CSE 493G/599G: Deep Learning

Section 2: Backpropagation

Welcome to section, we're glad you could make it!

0. Reference Material

Intuition for Backprop

Recall some basic facts:

1) The loss function L measures how "bad" our current model is.
2) L is a function of our parameters W.
3) We want to minimize L.

Thus, we update W to minimize L using $\frac{\partial L}{\partial W}$.

For example, if $\frac{\partial L}{\partial W_1}$ was positive, increasing W_1 would increase L. Accordingly, we'd choose to decrease W_1.

More generally, `weights += (-1 * step_size * gradient)`.

Unfortunately, taking the derivative $\frac{\partial L}{\partial W}$ can get extremely difficult, especially at the scale of state-of-the-art models. For instance, GLM-4.5 has 92 hidden layers and 32 billion parameters. Imagine taking 32 billion derivatives, with each derivative having hundreds of applications of chain rule.

Instead, we employ a technique known as **backprop**.

First, we split our function into multiple equations until there is *one operation per equation*. This process is known as **staged computation**. Next, we take the derivatives of each of these smaller equations, before finally linking them together using **chain rule**.

Common Gates

Feel free to take notes on the common backprop gates here.

1. Compute and Conquer

For each function below, use the staged computation approach to split it into smaller equations.

(a) $f(x, y, z) = (x + y)z$

(b) $h(x, y, z) = (x^2 + 2y)z^3$

(c) $g(x, y, z) = \big(\ln(x) + \sin(y)\big)^2 + 4x$

2. Oh, node way!

For each function below:

(i) construct a computational graph

(ii) do a forward and backward pass through the graph using the provided input values

(iii) complete the Python function for a combined forward and backward pass

It may be useful to consider how you split these functions into smaller equations in the question above.

(a) $f(x, y, z) = (x + y)z$ with input values $x = 1, y = 3, z = 2$

```
import numpy as np

# inputs: NumPy arrays `x`, `y`, `z` of identical size
# outputs: forward pass in `out`, gradients for x, y, z in `fx`, `fy`, `fz` respectively
def q2a(x, y, z):














    return out, fx, fy, fz
```

Ignore the line numbers, they do NOT correspond to the number of lines you need to write.

(b) $h(x, y, z) = (x^2 + 2y)z^3$ with input values $x = 3, y = 1, z = 2$

```python
import numpy as np

# inputs: NumPy arrays `x`, `y`, `z` of identical size
# outputs: forward pass in `out`, gradients for x, y, z in `hx`, `hy`, `hz` respectively
def q2b(x, y, z):























    return out, hx, hy, hz
```

Ignore the line numbers, they do NOT correspond to the number of lines you need to write.

(c) $g(x, y, z) = \big(\ln(x) + \sin(y)\big)^2 + 4x$ with input values $x = e, y = \frac{\pi}{2}, z = 2$

Python function printed on the following page.

```python
import numpy as np

# inputs: NumPy arrays `x`, `y`, `z` of identical size
# outputs: forward pass in `out`, gradients for x, y, z in `gx`, `gy`, `gz` respectively
def q2c(x, y, z):












































    return out, gx, gy, gz
```

Ignore the line numbers, they do NOT correspond to the number of lines you need to write.

3. Sigmoid Shenanigans

Consider the Sigmoid activation function:

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

Draw a computational graph and work through the backpropagation. Then, fill in the Python function. If you finish early, work through the analytical derivative for Sigmoid.

As a hint, you could split Sigmoid into the following functions:

$$a(x) = -x \qquad b(x) = e^x \qquad c(x) = 1 + x \qquad d(x) = \frac{1}{x}$$

Observe that chaining these operations gives us Sigmoid: $d(c(b(a(x)))) = \sigma(x)$.

Suppose $x = 2$. What would the gradient with respect to x be? Feel free to use a calculator on this part.

You should have gotten around 0.1. If the step size is 0.2, what would the value of x be after taking one gradient descent step? As a hint, remember that `parameters -= step_size * gradient`.

```python
import numpy as np

# inputs:
#   - a NumPy array `x`
# outputs:
#   - `out`: the result of the forward pass
#   - `fx` : the result of the backwards pass
def sigmoid(x):
    # provided: forward pass with cache
    a = -x
    b = np.exp(a)
    c = 1 + b
    d = c ** -1
    out = d

    # TODO: backwards pass, "fx" represents df / dx

return out, fx
```

Ignore the line numbers, they do NOT correspond to the number of lines you need to write.

4. A Backprop a Day Keeps the Derivative Away

Consider the following function:

$$f = \frac{\ln x \cdot \sigma\left(\sqrt{y}\right)}{\sigma\left((x+y)^2\right)}$$

Break the function up into smaller parts, then draw a computational graph and finish the Python function.

For reference, the derivative of Sigmoid is $\sigma(x) \cdot \left(1 - \sigma(x)\right)$.

The TA solution breaks the function into 8 additional equations and rewrites f in terms of 2 of those additional equations. Yours doesn't have to match this exactly.

Python function printed on the following page.

```python
import numpy as np

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

# inputs: NumPy arrays `x`, `y`
# outputs: forward pass in `out`, gradient for x in `fx`, gradient for y in `fy`
def complex_layer(x, y):

    # forward pass

    # backwards pass

    return out, fx, fy
```

Ignore the line numbers, they do NOT correspond to the number of lines you need to write.