CSE 493G1/599G1: Deep Learning

Solutions for Section 4: Backprop II & CNNs

Friday, January 26, 2024.

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

1. Sigmoid Shenanigans

Consider the Sigmoid activation function:

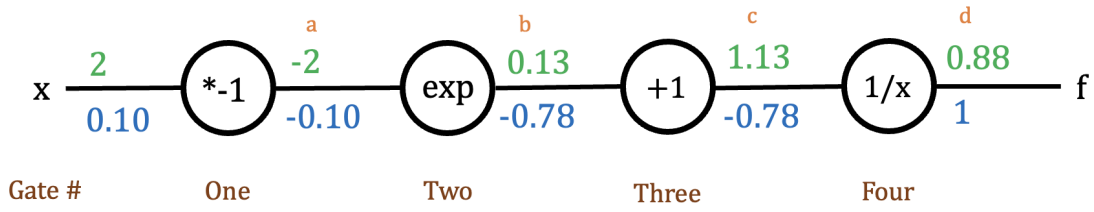
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Draw a computational graph and work through the backpropagation. Then, fill in the Python function. If you finish early, work through the analytical 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: $d(c(b(a(x)))) = \sigma(x)$.



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

> Recall that downstream = upstream × 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 $\frac{\partial}{\partial b}\Big(b + 1\Big) = 1$. Thus, the downstream is $-0.78 \times 1 = -0.78$.
>
> At Gate Two, the upstream is -0.78 and the local is $\frac{\partial}{\partial a}\Big(e^a\Big) = e^a = e^{-2} = 0.135$. Thus, the downstream is $-0.78 \times 0.135 = -0.10$.
>
> At Gate One, the upstream is -0.10 and the local is $\frac{\partial}{\partial x}\Big(-x\Big) = -1$. 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.

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`.

> 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, we the new value for x is $2 - 0.1(0.2) = 2 - 0.02 = 1.98$.

Python function printed on the following page.

```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
    f = 1/c
    out = f

    # TODO: backwards 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
```

2. Lost In The Sauce

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.

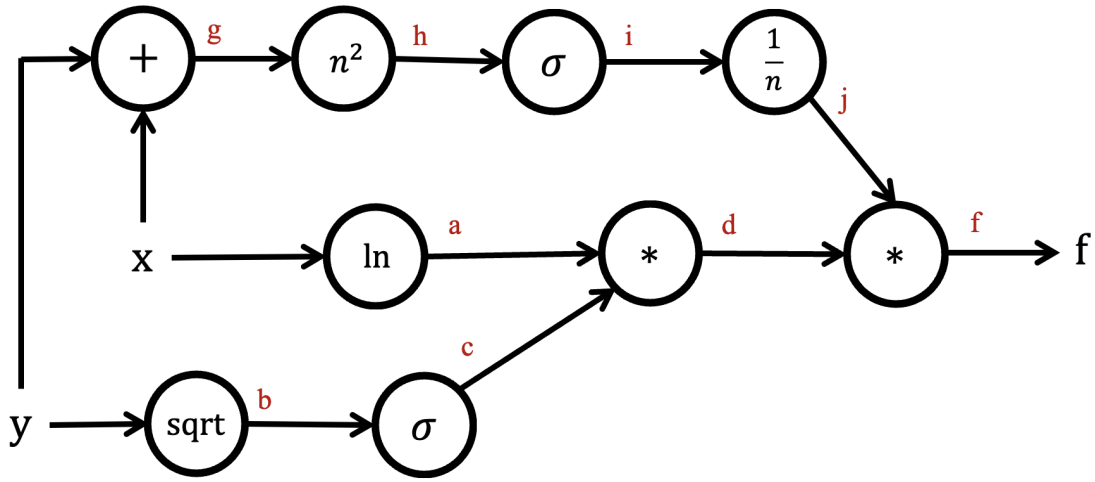
For reference, the derivative of Sigmoid is $\sigma(x) \cdot \big(1 - \sigma(x)\big)$.

The TA solution breaks f into equations a through i. Yours doesn't have to match this exactly.

