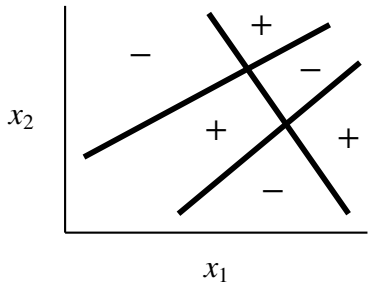
CS 189 Introduction to Machine Learning
Summer 2019 Marc Khoury & Brijen Thananjeyan

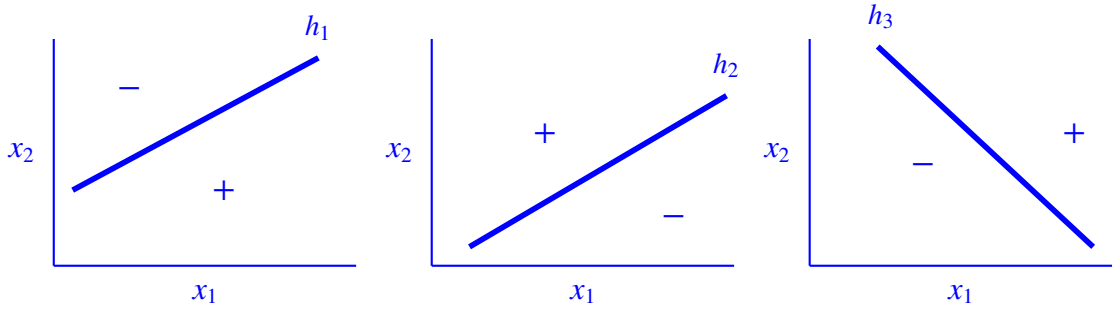
DIS11

1 The Multi-layer Perceptron (MLP)

(a) Consider a target function whose + and − regions are illustrated below. Draw the perceptron components, and express f as a Boolean function of its perceptron components.



Solution: The perceptrons correspond to the three lines in the target function. The choice of signs for each one is a bit arbitrary, but ours makes the notation in the next part convenient.



To form the Boolean function, we focus on when f is +. The top + region contributes $\bar{h}_1 h_2 h_3$ to f. The lower left + region contributes $h_1 h_2 \bar{h}_3$ to f. The lower right + region contributes $h_1 \bar{h}_2 h_3$ to f. Combining these, we have: $f = \bar{h}_1 h_2 h_3 + h_1 h_2 \bar{h}_3 + h_1 \bar{h}_2 h_3$.

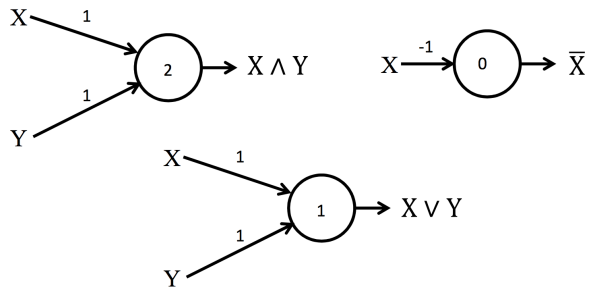
Take-away: a complicated target function, such as the one above, that is composed of perceptrons can be expressed as an `OR` of `AND`s of the component perceptrons.

Let's develop some intuition for why this might be useful.

(b) Over two inputs x_1 and x_2, how can `OR`, `AND`, and `NOT` each be implemented by a single perceptron? Assume each unit uses a *hard threshold* activation function. Recall the hard-threshold function: for a constant a,

$$h(x) = \begin{cases} 1, & \text{if } w^\top x \geq a \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Solution:



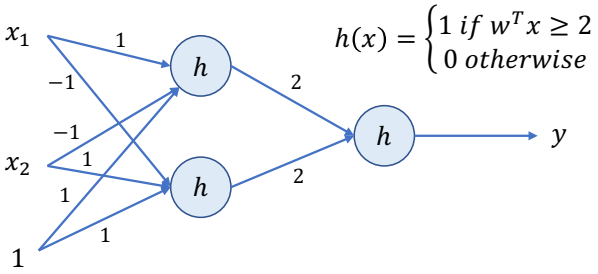
(c) For any Boolean function f, can we build a neural network with at most one hidden layer to compute f?

Solution: We can prove this by construction. Any Boolean function can be represented using a combination of `NOT`, `AND`, `OR` gates. Each of these gates is easily represented by a neural network. We don't provide a formal proof of this statement, but instead provide an example to motivate your intuition.

