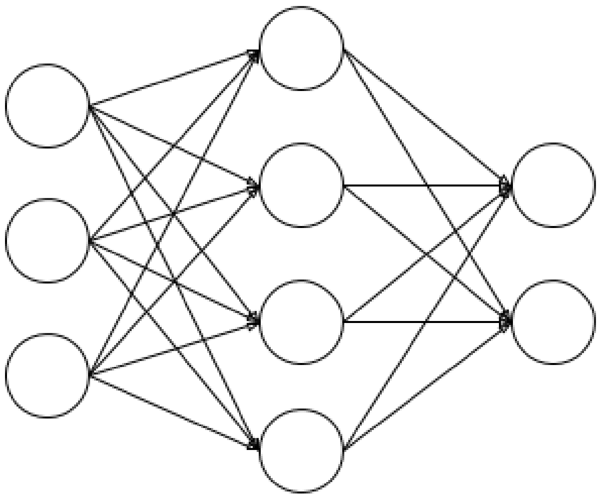
EECS 189 Introduction to Machine Learning
Fall 2020 Note 13

1 Neural Networks

Neural networks are a class of compositional function approximators. They come in a variety of shapes and sizes. In this class, we will only discuss **feedforward** neural networks, those networks whose computations can be modeled by a directed acyclic graph.[1] The most basic (but still commonly used) class of feedforward neural networks is the **multilayer perceptron**. Such a network might be drawn as follows:



Computation flows left-to-right. The circles represent **nodes**, a.k.a. **units** or **neurons**, which are loosely based on the behavior of actual neurons in the brain. Observe that the nodes are organized into **layers**. The first (left-most) layer is called the **input layer**, the last (right-most) layer is called the **output layer**, and any other layers (there is only one here, but there could be multiple) are referred to as **hidden layers**. The dimensionality of the input and output layers is determined by the function we want the network to compute. For a function from $\mathbb{R}^d$ to $\mathbb{R}^k$, we should have d input nodes and k output nodes. The number and sizes of the hidden layers are hyperparameters to be chosen by the network designer.

Note that in the diagram above, each non-input layer has the following property: every node in that layer is connected to every node in the previous layer. Layers that have this property are described as **fully connected**.[2] Each edge in the graph has an associated weight, which is the strength of the connection from the input node in one layer to the node in the next layer. Each node computes a weighted sum of its inputs, with these connection strengths being the weights, and then applies a

[1] There are also **recurrent** neural networks whose computation graphs have cycles.
[2] Later we will learn about **convolutional layers**, which have a different connectivity structure.

nonlinear function which is variously referred to as the **activation function** or the **nonlinearity**. Concretely, if $\mathbf{w}_i$ denotes the weights and σ_i denotes the activation function of node i, it computes the function

$$\mathbf{x} \mapsto \sigma_i(\mathbf{w}_i^\top \mathbf{x})$$

Let us denote the number of (non-input) layers by L, the number of units in layer $\ell \in \{0, \dots, L\}$ by n_ℓ (here n_0 is the size of the input layer), and the nonlinearity for layer $\ell \in \{1, \dots, L\}$ by $\boldsymbol{\sigma}_\ell : \mathbb{R}^{n_\ell} \to \mathbb{R}^{n_\ell}$. The weights for every node in layer ℓ can be stacked (as rows) into a matrix of weights $\mathbf{W}_\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$. Then layer ℓ performs the computation

$$\mathbf{x} \mapsto \boldsymbol{\sigma}_\ell(\mathbf{W}_\ell \mathbf{x})$$

Since the output of each layer is passed as input to the next layer, the function represented by the entire network can be written

$$\mathbf{x} \mapsto \boldsymbol{\sigma}_L(\mathbf{W}_L \sigma_{L-1}(\cdots \boldsymbol{\sigma}_2(\mathbf{W}_2 \boldsymbol{\sigma}_1(\mathbf{W}_1 \mathbf{x})) \cdots))$$

This is what we mean when we describe neural networks as *compositional*.

