cdot (-2.00) = 6.00$
2. **Addition of the two multiplication results**:
   - $-2.00 + 6.00 = 4.00$
3. **Adding w_2**:
   - $4.00 + (-3.00) = 1.00$
4. **Negation**:
   - -1.00
5. **Exponentiation**:
   - $\exp(-1.00) = 0.37$
6. **Addition of 1**:
   - $0.37 + 1 = 1.37$
7. **Inversion**:
   - $\frac{1}{1.37} = 0.73$

Backward Pass (Derivatives):

1. **Starting from the final output (Inversion)**:
   - The derivative with respect to the output is $\frac{\partial \text{invert}}{\partial \text{invert}} = 1.00$ (as given in red on the rightmost side).
2. **Derivative with respect to `plus1`**:
   - $d_{\text{plus1}} = d_{\text{invert}} \cdot \left(-\frac{1}{\text{plus1}^2}\right)$
   - $d_{\text{plus1}} = 1.00 \cdot \left(-\frac{1}{1.37^2}\right) = -0.53$ (matching the red value).
3. **Derivative with respect to `exp`**:
   - $d_{\text{exp}} = d_{\text{plus1}} \cdot 1 = -0.53$
4. **Derivative with respect to `neg`**:
   - $d_{\text{neg}} = d_{\text{exp}} \cdot \exp(\text{neg})$
   - $d_{\text{neg}} = -0.53 \cdot 0.37 = -0.20$ (matching the red value).
5. **Derivative with respect to `sum012` (negation)**:

- $d_{\text{sum012}} = d_{\text{neg}} \cdot (-1) = 0.20$ (matching the red value).

6. **Derivative with respect to w_2 and `sum01` (addition):**
   - $d_{w_2} = d_{\text{sum012}} = 0.20$
   - $d_{\text{sum01}} = d_{\text{sum012}} = 0.20$
7. **Derivative with respect to `mul0` and `mul1` (addition):**
   - $d_{\text{mul0}} = d_{\text{sum01}} \cdot 1 = 0.20$
   - $d_{\text{mul1}} = d_{\text{sum01}} \cdot 1 = 0.20$
8. **Derivative with respect to w_0, x_0, w_1, and x_1 (multiplication):**
   - $d_{w_0} = d_{\text{mul0}} \cdot x_0 = 0.20 \cdot (-1.00) = -0.20$
   - $d_{x_0} = d_{\text{mul0}} \cdot w_0 = 0.20 \cdot 2.00 = 0.39$
   - $d_{w_1} = d_{\text{mul1}} \cdot x_1 = 0.20 \cdot (-2.00) = -0.39$
   - $d_{x_1} = d_{\text{mul1}} \cdot w_1 = 0.20 \cdot (-3.00) = -0.59$

Summary of Derivatives:

- $d_{\text{invert}} = 1.00$
- $d_{\text{plus1}} = -0.53$
- $d_{\text{exp}} = -0.53$
- $d_{\text{neg}} = -0.20$
- $d_{\text{sum012}} = 0.20$
- $d_{w_2} = 0.20$
- $d_{\text{sum01}} = 0.20$
- $d_{\text{mul0}} = 0.20$
- $d_{\text{mul1}} = 0.20$
- $d_{w_0} = -0.20$
- $d_{x_0} = 0.39$
- $d_{w_1} = -0.39$
- $d_{x_1} = -0.59$

```python
import numpy as np

# Forward pass
w0, w1, w2 = 2.00, -3.00, -3.00
x0, x1 = -1.00, -2.00

mul0 = w0 * x0  # -2.00
mul1 =  w1 * x1  # 6.00
sum01 = mul0 + mul1  # 4.00
sum012 = sum01 + w2  # 1.00

neg = -sum012  # -1.00
```

```
exp = np.exp(neg)  # 0.37
plus1 = exp + 1  # 1.37
invert = 1/plus1  # 0.73


# Backward pass
# Remember *all* derivatives are with respect to the final function
invert
d_invert = 1.00  # this is d_invertd_invert, but we shorten to
d_invert
# Here's one completed for you :)
d_plus1 = d_invert * (-1/plus1**2)  # this is d_invertd_plus_1, but we
shorten to d_plus_1
d_exp = d_plus1 * 1
d_neg = d_exp * np.exp(neg)
d_sum012 = d_neg * (-1)

# Notice for gates with multple inputs, when going backward
# they are responsible for derivatives of multiple values!
d_w2, d_sum01 = d_sum012, d_sum012
d_mul0, d_mul1 = d_sum01, d_sum01

d_w1, d_x1 = d_mul1 * x1, d_mul1 * w1
d_w0, d_x0 = d_mul0 * x0, d_mul0 * w0

print("Backward Pass Results:")
print(f"d_w0: {d_w0}")
print(f"d_x0: {d_x0}")
print(f"d_w1: {d_w1}")
print(f"d_x1: {d_x1}")
print(f"d_w2: {d_w2}")
```

```
Backward Pass Results:
d_w0: -0.19661193324148188
d_x0: 0.39322386648296376
d_w1: -0.39322386648296376
d_x1: -0.5898357997244457
d_w2: 0.19661193324148188
```