Consider the boolean function from part (a), which was $f(x_1, x_2, x_3) = \bar{x}_1 x_2 x_3 + x_1 x_2 \bar{x}_3 + x_1 \bar{x}_2 x_3$. This can be expressed as a 3-input network with 3 nodes in a hidden layer that goes to a summation node with scalar output. Each node in the hidden layer computes one of the expressions in the sum.

(d) We have seen that `XOR` is not linearly separable, and hence it cannot be implemented by a single perceptron. Draw a fully connected three unit neural network that has binary inputs $X_1, X_2, \mathbf{1}$ and output Y, where Y implements the `XOR` function: $Y = \texttt{XOR}(X_1, X_2)$. Again, assume each unit uses a *hard threshold* activation function.

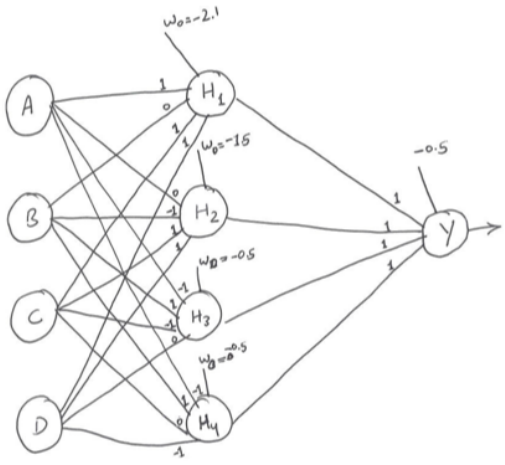
Solution:



(e) (self-study) Create a neural network with a single hidden layer (of any number of units) that implements the function: (A or not B) XOR (not C or not D).

Solution: `XOR` can be expressed in terms of `AND` and `OR` functions: p `XOR` $q = (p\bar{q}) + (\bar{p}q)$. Therefore, we can re-write the original expression as: ((A or not B) and not (not C or not D)) or (not (A or not B) and (not C or not D)). Simplifying gives: (A and C and D) or (not B and C and D) or (not A and B and not C) or (not A and B and not D).

We can represent this with 4 inputs (A,B,C,D), and 4 hidden nodes in one hidden layer that feed into the summation node that gives the output. Each hidden node corresponds to one `AND` expression (i.e., one expression in parentheses) in the simplified expression of `OR`s above. One possible network that implements this is:

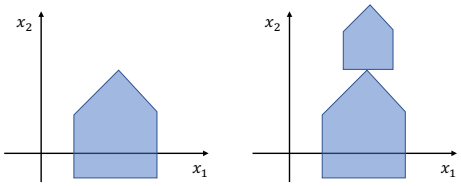


Take-away: If f can be decomposed into perceptrons using an `OR` of `AND`s, then it can be implemented by a 3-layer perceptron. A one-hidden-layer MLP is a universal Boolean function. How cool is that?!

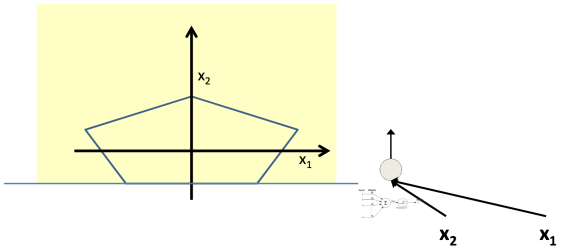
Solution: Very cool!! Though in practice though this is not particularly useful, as it can require an exponentially large number of perceptrons.