Note that in most layers, the nonlinearity will be the same for each node within that layer, so it makes sense to refer to a scalar function $\sigma_\ell : \mathbb{R} \to \mathbb{R}$ as "the" nonlinearity for layer ℓ, and apply it element-wise:

$$\boldsymbol{\sigma}_\ell(\mathbf{x}) = \begin{bmatrix} \sigma_\ell(x_1) \\ \vdots \\ \sigma_\ell(x_{n_\ell}) \end{bmatrix}$$

The principle exception here is the **softmax** function $\boldsymbol{\sigma} : \mathbb{R}^k \to \mathbb{R}^k$ defined by

$$\boldsymbol{\sigma}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^k e^{x_j}}$$

which is often used to produce a discrete probability distribution over k classes. Note that every entry of the softmax output depends on every entry of the input. Also, softmax preserves ordering, in the sense that sorting the indices $i = 1, \dots, k$ by the resulting value $\boldsymbol{\sigma}(\mathbf{x})_i$ yields the same ordering as sorting by the input value x_i. In other words, more positive x_i leads to larger $\boldsymbol{\sigma}(\mathbf{x})_i$. This nonlinearity is used most commonly (but not always) at the output layer of the network.

1.1 Expressive power

It is the repeated combination of nonlinearities that gives deep neural networks their remarkable expressive power. Consider what happens when we remove the activation functions (or equivalently, set them to the identity function): the function computed by the network is

$$\mathbf{x} \mapsto \underbrace{\mathbf{W}_L \mathbf{W}_{L-1} \cdots \mathbf{W}_2 \mathbf{W}_1}_{=:\widetilde{\mathbf{W}}} \mathbf{x}$$

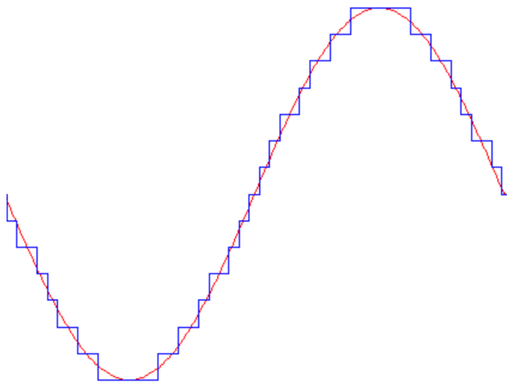
which is linear in its input! Moreover, the size of the smallest layer restricts the rank of $\widetilde{\mathbf{W}}$, as

$$\operatorname{rank}(\widetilde{\mathbf{W}}) \le \min_{\ell \in \{1,\dots,L\}} \operatorname{rank}(\mathbf{W}_\ell) \le \min_{\ell \in \{0,\dots,L\}} n_\ell$$

Despite having many layers of computation, this class of networks is not very expressive; it can only represent linear functions.

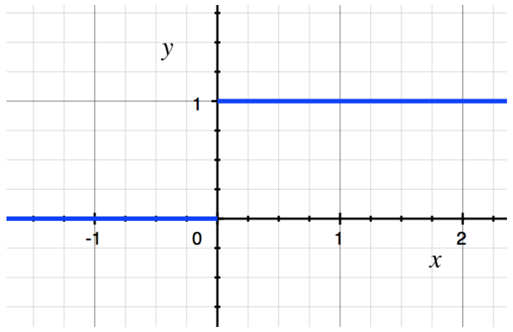
We would like to produce a class of networks that are **universal function approximators**. This essentially means that given any continuous function, we can choose a network in this class such that the output of the circuit can be made arbitrarily close to the output of the given function for all given inputs. We make a more precise statement later.

