CS 189 Spring 2023 — Introduction to Machine Learning — Jonathan Shewchuk

DIS10

1 Simple Gradients and their Interpretations

For each of the functions below, what are the partial derivatives with respect to each variable? For (a) only, what is ∇f? For parts (a)–(c), describe how changes in each variable affect f.

(a) $f(x, y) = xy$

Solution: $\frac{\partial f}{\partial x} = y$ and $\frac{\partial f}{\partial y} = x$. The derivative on each variable tells you the sensitivity of the entire expression to the variable. For example, consider $x = 1, y = -2$. If we increase the value of x by a tiny amount, this would cause the overall function to decrease by twice that tiny amount.

Recall that the gradient ∇f is the vector of partial derivatives: $\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right] = [y, x]$. Often for simplicity, the "partial derivative of the function on x" is referred to as the "gradient on x".

(b) $f(x, y) = x + y$

Solution: $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} = 1$. Increasing either x or y does not affect the *rate* of increase of f with respect to x or y, though it would increase the overall function value.

(c) $f(x, y) = \max\{x, y\}$

Solution: $\frac{\partial f}{\partial x} = \mathbb{1}(x \geq y)$ and $\frac{\partial f}{\partial y} = \mathbb{1}(y \geq x)$. In other words, the (sub)gradient is 1 on the input that was larger and 0 on the other input. This makes intuitive sense. The function will only be sensitive to changes in the max. For example, consider $x = 12, y = 3$, which has max $= 12$. The function is not sensitive to the value of y. If we were to keep increasing y by tiny amounts, the function would not change, and the gradient of the function would be 0 wrt y. We would have to change y by a lot in order to affect the value of f, but we know that the derivatives don't tell us anything about the effect of such large changes.

Note that if $x = y$, the derivatives are technically not defined. But for practical purposes, we can use 0 or 1 (or anything in between) for gradient descent, and it tends to work. (If you want to learn more, look up "subderivative" on Wikipedia.)

(d) $\tanh x = \dfrac{\sinh x}{\cosh x} = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$. Hint: $\tanh x = \dfrac{1 - e^{-2x}}{1 + e^{-2x}} = s(2x) - (1 - s(2x)) = 2s(2x) - 1$.

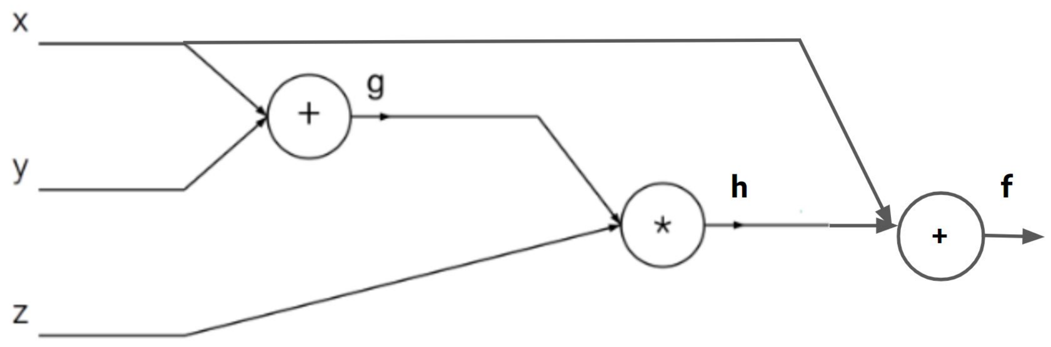
Solution: By the chain rule, the derivative is $4s'(2x) = 4s(2x)(1 - s(2x))$.

2 Backprop in Practice: Staged Computation

Consider the function $f(x, y, z) = (x + y)z + x$.

(a) Draw a directed acyclic graph (DAG)/circuit/network that represents the computation of f. Assign a variable name to each intermediate result.

Solution:



We use the intermediate variables $g = x + y$, $h = gz$, and $f = h + x$. The original expression $f(x, y, z)$ is simple enough to differentiate directly, but this approach of thinking about it in terms of subexpressions can help with understanding the intuition behind backprop.

