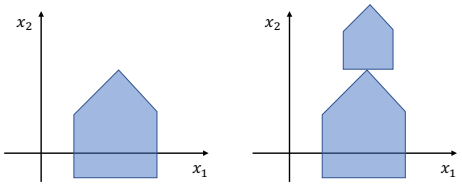
CS 189 Introduction to Machine Learning
Spring 2021 Jonathan Shewchuk

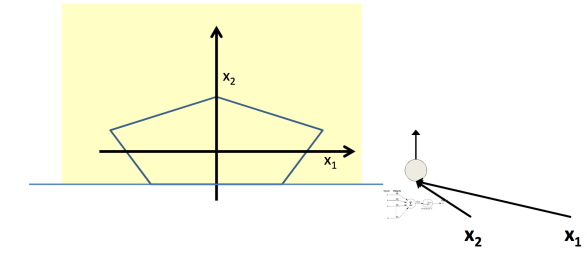
DIS11

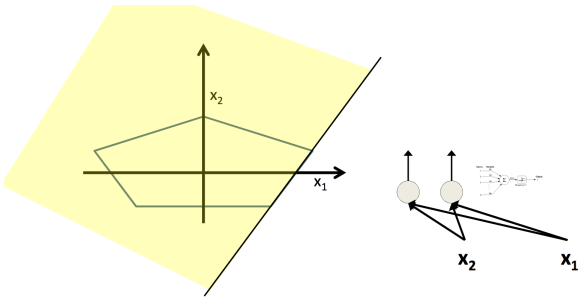
1 Decision Space

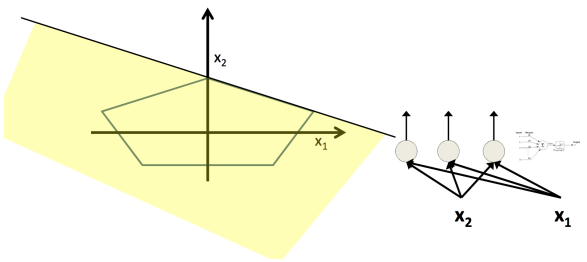
Let's further the intuition about how we can compose arbitrarily complex decision boundaries with a neural network. Consider the images below. For each one, build a network of units with a single output that fires if the input is in the shaded area.

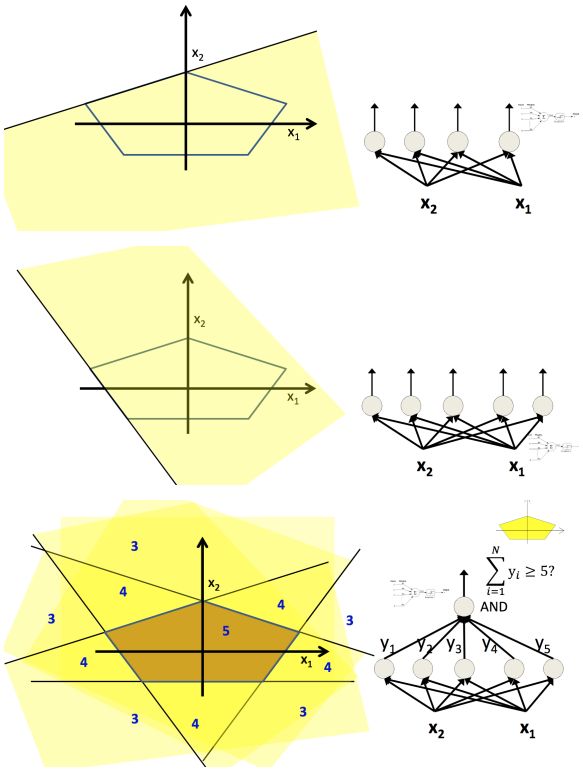


Solution: This is like the perceptron composition we saw in the first problem. But now, we have the composition of 5 `OR`s and 1 `AND`. The following sequence of figures builds up the composition, where the network's new unit fires if the input is in the (darkest) shaded area.

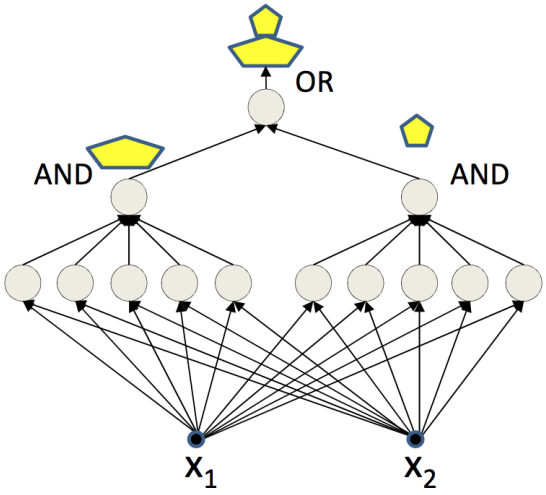








This network is a composition of the previous network, and a new network that captures the smaller pentagon above.



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.

2 Backrop in Practice: Staged Computation

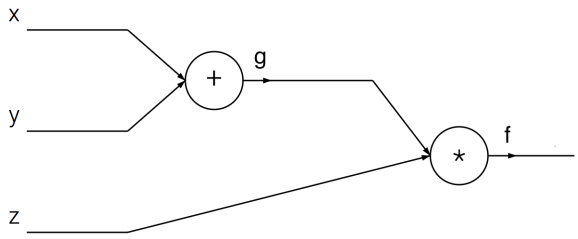
For the function $f(x, y, z) = (x + y)z$:

(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.