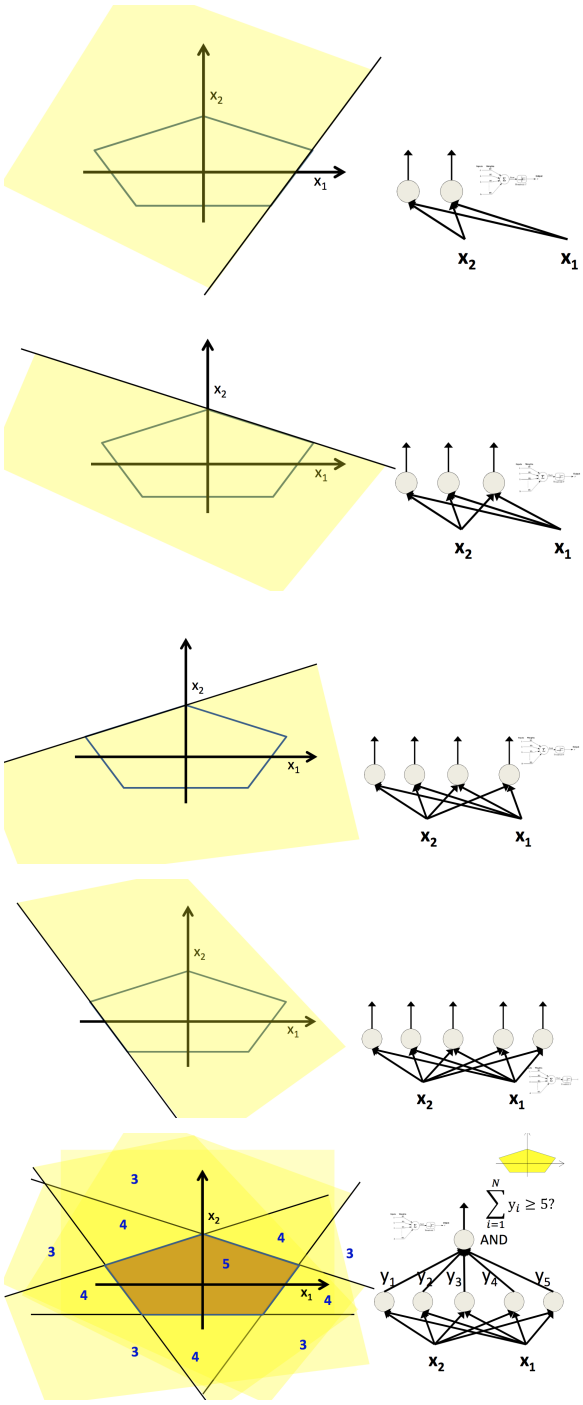
2 Decision Space

Let's further the intuition about how we can compose arbitrarily complex decision boundaries with a neural network. Consider the images below. For each one, build a network of units with a single output that fires if the input is in the shaded area.

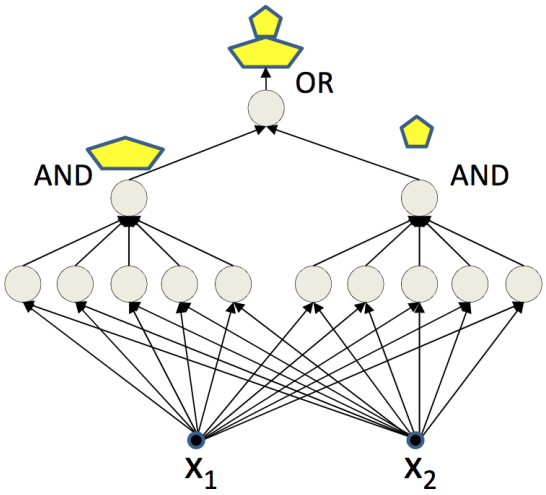


Solution: This is like the perceptron composition we saw in the first problem. But now, we have the composition of 5 `OR`s and 1 `AND`. The following sequence of figures builds up the composition, where the network's new unit fires if the input is in the (darkest) shaded area.





This network is a composition of the previous network, and a new network that captures the smaller pentagon above.

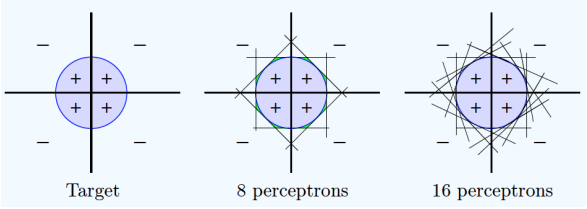


Take-away: MLPs can capture any classification boundary. MLPs are universal classifiers. Note that we haven't said anything yet about their ability to generalize.

3 What else can a network represent?

So far, we have seen target functions that are exactly decomposable into perceptrons. Even in the case where f is not decomposable into perceptrons, if f is a continuous function then there exists a 3-layer perceptron that approximates f arbitrarily closely. The formal proof for this is beyond the scope of the class. The important point is: how can perceptrons be used to approximate functions with curves? Consider a disc target function (i.e., a circle with positive on the inside and negative on the outside), and illustrate how perceptrons could approximate this f.

Solution: Again, the formal proof of the universal function approximator statement is a bit beyond the scope of this class. If we think of the perceptron as an analog to the step function, then our statement to prove is somewhat analogous to the calculus theorem that says that any continuous function on a compact set can be approximated arbitrarily closely by step functions. In lieu of a proof, we provide an example with a disc target function to motivate your intuition.