(b) Write pseudocode for the forward pass and backward pass (backpropagation) in the network.

Solution:

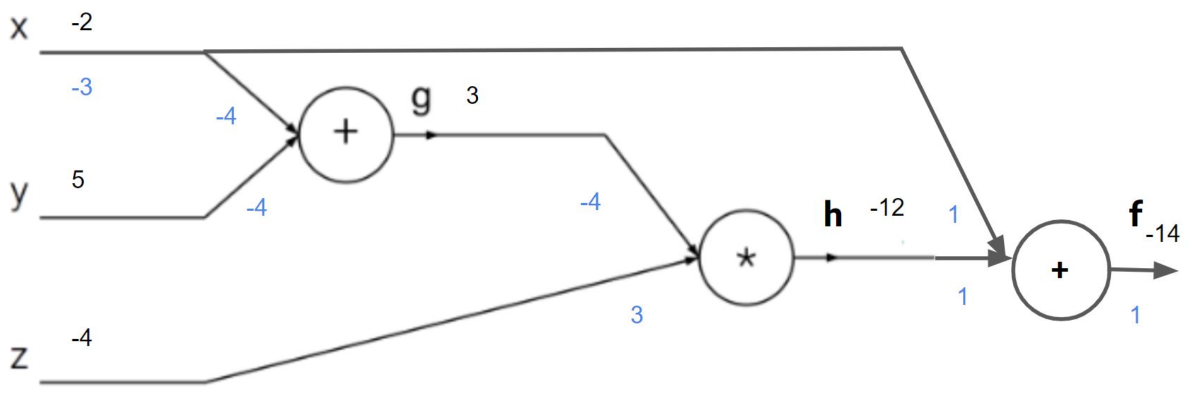
```
## forward pass
g = x + y
h = g * z
f = h + x

## backward pass
# backprop through f = h + x first
dfdh = 1.0
dfdx = 1.0
# backprop through h = g * z, using chain rule
dfdg = dfdh * z
dfdz = dfdh * g
# backprop through g = x + y, using chain rule
dfdx = 1.0 + dfdg * 1.0
dfdy = dfdg * 1.0
```

(c) On your network drawing, write the intermediate values in the forward and backward passes when the inputs are $x = -2$, $y = 5$, and $z = -4$.

Solution: We first run the forward pass (shown in black) of the network, which computes values from inputs to output. Then, we do the backward pass (blue), which starts at the end of the network and recursively applies the chain rule to compute the gradients. Think of the gradients as flowing backwards through the network. By the time we reach the beginning of the network in backprop, each gate will has learned about the gradient of its output value on the final output of the entire circuit. There are two takeaways about backpropagation: 1)

Backprop can thus be thought of as message passing (via the gradient signal) between gates about whether they want their outputs to increase or decrease, and how strongly, to make the final output value higher. 2) Backprop is a local process.



3 Model Intuition

(a) What can go wrong if you just initialize all the weights in a neural network to exactly zero? What about to the same nonzero value?

Solution: Either of these neural networks will have an undesirable property: symmetry.

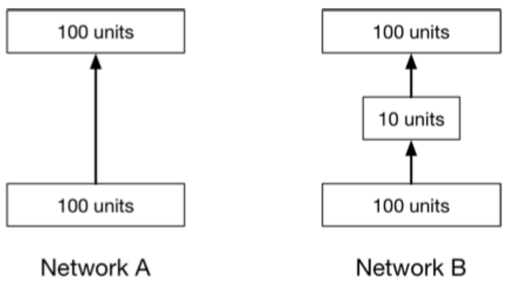
> Perhaps the only property known [about initialization] with complete certainty is that the initial parameters need to "break symmetry" between different units. If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters. If they have the same initial parameters, then a deterministic learning algorithm applied to a deterministic cost and model will constantly update both of these units in the same way.

–*Deep Learning*, p. 301 (Goodfellow, Bengio,& Courville)

(b) Adding nodes in the hidden layer gives the neural network more approximation ability, because you are adding more parameters. How many weight parameters are there in a neural network with architecture specified by $n = \left[n^{(0)}, n^{(1)}, ..., n^{(\ell)}\right]$, a vector giving the number of nodes in each of the $\ell + 1$ layers? (Layer 0 is the input layer, and layer ℓ is the output layer.) Evaluate your formula for a network n = [8, 10, 10, 3].

