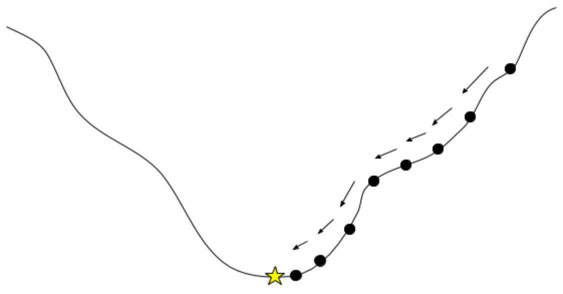
CS 189 Introduction to Machine Learning
Fall 2017 **Note 12**

1 Gradient Descent

Gradient descent is an iterative algorithm for finding local minima of differentiable functions. For an analogy, imagine walking downhill surrounded a thick fog that prevents you from seeing the height of the land around you, other than being able to tell which direction is steepest.



Recall that the gradient of f at x, denoted $\nabla f(x)$, points in the direction of steepest ascent. Conversely, the negative gradient points in the direction of steepest descent. Therefore, if we take a small step in the direction of the negative gradient, we will decrease the value of the function.

The update performed is

$$\vec{x}_{i+1} = \vec{x}_i - \alpha_i \nabla f(\vec{x}_i)$$

where $\alpha_i > 0$ is the step size. We may choose α_i to be a fixed constant, but in many cases it is decayed to zero over the course of training.

2 Neural Networks

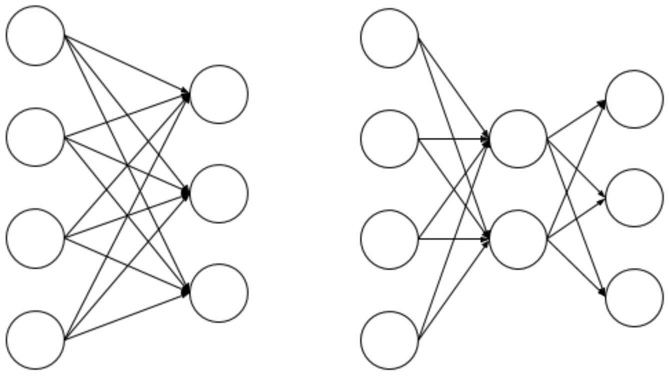
Neural networks are a class of compositional function approximators. Unlike other function approximators we have seen (e.g. polynomials), they are nonlinear in their parameters.

(Aside) In information processing we have several perspectives:

- Procedural perspective – thinking in terms of an imperative programming language
- Functional perspective – mathematical equations and reasoning
- Circuit/graph perspective – information is processed as it flows through the system

We will consider neural nets from the circuit/graph perspective.

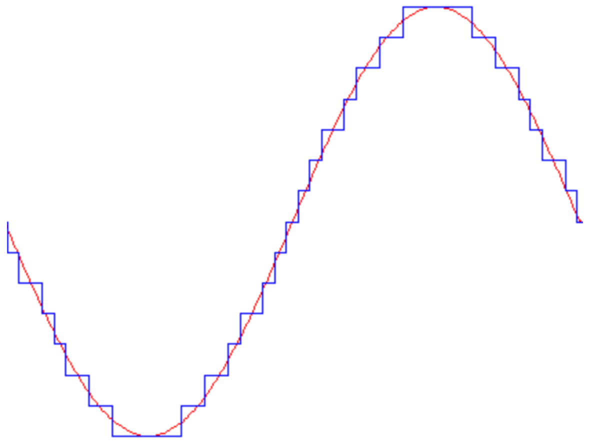
Consider the circuits below:



These circuits represent matrix multiplications, with the weights on the edges being the entries of the matrix. Assuming that the flow of information is from left to right, the circuit on the left is multiplication by a 3×4 matrix, and the circuit on the right is multiplication by a 2×4 matrix followed by multiplication by a 3×2 matrix.

Are these two circuits equally expressive? Certainly not, as the one on the left has rank at most 2, while the one on the right may have rank 3. However, the one on the left has more layers of processing, so it seems like it should be more expressive. The key thing that is missing is *nonlinearity*.

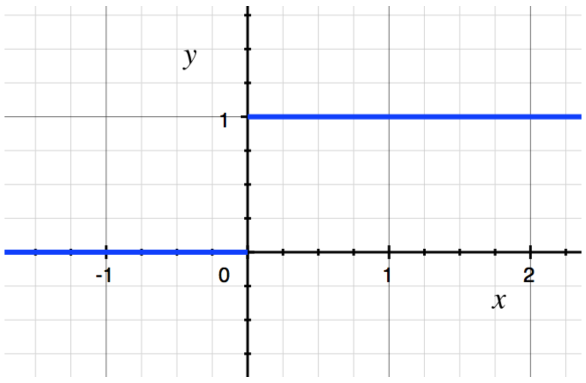
Let's insert a nonlinear function, called an *activation function*, after these linear computations. We would like to choose this activation function to make the circuit a universal function approximator.[1] A key observation is that piecewise-constant functions are universal function approximators:



The nonlinearity we use, then, is the step function:

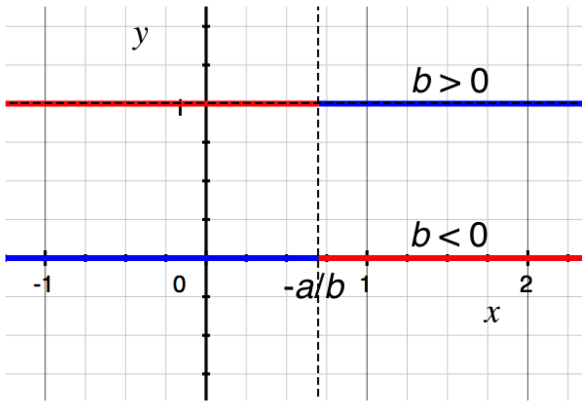
$$u(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

[1] This essentially means that given any continuous function, we can choose the weights such that the output of the circuit can be made arbitrarily close to the output of the given function for all inputs.



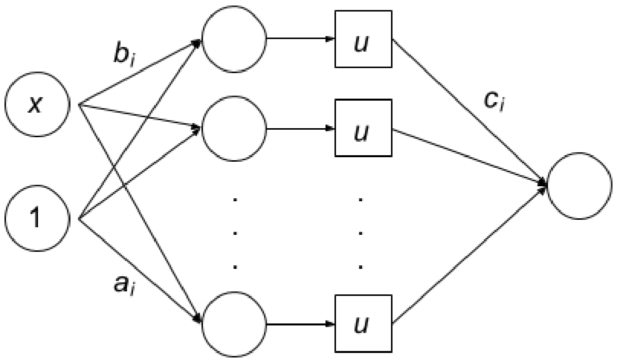
We can build very complicated functions from this simple step function by combining translated and scaled versions of it. Observe that

- If $a, b \in \mathbb{R}$, the function $x \mapsto u(a + bx)$ is a translated (and, depending on the sign of b, possibly flipped) version of the step function:



- If $c \neq 0$, the function $x \mapsto cu(x)$ is a vertically scaled version of the step function.

Assume that our circuit has the following structure:



The input x is one-dimensional, and the weight on x to neuron i is b_i. We also introduce a constant 1, whose weight a_i into neuron i is a_i. (This is referred to as the *bias*, but it has nothing to do with bias in the sense of the bias-variance tradeoff. It's just there to provide the network with the ability to shift the function.) The outputs of the intermediate nodes are $a_i + b_i x$, and then we pass each of these through the activation function u. The output of the network is a linear combination of the outputs of the activation functions:

$$h(x) = \sum_{i=1}^{d} c_i u(a_i + b_i x)$$

where d is the number of intermediate neurons.

2.1 Choosing weights

With a proper choice of a_i, b_i, and c_i, this function can approximate any continuous function we want. But the question remains: given some target function, how do we choose these parameters in the appropriate way?

Let's try a familiar technique: least squares. Assume we have training data $\{(x_k, y_k)\}_{k=1}^{n}$. We aim to solve the optimization problem

$$\min_{\vec{a},\vec{b},\vec{c}} \underbrace{\sum_{k=1}^{n} e_k}_{f(\vec{a},\vec{b},\vec{c})}$$

where

$$e_k = (y_k - h(x_k))^2$$

To run gradient descent, we need derivatives of the loss with respect to our optimization variables. We compute via the chain rule

$$\frac{\partial f}{\partial c_i} = \sum_{k=1}^{n} -2(y_k - h(x_k)) \underbrace{\frac{\partial h}{\partial c_i}(x_k)}_{=u(a_i+b_i x_k)}$$

We see that if this particular step is "off", as in $u(a_i + b_i x_k) = 0$, then

$$\frac{\partial e_k}{\partial c_i} = 0$$

so no update will be made for that example. More notably, consider the derivative with respect to a_i:

$$\frac{\partial f}{\partial a_i} = \sum_{k=1}^{n} -2(y_k - h(x_k)) \underbrace{\frac{\partial h}{\partial a_i}(x_k)}_{0}$$

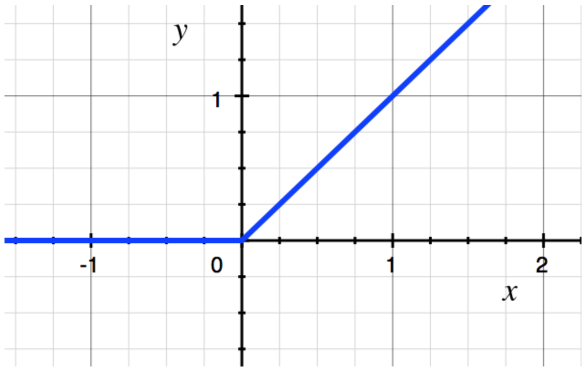
Similarly,

$$\frac{\partial f}{\partial b_i} = 0$$

The derivative at the jump is undefined, but in practice we will never hit that point of discontinuity. The bigger issue is that gradient descent will do nothing to optimize the a_i and b_i weights. Even though the step function is useful for the purpose of showing the approximation capabilities of neural networks, it is seldom used in practice because it cannot be trained by conventional gradient-based methods.