Take-away: Neural networks are universal function approximators. Wow!! Refer to this for an intuitive explanation and visual proof. It's important to note that neural networks are not the only universal function approximators. **Polynomials** were the first provable universal approximators, see the Stone–Weierstrass theorem. In addition, the **Fourier** basis is a popular function approximator in, e.g., signal processing. This Quora post has more details, and a cute picture of the three

bases (polynoial, Fourier, and neural network) fitting functions.

We still haven't said anything about the ability to generalize. There is a classic line from John von Neumann on overfitting: With four parameters I can fit an elephant, and with five I can make him wiggle his trunk. Jürgen Mayer took this statement literally in a 2009 publication, *Drawing an elephant with four complex parameters*. You can see the elephant wiggling the trunk here.

Okay, now that neural nets sound amazingly expressive, how can we use them?! A neural network is parameterized by the weights and biases of its units. We use gradient descent to update the network parameters, which means the updates are based on the gradients of the parameters. Backpropagation is a way of computing gradients of expressions in a network through recursive application of the chain rule. Your understanding and intuition of backpropagation will guide you as you design, develop, and debug your neural networks.

4 Simple gradients and their interpretations

For each of the functions below, what are the partial derivatives with respect to each variable? For the first three functions only, what is the interpretation? For the first function only, what is ∇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 on its value. 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 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, 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.

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

Solution: Taking the derivative via chain rule, we have

$$\sigma'(x) = -\frac{1}{(1+e^{-x})^2}(-e^{-x}) = \frac{1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right) = \sigma(x)(1-\sigma(x)).$$

(e) (self-study) $\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

Solution: Notice that $\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}} = \sigma(2x) - (1 - \sigma(2x)) = 2\sigma(2x) - 1$.

Hence, by chain rule, it is clear that the derivative is just $4\sigma'(2x) = 4\sigma(2x)(1 - \sigma(2x))$.

5 Backrop in practice: staged computation

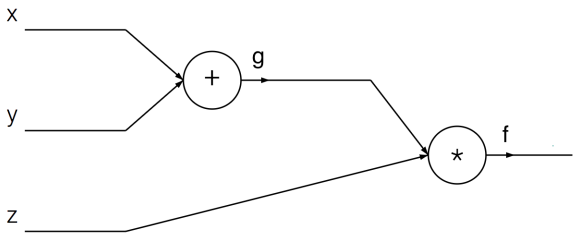
For the function $f(x, y, z) = (x + y)z$:

(a) Decompose f into two simpler functions.

Solution: We can conveniently express f as: $g = x + y$, $f = gz$. The original $f(x, y, z)$ expression is simple enough to differentiate directly, but this particular approach of thinking about it in terms of its decomposition can help with understanding the intuition behind backprop.

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

Solution:



(c) Write the forward pass and backward pass (backpropagation) in the network.

Solution:

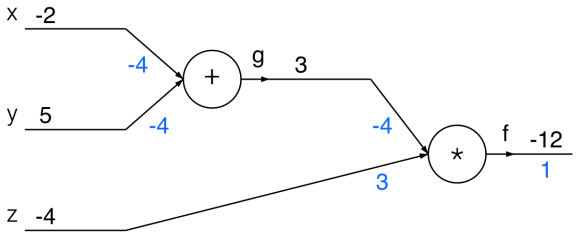
```
## forward pass
g = x + y
f = g * z

## backward pass
# backprop through f = gz first
dfdz = g
dfdg = z
# backprop through g = x + y, using chain rule
dfdx = 1.0 * dfdg # df/dx = dg/dx * df/dg
dfdy = 1.0 * dfdg # dg/dy = dg/dx = 1
```

(d) Update your network drawing with the intermediate values in the forward and backward pass. Use the following inputs $x = -2$, $y = 5$, 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 take-aways about backprop from this: 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), so as to make the final output value higher. 2) Backprop is a local process.



6 More backrop in practice: staged computation (self-study)

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

(a) Decompose f into two simpler functions.