A key observation is that piecewise-constant functions are universal function approximators:



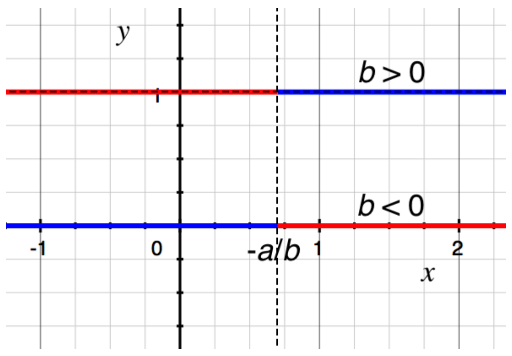
The nonlinearity we use, then, is the step function:

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



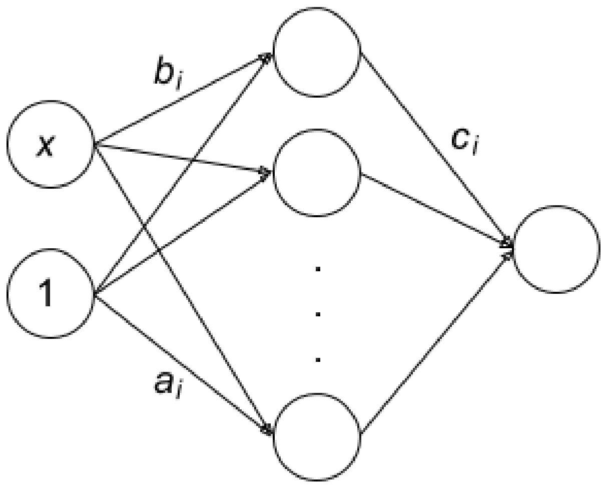
We can build very complicated functions from this simple step function by combining translated and scaled versions of it. Observe that

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



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

It turns out that only one hidden layer is needed for universal approximation, and for simplicity we assume a one-dimensional input. Thus from here on we consider networks with the following structure:



The input x is one-dimensional, and the weight on x to node j is b_j. We also introduce a constant 1, whose weight into node j is a_j. (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 node with the ability to shift its input.) The function implemented by the network is

$$h(x) = \sum_{j=1}^{k} c_j \sigma(a_j + b_j x)$$

where k is the number of hidden units.

1.2 Choosing weights

With a proper choice of a_j, b_j, and c_j, 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 $\{(x_i, y_i)\}_{i=1}^n$. We aim to solve the optimization problem

$$\min_{\mathbf{a},\mathbf{b},\mathbf{c}} \underbrace{\sum_{i=1}^{n}(y_i - h(x_i))^2}_{f(\mathbf{a},\mathbf{b},\mathbf{c})}$$

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

$$\frac{\partial (y_i - h(x_i))^2}{\partial c_j} = -2(y_i - h(x_i))\frac{\partial h(x_i)}{\partial c_j} = 2(y_i - h(x_i))\sigma(a_j + b_j x_i)$$

We see that if this particular step is "off", as in $\sigma(a_j + b_j x_i) = 0$, then

$$\frac{\partial (y_i - h(x_i))^2}{\partial c_j} = 2(y_i - h(x_i))\underbrace{\sigma(a_j + b_j x_i)}_{0} = 0$$

so no update will be made for that example. More egregiously, consider the derivatives with respect to a_j[3]:

$$\frac{\partial f}{\partial a_j} = \sum_{i=1}^{n} -2(y_i - h(x_i))\underbrace{\frac{\partial h(x_i)}{\partial a_j}}_{0} = 0$$

and b_j:

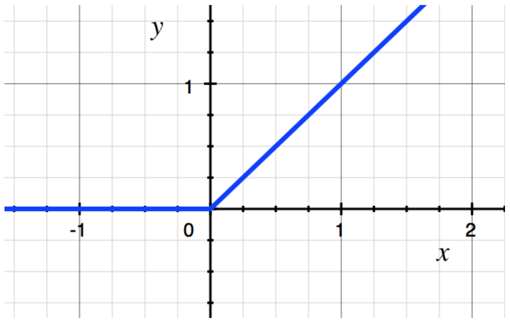
$$\frac{\partial f}{\partial b_j} = \sum_{i=1}^{n} -2(y_i - h(x_i))\underbrace{\frac{\partial h(x_i)}{\partial b_j}}_{0} = 0$$

Since gradient descent changes weights in proportion to their gradient, it will never modify **a** or **b**! 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.