The next simplest universal approximator is the class of **piecewise-linear functions**. Just as piecewise-constant functions can be achieved by combinations of the step function as a nonlinearity, piecewise-linear functions can be achieved by combinations of the *rectified linear unit* (ReLU) function

$$u(x) = \max(0, x)$$



Now we can calculate the gradients:

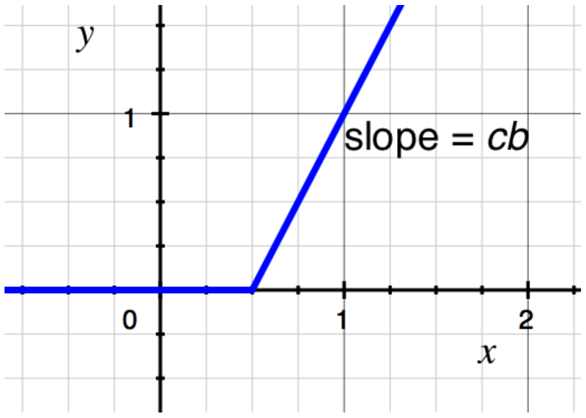
$$\frac{\partial f}{\partial c_i} = \sum_{k=1}^{n} -2(y_k - h(x_k))\max(0, a_i + b_i x)$$

$$\frac{\partial f}{\partial a_i} = \sum_{k=1}^{n} -2(y_k - h(x_k))c_i \frac{\partial}{\partial a_i}\max(0, a_i + b_i x) = \sum_{k=1}^{n} -2(y_k - h(x_k))c_i \left(\begin{cases} 0 & \text{if } a_i + b_i x < 0 \\ 1 & \text{if } a_i + b_i x > 0 \end{cases}\right)$$

$$\frac{\partial f}{\partial b_i} = \sum_{k=1}^{n} -2(y_k - h(x_k))c_i \frac{\partial}{\partial b_i}\max(0, a_i + b_i x) = \sum_{k=1}^{n} -2(y_k - h(x_k))c_i \left(\begin{cases} 0 & \text{if } a_i + b_i x < 0 \\ x_i & \text{otherwise} \end{cases}\right)$$

Crucially, we see that the gradient with respect to $\vec{a}$ and $\vec{b}$ is not uniformly zero, unlike with the step function.

If the ReLU is active, both weights are adjustable. Depending on the gradient of the objective function, our ReLUs can move to the left or right, increase or decrease their slope, and flip direction.



If the weight initialization turns off the ReLU for every training point, then the gradient descent updates will not change the parameters of the neuron, and we say it is dead. Random initialization should result in a reasonable number of active neurons. There are more sophisticated initialization methods, such as “Glorot” initialization[2], which take into account the number of connections and are more effective in practice. Leaky ReLUs, which have a small slope in the section where the ReLU is flat (say, $u(x) = .01x$ when $x < 0$) are sometimes used to provide some small gradient signal to avoid dead neurons.

2.2 Neural networks are universal function approximators

The celebrated neural network universal approximation theorem, due to Kurt Hornik[3], tells us that neural networks are universal function approximators in the following sense.

Theorem. Suppose $u : \mathbb{R} \to \mathbb{R}$ is nonconstant, bounded, nondecreasing, and continuous[4], and let $S \subseteq \mathbb{R}^n$ be closed and bounded. Then for any continuous function $f : S \to \mathbb{R}$ and any $\varepsilon > 0$, there exists a neural network with one hidden layer and a finite number of neurons, which we can write

$$h(\vec{x}) = \sum_{i=1}^{d} c_i u(a_i + \vec{b}_i^\top \vec{x})$$

such that

$$|h(\vec{x}) - f(\vec{x})| < \varepsilon$$

for all $\vec{x} \in S$.

There’s some subtlety in the theorem that’s worth noting. It says that for any given continuous function, there exists a neural network of finite size that uniformly approximates the given function. However, it says nothing about how well any particular architecture you’re considering will approximate the function. It also doesn’t tell us how to compute the weights.

It’s also worth pointing out that in the theorem, the network consists of just one hidden layer. In practice, people find that using more layers works better.

[2] See *Understanding the difficulty of training deep feedforward neural networks*.
[3] See *Approximation Capabilities of Multilayer Feedforward Networks*.
[4] Both ReLU and sigmoid satisfy these requirements.