CSE 493G1/599G1: Deep Learning

Section 4: Backpropagation & Convolutions

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, 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**.

Equations for Convolutions

Assuming the following variables, which imply that the input image has size (W, H, C),

- W is the width of the input image
- H is the height of the input image
- C is the number of channels in the input image
- F is the receptive field size (i.e., the height and width of the conv field)
- S is the stride with which the convolution is applied
- P is the padding
- K is the depth of the conv layer (i.e., the number of filters applied)

The output will have size $\left(\frac{W-F+2P}{S}+1,\ \frac{H-F+2P}{S}+1,\ K\right)$.

The conv layer will have $K\left(F^2C+1\right)$ trainable parameters.

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:

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

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.

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.

5. As Convoluting As Possible

(a) What's the formula for determining a conv layer's output size? Assume that the receptive field is a square. Define all the variables you use.

(b) Consider a conv layer that takes a $32 \times 32 \times 3$ input and applies a $5 \times 5 \times 3$ filter with no padding. Compute the output sizes if we use strides of 1, 2, and 3. Then, compute the output sizes if we use strides of 2 and 3 with a padding of 3.

Hint: certain strides may result in an invalid configuration for this conv layer.

Note: People will often leave out the channels dimension when writing out a filter (i.e., they might refer to a $5 \times 5 \times 3$ filter as a 5×5 filter). We will now adopt this shorthand as well.

(c) Consider the first conv layer of AlexNet, which takes an input of size $227 \times 227 \times 3$ and applies 96 separate 11×11 convolutional filters with stride of 4 and no padding. People will sometimes write this as applying one 11×11 filter with depth 96; it means the same thing. What are our output dimensions?

(d) Develop a formula for the number of trainable parameters in a conv layer. Assume that the receptive field is a square and that the conv layer has biases. Define all the variables you use.

(e) Consider the first conv layer of AlexNet mentioned above. How many trainable parameters are there?

(f) Consider a conv layer which takes a $31 \times 31 \times 5$ input and applies a 3×3 filter with depth 25, stride 2, and padding 1. How many trainable parameters does it have?

Takeaway: The values you set K and F to (these are hyperparameters!) will have a significant impact on the number of parameters your model has. You must be careful not to add too many parameters to your model.

(g) Recall your answer for the output of AlexNet's first conv layer from above. This output is fed directly into a max pool layer which applies a 3x3 pool filter at stride 2 with no padding. What will the output size be? How many trainable parameters does this layer introduce?

(h) Did the pool layer change the number of channels? Does this pattern generalize to pool layers of all sizes?

(i) AlexNet precipitated the deep learning revolution. Explain one of the paper's key contributions.

6. Kernel of Truth

Consider the following input matrix I and filter F.

As an aside, you may also hear a convolution filter referred to as a kernel, mask, or convolutional matrix.

$$I = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \\ -1 & 0 & 1 & 2 \\ 0 & -2 & 4 & 0 \end{bmatrix} \qquad F = \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix}$$

(a) Apply F on I with padding 0 and stride 1.

(b) Apply F on I with padding 1 and stride 2.

(c) Apply a max pool on I with a 3×3 filter using a padding of 1 and a stride of 3.[1]

[1]In practice, pool layers are usually applied after conv layers; they are typically *not* applied directly on the input.