Solution: The number of parameters for the connections between two layers is $n^{(i)}n^{(i+1)}$. If we are computing the number of weights and biases, there is an additional +1 in the formula, which becomes $(n^{(i)} + 1)n^{(i+1)}$. The total number of weight parameters in the architecture is $\sum_{i=0}^{\ell-1} n^{(i)}n^{(i+1)}$. Applying this formula to the specified network gives $8\times10+10\times10+10\times3 = 210$ weights.

(c) Consider the two networks in the image below, where the added layer in Network B has 10 units with **linear activation**. Give one advantage of Network A over Network B, and one advantage of Network B over Network A.



Solution: Adding a linear-activated hidden layer does not make a network more powerful. It actually limits the functions that the network can represent.

Possible advantages of Network A over Network B: 1) A is more expressive than B. Specifically, it can learn any function that B can learn, plus some additional functions. 2) A has fewer layers, so it is less susceptible to problems with exploding or vanishing gradients.

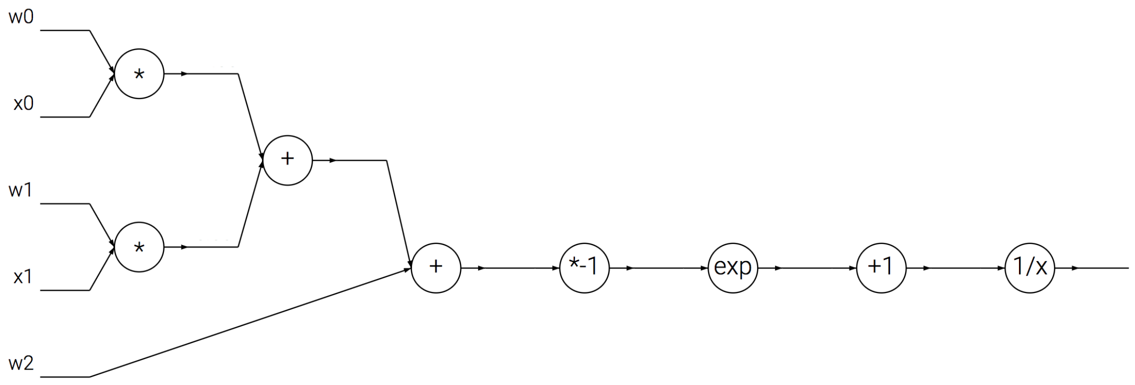
Possible advantages of Network B over Network A: 1) B is computationally cheaper because a matrix-vector product of size 10×100 followed by one of size 100×10 requires fewer operations than one of size 100×100. 2) B has an embedding (or bottleneck) layer, so the network is forced to learn a more compact representation. Bottleneck layers like these are used in practice to create compact representations of data.

4 More Backprop in Practice: Staged Computation

Consider the function $f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$.

(a) Draw a network that represents the computation of f.

Solution:



In addition to add and multiply gates, we use two other operations in the diagram above.

$$
\begin{aligned}
g(x) &= \frac{1}{x} \to \frac{dg}{dx} = -\frac{1}{x^2}. \\
h(x) &= e^x \to \frac{dh}{dx} = e^x.
\end{aligned}
$$

(b) Write pseudocode for the forward pass and backward pass (backpropagation) of the network.