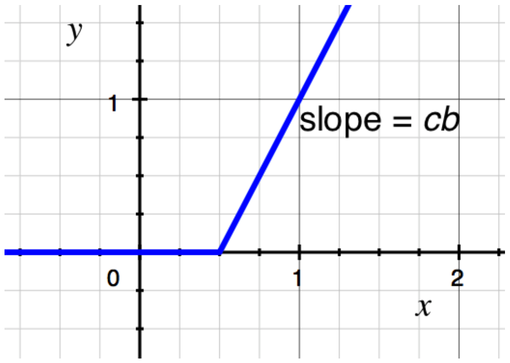
The next simplest universal approximator is the class of **piecewise-linear functions**. Just as piecewise-constant functions can be achieved by combinations of the step function as a nonlinearity, piecewise-linear functions can be achieved by combinations of the *rectified linear unit* (ReLU) function

$$\sigma(x) = \max\{0, x\}$$

[3] Technically, the derivative of σ is not defined at zero, where there is a discontinuity. However it is defined (and zero) everywhere else. In practice, we will almost never hit the point of discontinuity because it is a set of measure zero.



Depending on the weights **a** and **b**, our ReLUs can move to the left or right, increase or decrease their slope, and flip direction.



Let us calculate the gradients again, assuming we replace the step functions by ReLUs:

$$\frac{\partial f}{\partial c_j} = \sum_{i=1}^{n} -2(y_i - h(x_i)) \max\{0, a_j + b_j x_i\}$$

$$\frac{\partial f}{\partial a_j} = \sum_{i=1}^{n} -2(y_i - h(x_i)) c_j \frac{\partial}{\partial a_j} \max\{0, a_j + b_j x_i\} = \sum_{k=1}^{n} -2(y_i - h(x_i)) c_j \left(\begin{cases} 0 & \text{if } a_j + b_j x_i < 0 \\ 1 & \text{if } a_j + b_j x_i > 0 \end{cases} \right)$$

$$\frac{\partial f}{\partial b_j} = \sum_{i=1}^{n} -2(y_i - h(x_i)) c_j \frac{\partial}{\partial b_j} \max\{0, a_j + b_j x_i\} = \sum_{i=1}^{n} -2(y_i - h(x_i)) c_j \left(\begin{cases} 0 & \text{if } a_j + b_j x_i < 0 \\ x_i & \text{if } a_j + b_j x_i > 0 \end{cases} \right)$$

Crucially, we see that the gradient with respect to **a** and **b** is not uniformly zero, unlike with the step function.

Later we will discuss **backpropagation**, a dynamic programming algorithm for efficiently computing gradients with respect to a neural network's parameters.

1.3 Neural networks are universal function approximators

The celebrated neural network universal approximation theorem, due to Kurt Hornik[4], tells us that neural networks are universal function approximators in the following sense.

[4] See *Approximation Capabilities of Multilayer Feedforward Networks*.

Theorem. Suppose $\sigma : \mathbb{R} \to \mathbb{R}$ is nonconstant, bounded, nondecreasing, and continuous[5], and let $S \subseteq \mathbb{R}^d$ be closed and bounded. Then for any continuous function $f : S \to \mathbb{R}$ and any $\epsilon > 0$, there exists a neural network with one hidden layer containing finitely many nodes, which we can write

$$h(\mathbf{x}) = \sum_{j=1}^{k} c_j \sigma(a_j + \mathbf{b}_j^\top \mathbf{x})$$

such that

$$|h(\mathbf{x}) - f(\mathbf{x})| < \epsilon$$

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

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.

[5] Both ReLU and sigmoid satisfy these requirements.