Solution: The wording of this part is a bit off, as it was copied directly from the previous problem. In the previous problem, it made sense to think of $f(x,y,z)$ in terms of the decomposition of only two functions. For this problem, that doesn't make sense as much sense, as $f(w,x)$ is made up of more than just two gates. In particular, in addition to the add and multiply gates that we saw in the previous problem, $f(w,x)$ is as also composed of:

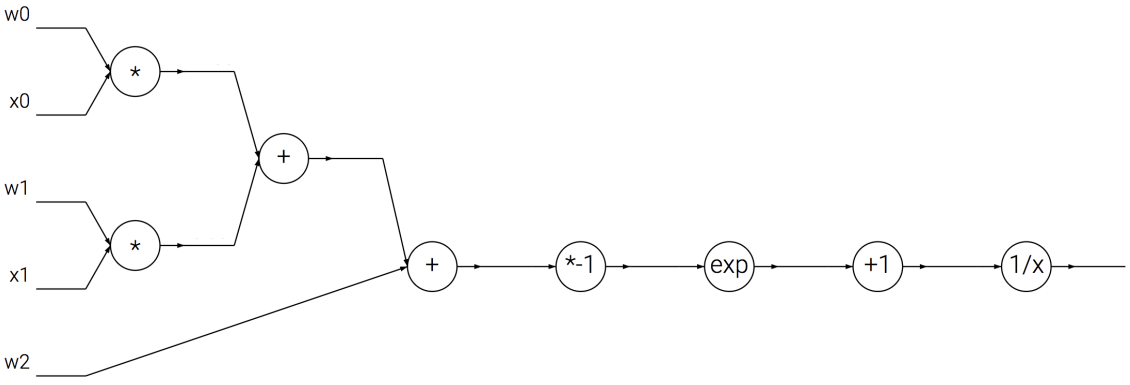
$$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -\frac{1}{x^2}$$

$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$$

Which parts of the forward function are considered gates is a matter of convenience. As we'll see in later parts, it is useful to be aware of which parts of f have easy local gradients. This way when everything is chained together, we've minimized the code and effort for backprop.

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

Solution:



(c) Write the forward pass and backward pass (backpropagation) in the network.

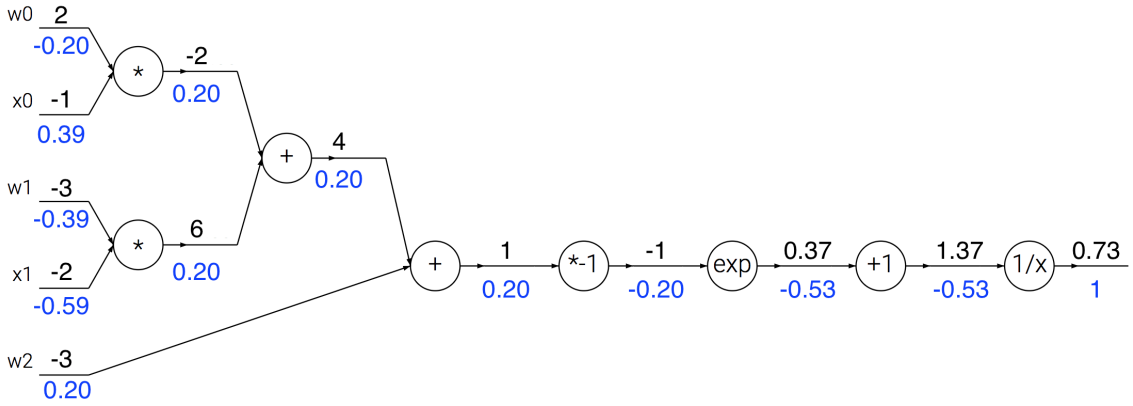
Solution:

```
## 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 derivation from 4d
dx = [w[0] * ddot, w[1] * ddot] # backprop into x
dw = [x[0] * ddot, x[1] * ddot, 1.0 * ddot] # backprop into w
```

(d) Update your network drawing with the intermediate values in the forward and backward pass. Use the following weight initialization and inputs $w = [2, -3, -3]$ and $x = [-1, -2]$.

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



(e) Now consider the function $f(x, y) = \frac{x+\sigma(y)}{\sigma(x)+(x+y)^2}$. What are the forward and backward passes?