Question 2 of 3

Q2B: Coding exercise: Calculus derivation (3)

Problem Statement:

Given the following vectors and function:

$$w=\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}, x=\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, f(w,x)=w^T x=w_1 x_1+w_2 x_2+w_3 x_3$$

To Compute $ \frac{df}{dw} $:

Since the function $ f(w, x) $ is a linear combination of the elements of $ w $ and $ x $, we can calculate the derivative of $ f $ with respect to each component of $ w $.

1. The partial derivative of $ f $ with respect to $ w_1 $ is: [\frac{\partial f}{\partial w_1} = x_1]
2. The partial derivative of $ f $ with respect to $ w_2 $ is: [\frac{\partial f}{\partial w_2} = x_2]
3. The partial derivative of $ f $ with respect to $ w_3 $ is: [\frac{\partial f}{\partial w_3} = x_3]

Thus, the gradient of $ f(w, x) $ with respect to $ w $ is:

$ \frac{df}{dw} = \begin{bmatrix} \frac{\partial f}{\partial w_1} \ \frac{\partial f}{\partial w_2} \ \ frac{\partial f}{\partial w_3} \end{bmatrix} = \begin{bmatrix} x_1 \ x_2 \ x_3 \end{bmatrix} = x $

This result shows that the derivative of the function $ f $ with respect to $ w $ is simply the vector $ x $.

Core Questions (3): Coding exercise: Sigmoid example (vectorized)

Building towards the real world: real world inputs can be tens, even thousands of dimensions (even MNIST is dimensions). We can't just create a variable for each of these dimensions, so we store them as vectors (and you'll see in class, matrices), which are easier to work with once you get the hang of them. Again, the underlying math does not change, only our implementation does.

Exercise: Complete the code for the backwards pass; however this time and are vectors.

Hints: Note that we've fused the multiply and add operations () into the matrix multiply . You derived the gradients for both and above, so use it when you're coding now!

```
import numpy as np

# Forward pass
w = np.array([2.00, -3.00])
b = -3.00
x = np.array([-1.00, -2.00])

dot = w.T @ x  # 4.00
summed = dot + b  # 1.00
```

```
neg = -summed  # -1.00
exp = np.exp(neg)  # 0.37
plus1 = exp + 1  # 1.37
invert = 1/plus1  # 0.73

# Backward pass
d_invert = 1.00
d_plus1 = d_invert * (-1 / plus1**2)  # -0.53
d_exp = d_plus1 * 1  # -0.53
d_neg = d_exp * np.exp(neg)  # -0.20
d_summed = d_neg * (-1)  # 0.20
d_b = d_summed  # 0.20
d_dot = d_summed  # 0.20

# Be careful, the next two are vectors!
d_w = d_dot * x  # [-0.20, -0.40]
d_x = d_dot * w  # [0.40, -0.60]

# Print results
print("d_b:", d_b)
print("d_w:", d_w)
print("d_x:", d_x)
```

```
d_b: 0.19661193324148188
d_w: [-0.19661193 -0.39322387]
d_x: [ 0.39322387 -0.5898358 ]
```