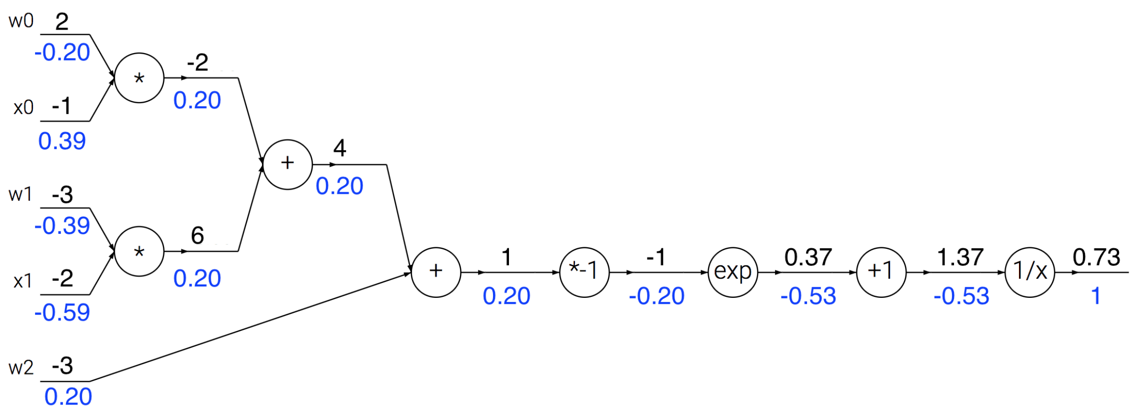
Solution:

```
## forward pass
dot = w[0]*x[0] + w[1]*x[1] + w[2]
f = 1.0 / (1 + math.exp(-dot)) # sigmoid function

## backprop
ddot = (1 - f) * f # gradient on dot variable, using the sigmoid gradient
dx0 = w[0] * ddot # backprop into x
dx1 = w[1] * ddot
dw0 = x[0] * ddot # backprop into w
dw1 = x[1] * ddot
dw2 = 1.0 * ddot
```

(c) With the weights $w = [2,-3,-3]$ and inputs $x = [-1,-2]$, write the intermediate values in the forward and backward passes on your network diagram.

Solution: Again, the forward pass is in black and the backward pass is in blue.



(d) Now consider a network that computes the function $f(x, y) = \frac{x + s(y)}{s(x) + (x + y)^2}$. Write pseudocode for the forward and backward passes of the network.

Solution: You would end up with a very large and complex expression if you tried to directly differentiate $f(x, y)$ with respect to x or y. Thankfully, we don't need an explicit function for evaluating the gradient because we only need to know *how* to compute it.

```
## forward pass
sigy = 1.0 / (1 + math.exp(-y)) # sigmoid in numerator   #(1)
num = x + sigy # numerator                               #(2)
sigx = 1.0 / (1 + math.exp(-x)) # sigmoid in denominator #(3)
xpy = x + y                                              #(4)
xpysqr = xpy**2                                          #(5)
den = sigx + xpysqr # denominator                        #(6)
invden = 1.0 / den                                       #(7)
f = num * invden # done!                                 #(8)
```

The code is structured such that it contains multiple intermediate variables, each is a simple expression for which we already know the local gradient. This makes backprop easy because for every variable along the way in the forward pass (`sigy`, `num`, `sigx`, `xpy`, `xpysqr`, `den`, `invden`) we will have a corresponding variable that begins with a `d`, which will hold the gradient of the output of the circuit with respect to that variable. In the comments, we highlight which part of the forward pass this computation corresponds to.

```
## backward pass
# backprop f = num * invden
dnum = invden # gradient on numerator                              #(8)
dinvden = num                                                      #(8)
# backprop invden = 1.0 / den
dden = (-1.0 / (den**2)) * dinvden                                 #(7)
# backprop den = sigx + xpysqr
dsigx = (1) * dden                                                 #(6)
dxpysqr = (1) * dden                                               #(6)
# backprop xpysqr = xpy**2
dxpy = (2 * xpy) * dxpysqr                                         #(5)
# backprop xpy = x + y
dx = (1) * dxpy                                                    #(4)
dy = (1) * dxpy                                                    #(4)
# backprop sigx = 1.0 / (1 + math.exp(-x))
dx += ((1 - sigx) * sigx) * dsigx # Notice we accumulate gradients  #(3)
# backprop num = x + sigy
dx += (1) * dnum                                                   #(2)
dsigy = (1) * dnum                                                 #(2)
# backprop sigy = 1.0 / (1 + math.exp(-y))
dy += ((1 - sigy) * sigy) * dsigy                                  #(1)
```