We begin by breaking the function down:

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

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
```

3. As Convoluting As Possible

Suppose that both your input and filter are squares. Let W be the width of your input, F be the width of your filter, P be the amount of zero-padding utilized, and S be the stride.

Recall that convolution layers utilize a dot product. For instance, when you apply a $5 \times 5 \times 3$ conv filter, it reduces the corresponding 75 input values into 1 output value. One way to phrase this, borrowed from the cs231n notes, would be "every neuron in the conv layer has 180 connections to the input volume".

What's the formula for determining a conv layer's output size, ignoring padding?

$$\frac{W - F}{S} + 1$$

What's the formula for determining a conv layer's output size, taking padding into account?

$$\frac{W - F + 2P}{S} + 1$$

Consider a layer that takes a $32 \times 32 \times 3$ input and applies a $5 \times 5 \times 3$ filter (with stride 1 and no padding). What would the output size be?

> First, recognize $W = 32$, $F = 5$, and $S = 1$. Applying our formula, $\frac{W-F}{S} + 1 = \frac{32-5}{1} + 1 = 28$. So our output would be of size $28 \times 28 \times 1$

What would happen if we changed our stride to 2?

> This would not work, because the number of times our filter is "slid" across the input is not a whole number: $\frac{(W-F)}{S} + 1 = \frac{32-5}{2} + 1 = \frac{27}{2} + 1 = 13.5 + 1 = 14.5$

What about a stride of 3?

$$\frac{W - F}{S} + 1 = \frac{32 - 5}{3} + 1 = \frac{27}{3} + 1 = 9 + 1 = 10$$

> Our output volume would be $10 \times 10 \times 1$

What about a stride of 3 with a padding of 3 (i.e., three zero-values go on *all* sides of our input)?

$$\frac{W - F + 2P}{S} + 1 = \frac{32 - 5 + 2 \cdot 3}{3} + 1 = \frac{27 + 6}{3} + 1 = \frac{33}{3} + 1 = 11 + 1 = 12$$

> Our output volume would be $12 \times 12 \times 1$

Recall that filters must operate along all channels. Accordingly, people often leave out the channels dimension when writing out a filter size (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.

Consider the first conv layer of AlexNet, which took an input of size $227 \times 227 \times 3$. The layer applied 96 separate 11×11 filters with stride of 4 and no padding. You might sometimes see people write this as applying one 11×11 filter with depth 96; thy mean the same thing. What would our output dimensions be?

$$\frac{W-F}{S}+1=\frac{227-11}{S}+1=\frac{216}{4}+1=54+1=55$$

Our output volume would be $55 \times 55 \times 96$ because we applied 96 filters (or equivalently, our layer had a depth of 96).

Quoting from cs231n notes:

> Each of the $55 \cdot 55 \cdot 96$ neurons in this volume was connected to a region of size [11x11x3] in the input volume. Moreover, all 96 neurons in each depth column are connected to the same $[11 \times 11 \times 3]$ region of the input, but of course with different weights.

We can use K to refer to the number of filters we apply at a conv layer.

Also, let C be the number of channels in our input. For example, $C = 3$ for a RGB image.

For any given conv layer, there will be KF^2C parameters for the "weights" of our filters, and K parameters for our biases. More simply, there are $K(F^2C + 1)$ trainable parameters.

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

$$96 \cdot (11^2 \cdot 3 + 1) = 34944 \text{ trainable parameters}$$

Consider a conv layer which takes a $31 \times 31 \times 5$ input and applies 25 filters of size $3 \times 3 \times 5$ with stride 2 and padding 1. How many trainable parameters does it have?

$$25 \cdot (3^2 \cdot 5 + 1) = 1150 \text{ trainable parameters}$$

A key takeaway here is that the values you choose to 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, especially since we have limited compute in class.

Recall your answer for the output of AlexNet's first layer from above. The second layer applies a 3x3 **pool** filter at stride 2, again with no padding. What will the output size be? How many trainable parameters will this layer introduce?

> Recall from earlier that the output of the first layer (and thus the input to the second layer) has volume $55 \times 55 \times 96$.
>
> $$\frac{W-F}{S}+1=\frac{55-3}{2}+1=\frac{52}{2}+1=26+1=27$$
>
> There are no trainable parameters introduced at this layer since it is a POOL layer. Since pool layers have a hard-coded rule (i.e., max pool layers keep the maximum value in a given receptive field), they have nothing to "learn".

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

> No, it did not change. This generalizes since POOL layers always preserve depth.