Preview: Auto-magic gradients Even though we're no longer computing by hand, that was quite tedious! You had to write double the code: once for the actual computation (forward pass) then once more for the gradients (backward pass), and this is how researchers did this during the early 2010's. We've come quite a long way, with cool new libraries that does this for us. Here's a snippet introducing JAX, which we'll be using in class:

```
%%capture
%pip install jax[cpu]

import jax
import jax.numpy as jnp  # JAX has a version of NumPy that acts...just like NumPy!

# We first define a function for *just* the forward pass
def sigmoid_example(params, x):
    dot = params['w'].T @ x  # 4.00
    summed = dot + params['b']  # 1.00

    neg = -summed  # -1.00
    exp = jnp.exp(neg)  # 0.37
    plus1 = exp + 1  # 1.37
    invert = 1/plus1  # 0.73
```

```
    return invert

params = {'w': np.array([2.00, -3.00]), 'b': -3.00}  # We define our parameters as a dict.
x = np.array([-1.00, -2.00])

sigmoid_example(params, x)  # We get the expected value of 0.73 back.
```

```
Array(0.7310586, dtype=float32)
```

```
grad_sigmoid_example = jax.grad(sigmoid_example)
grad_sigmoid_example(params, x)
```

```
{'b': Array(0.19661197, dtype=float32, weak_type=True),
 'w': Array([-0.19661197, -0.39322394], dtype=float32)}
```

Extensions (4): Coding exercise - training (4)

Exercise: Extend the code snipped from Q2C, by creating a function that runs gradient descent on the weights for 1000 iterations (that is, find a value of such that the output of the sigmoid example is close to 0). This means: Instead of computing the gradient once, you'll need to do it 1000 times (so you'll probably want the gradient computation as a function of the weights). You'll need to update your weights after every iteration, with a step size of your choosing. You can either use the manually written version of the gradient computation, or the ones computed by JAX. Feel free to read the docs if you do the latter!

```
# Best wishes!
import numpy as np

# Sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Function to compute forward pass and loss
def forward_pass(w, b, x):
    dot = w.T @ x
    summed = dot + b
    neg = -summed
    exp = np.exp(neg)
    plus1 = exp + 1
    invert = 1 / plus1
    return invert

# Function to compute gradients
def compute_gradients(w, b, x):
    dot = w.T @ x
    summed = dot + b
    neg = -summed
    exp = np.exp(neg)
```

```python
    plus1 = exp + 1
    invert = 1 / plus1

    # Backward pass
    d_invert = 1.00
    d_plus1 = d_invert * (-1 / plus1**2)
    d_exp = d_plus1 * 1
    d_neg = d_exp * np.exp(neg)
    d_summed = d_neg * (-1)

    d_b = d_summed
    d_dot = d_summed

    # Derivatives with respect to w and x
    d_w = d_dot * x
    return d_w, d_b

# Gradient descent function
def gradient_descent(w, b, x, learning_rate=0.01, iterations=1000):
    for i in range(iterations):
        # Compute gradients
        d_w, d_b = compute_gradients(w, b, x)

        # Update weights and bias
        w = w - learning_rate * d_w
        b = b - learning_rate * d_b

        # Every 100 iterations, print the current output
        if i % 100 == 0:
            output = forward_pass(w, b, x)
            print(f"Iteration {i}, Output: {output}")

    return w, b

# Initialize weights, bias, and input
w = np.array([2.00, -3.00])  # weights vector
b = -3.00                    # bias
x = np.array([-1.00, -2.00]) # input vector

# Run gradient descent
learning_rate = 0.01
iterations = 1000
final_w, final_b = gradient_descent(w, b, x, learning_rate,
iterations)

# Final output
final_output = forward_pass(final_w, final_b, x)
print(f"Final weights: {final_w}, Final bias: {final_b}")
print(f"Final output: {final_output}")
```