CSE 493G1/599G1: Deep Learning

Solutions for Section 2: Project Design & Backpropagation

Thanks for attending section, we hope you found it helpful.

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, LLaMA 2-70B has 80 transformer layers and 70 billion parameters. Imagine taking 70 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**.

1. Compute and Conquer

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

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

Solution:

Decompose the function as follows:

- $a = x + y$
- $b = z$
- $f = ab$

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

Solution:

Decompose the function as follows:

- $a = x^2$
- $b = 2y$
- $c = a + b$
- $d = z^3$
- $h = cd$

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

Solution:

Decompose the function as follows:

- $a = \ln(x)$
- $b = \sin(y)$
- $c = a + b$
- $d = c^2$
- $f = 4x$
- $g = d + f$

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

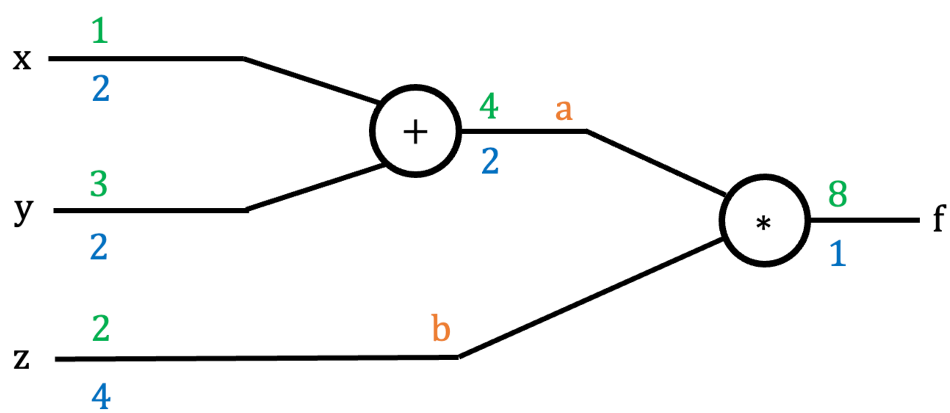
(iii) complete the Python function for a combined forward and backward pass

Hint: 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 $x = 1$, $y = 3$, $z = 2$

Solution:

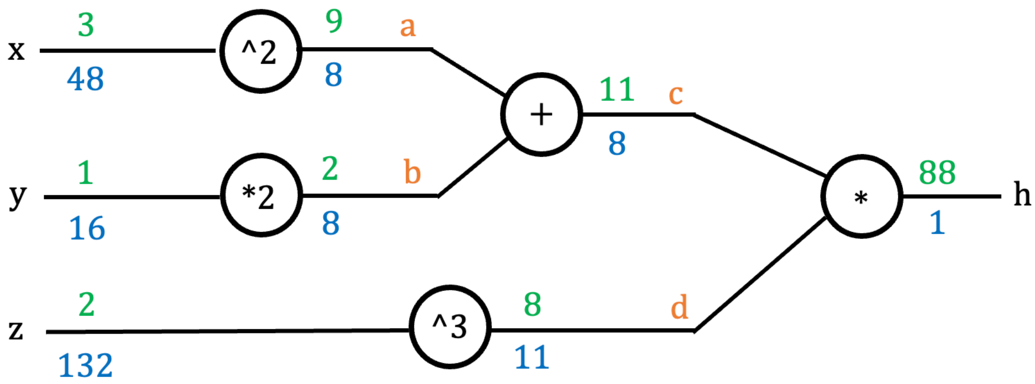
Forward pass values are displayed in green; backward pass values are displayed in blue. The orange letters correspond to the mini-equations from Question 1.




```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 `fx`, `fy`, `fz` respectively
def q2a(x, y, z):
    # forward pass
    a = x + y
    b = z
    f = a * b
    out = f

    # backward pass
    ff = 1
    fb = ff * a
    fa = ff * b
    fz = fb * 1
    fx = fa
    fy = fa

    return out, fx, fy, fz
```

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

Solution:

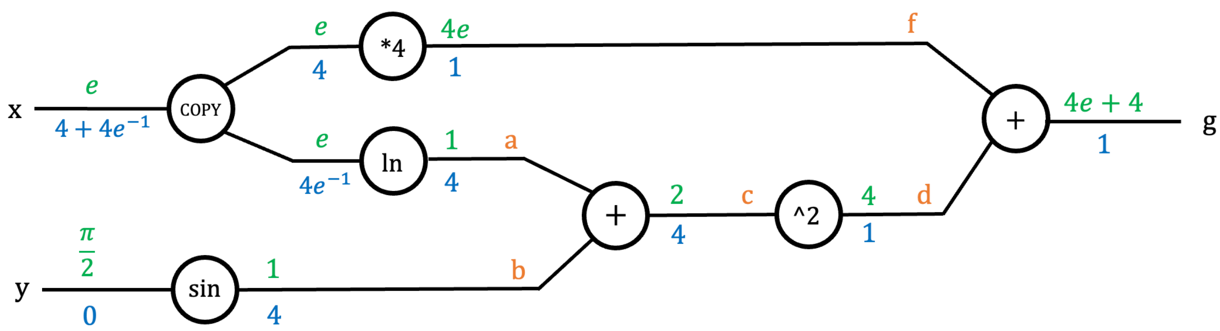



```python
import numpy as np

# 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):
    # forward pass
    a = x ** 2
    b = 2 * y
    c = a + b
    d = z ** 3
    h = c *d
    out = h

    # backward pass -- right-most gate
    hh = 1
    hc = hh * d
    hd = hh * c

    # backward pass -- top branches
    ha = hc
    hb = hc
    hx = ha * (2 * x)
    hy = hb * 2

    # backward pass -- bottom branch
    hz = hd * (3 * (z ** 2))

    return out, hx, hy, hz
```

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

Solution:

We omit z in the computational graph below since it does not appear in the formula for g. It is important to realize that the gradient with respect to z is 0.



A few observations:

- We have a gradient (4) flowing back to y, but it dies on the last gate since $\frac{d}{dy}(\sin(y)) = \cos(x)$ and $\cos(\frac{\pi}{2}) = 0$. This is problematic since it means we don't change y on this gradient descent step despite having feedback suggesting that y should be decremented.
- Since $\ln(x) = \frac{1}{x}$, the local gradient associated with equation a can be undefined if $x = 0$. If you were asked to implement this function and its backwards pass in Python, what are some potential workarounds you might employ?

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):
    # forward pass
    a = np.log(x)
    b = np.sin(y)
    c = a + b
    d = c ** 2
    f = 4 * x
    g = d + f
    out = g

    # backward pass -- right-most gate
    gg = 1
    gf = gg
    gd = gd

    # backward pass -- path via `d`
    gc = gd * (2 *c)
    ga = gc
    gb = gc
    gx_1 = ga * (x ** -1)
    gy = gb * np.cos(y)

    # backward pass -- path via `f`
    gx_2 = gf * 4

    # backward pass -- reconciliation at copy gate
    gx = gx_1 + gx_2

    # z never appears in the function, so it has no gradient
    gz = 0

    return out, gx, gy, gz
```

3. Sigmoid Shenanigans

Consider the Sigmoid activation function:

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

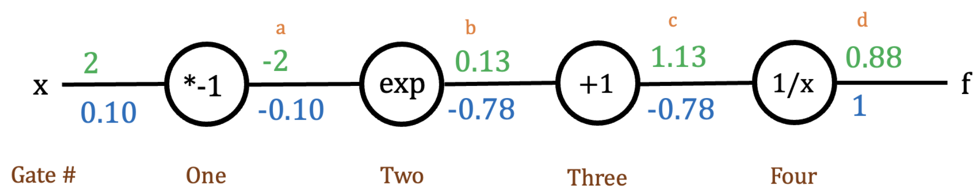
(a) Draw a computational graph and work through the backpropagation. Then, fill in the Python function. If you finish early, work through the analytical derivation for Sigmoid.

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: $d(c(b(a(x)))) = \sigma(x)$.

Solution:



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

Solution:

Recall that downstream = upstream $\times$ local.

At Gate Four, the upstream gradient is 1 and the local gradient is $\frac{\partial}{\partial c}\left(\frac{1}{c}\right) = -\frac{1}{c^2} = -\frac{1}{(1.13)^2} = -0.78$. Thus, the downstream gradient is $1 \times -0.78 = -0.78$.

At Gate Three, the upstream is -0.78 and the local is $\frac{\partial}{\partial b}\left(b + 1\right) = 1$. Thus, the downstream is $-0.78 \times 1 = -0.78$.

At Gate Two, the upstream is -0.78 and the local is $\frac{\partial}{\partial a}\left(e^a\right) = e^a = e^{-2} = 0.135$. Thus, the downstream is $-0.78 \times 0.135 = -0.10$.

At Gate One, the upstream is -0.10 and the local is $\frac{\partial}{\partial x}\left(-x\right) = -1$. Thus, the downstream is $-0.10 \times -1 = 0.10$.

Therefore, $\frac{df}{dx} \approx 0.10$. We use $\approx$ here because we rounded decimals throughout our calculations.

(c) 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`.

Solution:

Our parameter, x, started off at 2. Our step size was 0.2 and our gradient is 0.1. Plugging into the equation for gradient descent, the new value for x is $2 - 0.2(0.1) = 2 - 0.02 = 1.98$.

(d) Implement the function below for a full forward and backward pass through Sigmoid.

Solution:

```
import numpy as np

# inputs:
#   - a numpy array `x`
# outputs:
#   - `out`: the result of the forward pass
#   - `fx` : the result of the backward pass
def sigmoid(x):
    # provided: forward pass with cache
    a = -x
    b = np.exp(a)
    c = 1 +b
    f = 1/c
    out = f

    # TODO: backward pass, "fx" represents df / dx
    ff = 1
    fc = ff * -1/(c**2)
    fb = fc * 1
    fa = fb * np.exp(a)
    fx = fa * -1

    return out, fx
```

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 $\sigma(x) \cdot \big(1 - \sigma(x)\big)$.

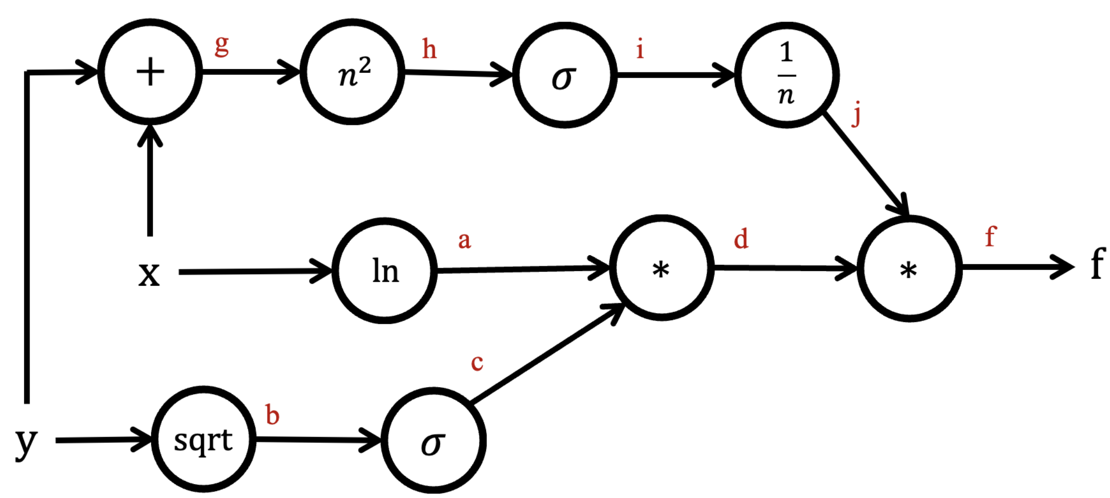
The TA solution breaks the function into 8 additional equations and rewrites f in terms of 2 of those additional equations. Yours doesn't have to match this exactly.

Solution:

We begin by breaking the function down:

Numerator:	$a = \ln x$	$b = \sqrt{y}$	$c = \sigma(b)$	$d = a \cdot c$
Denominator:	$g = x + y$	$h = g^2$	$i = \sigma(h)$	$j = \dfrac{1}{i}$
Final:	$f = dj$			

Although $f = \frac{d}{i}$ is a valid, one-operation gate, we generally try to avoid quotient rule. Therefore, we introduce an extra operation, $i = \frac{1}{j}$, leaving us with $f = di$.



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
    a = np.log(x)
    b = np.sqrt(y)
    c = sigmoid(b)
    d = a * c
    g = x + y
    h = g ** 2
    i = sigmoid(h)
    j = 1 / i
    out = d * j

    # backward pass -- output gate
    ff = 1
    fd = ff * j
    fj = ff * d

    # backward pass -- top branch
    fi = fj * -1 / (i ** 2)
    fh = fi * sigmoid(h) * (1 - sigmoid(h))
    fg = fh * 2 * g
    fx_1 = fg
    fy_1 = fg

    # backward pass -- middle branch
    fa = fd * c
    fx_2 = fa / x

    # backward pass -- bottom branch
    fc = fd * a
    fb = fc * sigmoid(b) * (1 - sigmoid(b))
    fy_2 = fb / (2 * np.sqrt(y))

    # backward pass -- reconciliation
    fx = fx_1 + fx_2
    fy = fy_1 + fy_2

    return out, fx, fy
```