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)
```

7 Backpropagation Practice (self-study)

(a) Chain rule of multiple variables: Assume that you have a function given by $f(x_1, x_2, \ldots, x_n)$, and that $g_i(w) = x_i$ for a scalar variable w. How would you compute $\frac{\mathrm{d}}{\mathrm{d}w} f(g_1(w), g_2(w), \ldots, g_n(w))$? What is its computation graph?

Solution: This is the chain rule for multiple variables. In general, we have

$$\frac{\mathrm{d}f}{\mathrm{d}w} = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}\frac{\partial x_i}{\partial w} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial w}.$$

The function graph of this computation is given in Figure 1.

(b) Let $w_1, w_2, \ldots, w_n \in \mathbb{R}^d$, and we refer to these variables together as $W \in \mathbb{R}^{n\times d}$. We also have $x \in \mathbb{R}^d$ and $y \in \mathbb{R}$. Consider the function

$$f(W, x, y) = \left(y - \sum_{i=1}^{n} \phi(w_i^\top x + b_i)\right)^2.$$

Write out the function computation graph (also sometimes referred to as a pictorial representation of the network). This is a directed graph of decomposed function computations, with the function at one end, and the variables W, b, x, y at the other end, where $b = [b_1, \cdots, b_n]$.

Solution:

See Figure 2.

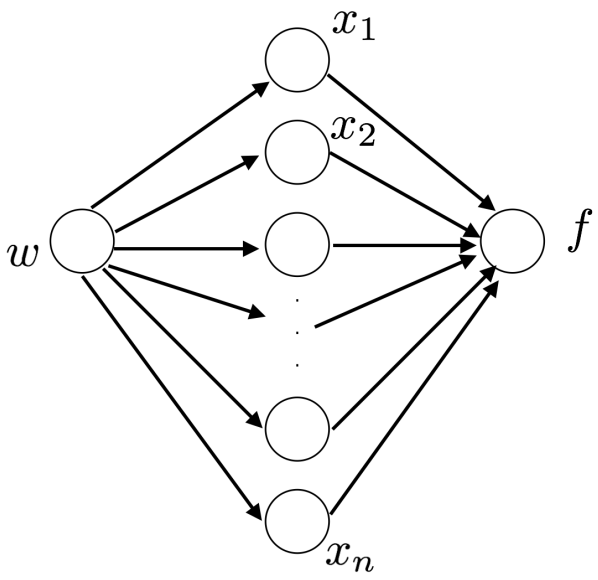


Figure 1: Example function computation graph

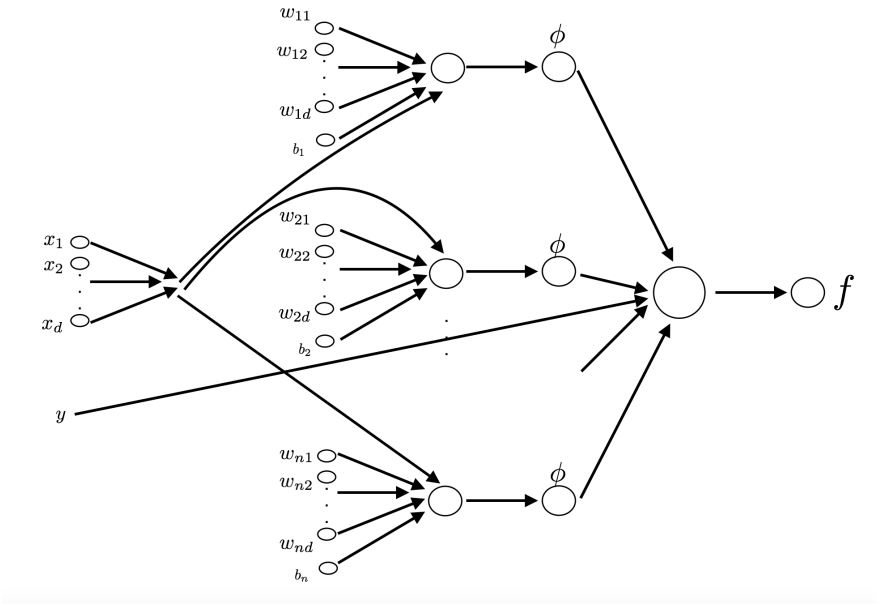


Figure 2: Example function computation graph

(c) Suppose $\phi(x) = \sigma(x)$ (from problem 1a). Compute the partial derivatives $\frac{\partial f}{\partial w_i}$ and $\frac{\partial f}{\partial b_i}$. Use the computational graph you drew in the previous part to guide you.

Solution: Denote $r = y - \sum_{i=1}^{n} \sigma(w_i^\top x + b_i)$ and $z_i = w_i^\top x + b_i$.

To remind ourselves, this is the 'forward' computation:

$$f = r^2$$

$$r = y - \sum_{i=1}^{n} \sigma(z_i)$$
$$z_i = w_i^\top x + b_i$$

Now the backward pass:

$$\frac{\partial f}{\partial r} = 2r$$
$$\frac{\partial r}{\partial z_i} = -\sigma(z_i)(1 - \sigma(z_i))$$
$$\frac{\partial z_i}{\partial w_i} = x^\top$$
$$\frac{\partial z_i}{\partial b_i} = 1$$

By applying chain rule

$$\frac{\partial f}{\partial w_i} = 2(\sum_{i=1}^{n} \sigma(w_i^\top x + b_i) - y)\sigma(w_i^\top x + b_i)(1 - \sigma(w_i^\top x + b_i))x^\top$$
$$\frac{\partial f}{\partial b_i} = 2(\sum_{i=1}^{n} \sigma(w_i^\top x + b_i) - y)\sigma(w_i^\top x + b_i)(1 - \sigma(w_i^\top x + b_i))$$

(d) Write down a single gradient descent update for $w_i^{(t+1)}$ and $b_i^{(t+1)}$, assuming step size η. You answer should be in terms of $w_i^{(t)}$, $b_i^{(t)}$, x, and y.

Solution:

$$w_i^{(t+1)} \leftarrow w_i^{(t)} - 2\eta(\sum_{i=1}^{n} \sigma({w_i^{(t)}}^\top x + b_i^{(t)}) - y)\sigma({w_i^{(t)}}^\top x + b_i^{(t)})(1 - \sigma({w_i^{(t)}}^\top x + b_i^{(t)}))x$$