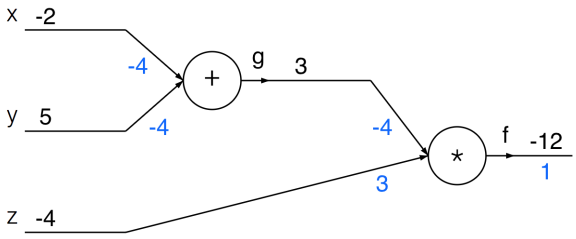
Solution:

```
## 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 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.



3 Backpropagation Practice

(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.

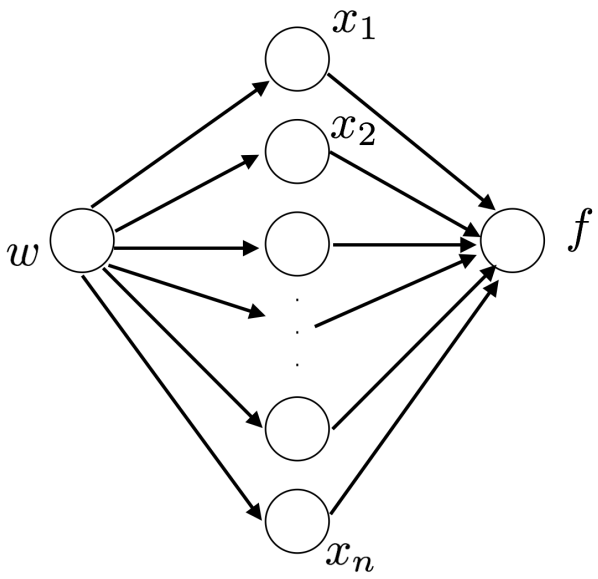


Figure 1: Example function computation graph

(b) Let $Z = XW + \mathbf{1}b$, where $Z \in \mathbb{R}^{d_n \times d_{out}}$, $X \in \mathbb{R}^{d_n \times d_{in}}$, $W \in \mathbb{R}^{d_{in} \times d_{out}}$, b is a row vector in $\mathbb{R}^{d_{out}}$, and $\mathbf{1}$ is a column vector in $\mathbb{R}^{d_{in}}$. Given $\frac{\partial L}{\partial Z} \in \mathbb{R}^{d_n \times d_{out}}$, where l is a scalar loss, calculate $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$.

Solution: For $\frac{\partial L}{\partial b}$:

$$\frac{\partial L}{\partial b_i} = \sum_{j,k} \frac{\partial L}{\partial Z_{jk}} \frac{\partial Z_{jk}}{\partial b_i} = \sum_{j} \frac{\partial L}{\partial Z_{ji}} \frac{\partial Z_{ji}}{\partial b_i} = \sum_{j} \frac{\partial L}{\partial Z_{ji}}$$

The second "=" sign is because only when $k = i$, $\frac{\partial Z_{jk}}{\partial b_i}$ is nonzero. Therefore

$$\frac{\partial L}{\partial b} = [\sum_j \frac{\partial L}{\partial Z_{j1}}, \sum_j \frac{\partial L}{\partial Z_{j2}} \cdots] = \mathbf{1}^T \frac{\partial L}{\partial Z}$$

For $\frac{\partial L}{\partial W}$, we note that $Z_{ij} = \sum_{k=1}^{d} X_{ik} W_{kj} + b_j$

$$\frac{\partial L}{\partial W_{ab}} = \sum_{i,j} \frac{\partial L}{\partial Z_{ij}} \frac{\partial Z_{ij}}{\partial W_{ab}} = \sum_i \frac{\partial L}{\partial Z_{ib}} X_{ia} = (\frac{\partial L}{\partial Z_{:,b}})^T (X_{:,a})$$

where $\frac{\partial L}{\partial Z_{:,b}}$ is a column of $\frac{\partial L}{\partial Z}$ and $X_{:,a}$ is a column of X. The second "=" sign is because $\frac{\partial Z_{ij}}{\partial W_{ab}} = X_{ia}$ if $j = b$ else 0.

Therefore,

$$\frac{\partial L}{\partial W} = \begin{bmatrix} & \cdots & \\ — & \text{column } a \text{ of X} & — \\ & \cdots & \end{bmatrix} \begin{bmatrix} & | & \\ \cdots & \text{column } b \text{ of } \frac{\partial L}{\partial Z} & \cdots \\ & | & \end{bmatrix}$$

Thus

$$\frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial Z}$$

4 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.

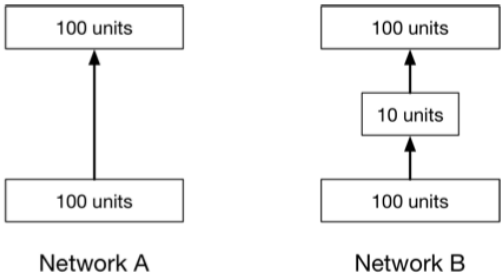
Perhaps the only property known [about initialization] with complete certainty is that the initial parameters need to "break symmetry" between different units. If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters.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

–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.