$$b_i^{(t+1)} \leftarrow b_i^{(t)} - 2\eta(\sum_{i=1}^{n} \sigma({w_i^{(t)}}^\top x + b_i^{(t)}) - y)\sigma({w_i^{(t)}}^\top x + b_i^{(t)})(1 - \sigma({w_i^{(t)}}^\top x + b_i^{(t)}))$$

(e) (optional) Define the cost function

$$\ell(x) = \frac{1}{2}\|W^{(2)}\Phi\left(W^{(1)}x + b\right) - y\|_2^2, \tag{2}$$

where $W^{(1)} \in \mathbb{R}^{d\times d}$, $W^{(2)} \in \mathbb{R}^{d\times d}$, and $\Phi : \mathbb{R}^d \to \mathbb{R}^d$ is some nonlinear transformation. Compute the partial derivatives $\frac{\partial \ell}{\partial x}, \frac{\partial \ell}{\partial W^{(1)}}, \frac{\partial \ell}{\partial W^{(2)}}$, and $\frac{\partial \ell}{\partial b}$.

Solution: First, we write out the intermediate variable for our convenience.

$$x^{(1)} = W^{(1)}x + b$$

$$
\begin{aligned}
x^{(2)} &= \Phi(x^{(1)}) \\
x^{(3)} &= W^{(2)} x^{(2)} \\
x^{(4)} &= x^{(3)} - y \\
\ell &= \frac{1}{2}\|x^{(4)}\|_2^2.
\end{aligned}
$$

Remember that the superscripts represents the index rather than the power operators. We have

$$
\begin{aligned}
\frac{\partial \ell}{\partial x^{(4)}} &= x^{(4)\top} \\
\frac{\partial \ell}{\partial x^{(3)}} &= \frac{\partial \ell}{\partial x^{(4)}} \frac{\partial x^{(4)}}{\partial x^{(3)}} = \frac{\partial \ell}{\partial x^{(4)}} \\
\frac{\partial \ell}{\partial x^{(2)}} &= \frac{\partial \ell}{\partial x^{(3)}} \frac{\partial x^{(3)}}{\partial x^{(2)}} = \frac{\partial \ell}{\partial x^{(3)}} W^{(2)} \\
\frac{\partial \ell}{\partial W^{(2)}} &= \frac{\partial \ell}{\partial x^{(3)}} \frac{\partial x^{(3)}}{\partial W^{(2)}} = x^{(2)} \frac{\partial \ell}{\partial x^{(3)}} \\
\frac{\partial \ell}{\partial x^{(1)}} &= \frac{\partial \ell}{\partial x^{(2)}} \frac{\partial \Phi}{\partial x^{(1)}} \\
\frac{\partial \ell}{\partial x} &= \frac{\partial \ell}{\partial x^{(1)}} \frac{\partial x^{(1)}}{\partial x} = \frac{\partial \ell}{\partial x^{(1)}} W^{(1)} \\
\frac{\partial \ell}{\partial b} &= \frac{\partial \ell}{\partial x^{(1)}} \frac{\partial x^{(1)}}{\partial b} = \frac{\partial \ell}{\partial x^{(1)}} \\
\frac{\partial \ell}{\partial W^{(1)}} &= \frac{\partial \ell}{\partial x^{(1)}} \frac{\partial x^{(1)}}{\partial W^{(1)}} = x \frac{\partial \ell}{\partial x^{(1)}}.
\end{aligned}
$$

A more formal way to solve these requires doing the expansion. For example, $\frac{\partial \ell}{\partial W^{(1)}} = \frac{\partial \ell}{\partial x^{(1)}} \frac{\partial x^{(1)}}{\partial W^{(1)}}$. However, the right hand side is a tensor, in which the matrix codebook does not provide us a useful information. We need to do that manually. Notice that $x_k^{(1)} = \sum_l W_{kl}^{(1)} x_l + b_k$, we have

$$
\begin{aligned}
\frac{\partial \ell}{\partial W_{ij}^{(1)}} &= \sum_k \frac{\partial \ell}{\partial x_k^{(1)}} \frac{\partial x_k^{(1)}}{\partial W_{ij}^{(1)}} \\
&= \sum_k \sum_l \frac{\partial \ell}{\partial x_k^{(1)}} \left(\epsilon_{ik} \epsilon_{jl} x_l \right) \\
&= \frac{\partial \ell}{\partial x_i^{(1)}} x_j
\end{aligned}
$$

so that

$$
\frac{\partial \ell}{\partial W^{(1)}} = \frac{\partial \ell}{\partial x^{(1)}} \frac{\partial x^{(1)}}{\partial W^{(1)}} = x \frac{\partial \ell}{\partial x^{(1)}}. \tag{3}
$$

(f) (optional) Suppose Φ is the identity map. Write down a single gradient descent update for $W_{t+1}^{(1)}$ and $W_{t+1}^{(2)}$ assuming step size η. Your answer should be in terms of $W_t^{(1)}, W_t^{(2)}, b_t$ and x, y.

Solution:

$$
W_{t+1}^{(1)} \leftarrow W_t^{(1)} - \eta (W_t^{(2)})^\top \left(W_t^{(2)} \left(W_t^{(1)} x + b_t \right) - y \right) x^\top
$$

$$W_{t+1}^{(2)} \leftarrow W_t^{(2)} - \eta\left(W_t^{(2)}\left(W_t^{(1)}x + b_t\right) - y\right)(W_t^{(1)}x + b)^\top$$

Side note: The computation complexity of computing the $\frac{\partial \ell}{\partial W}$ for Equation (2) using the analytic derivatives and numerical derivatives is quite different!

For numerical differentiation, what we do is to use the following first order formula

$$\frac{\partial \ell}{\partial W_{ij}} = \frac{\ell\left(W_{ij} + \epsilon, \cdot\right) - \ell\left(W_{ij}, \cdot\right)}{\epsilon}.$$

We need $O(d^4)$ operations in order to compute $\frac{\partial \ell}{\partial W}$. On the other hand, it only takes $O(d^2)$ operations to compute it analytically.

8 Model Intuition (self-study)

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

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

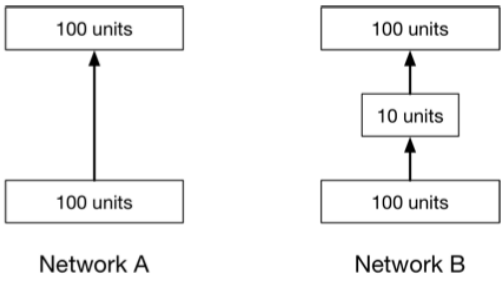
Perhaps the only property known [about initialization] with complete certainty is that the initial parameters need to "break symmetry" between different units. If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters. 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.

–page 301 of Deep Learning by Ian Goodfellow, Yoshua Bengio, Aaron 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 $d = \left[d^{(0)}, d^{(1)}, ..., d^{(N)}\right]$, a vector giving the number of nodes in each of the N layers? Evaluate your formula for a 2 hidden layer network with 10 nodes in each hidden layer, an input of size 8, and an output of size 3.

Solution: The number of parameters for the connections between two layers is: $d^{(i)} \times d^{(i+1)}$. Note that if we were computing the number of weights and biases, there would be an additional +1 in the formula, which would become: $(d^{(i)} + 1) \times d^{(i+1)}$. The total number of weight parameters in the architecture specified by d is: $\sum_{i=0}^{N-1} d^{(i)} \times d^{(i+1)}$. Applying this formula to the supplied network gives: $8 \times 10 + 10 \times 10 + 10 \times 3 = 210$ weights.

(c) Consider the two networks in the image below, where the added layer in going from Network A to 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 has fewer parameters, so it is less prone to overfitting. 2) 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. 3) B has an embedding (or bottleneck) layer, so the network is forced to learn a more compact representation.