02-251: Great Ideas in Computational Biology
Notes on Neural Networks

Phillip Compeau

Spring 2019

1. The Universality of Binary Functions with Neural Networks

Last time, we saw that some binary operations, such as AND and OR, can be replicated with a single perceptron. For example, given two binary inputs x_1 and x_2, a perceptron can produce x_1 AND x_2 if we set $w_1 = 1$, $w_2 = 1$, and $\theta = 2$. The perceptron will fire (output 1) only if $w_1 \cdot x_1 + w_2 \cdot x_2 = x_1 + x_2 \geq \theta = 2$, which occurs when x_1 and x_2 are both 1 (Figure 1 (left)).

Similarly, we can obtain x_1 OR x_2 if we set $w_1 = 1$, $w_2 = 1$, and $\theta = 1$. The perceptron will only fire when $x_1 + x_2 \geq \theta = 1$, which occurs when at least one of x_1 and x_2 is 1 (Figure 1 (right)).

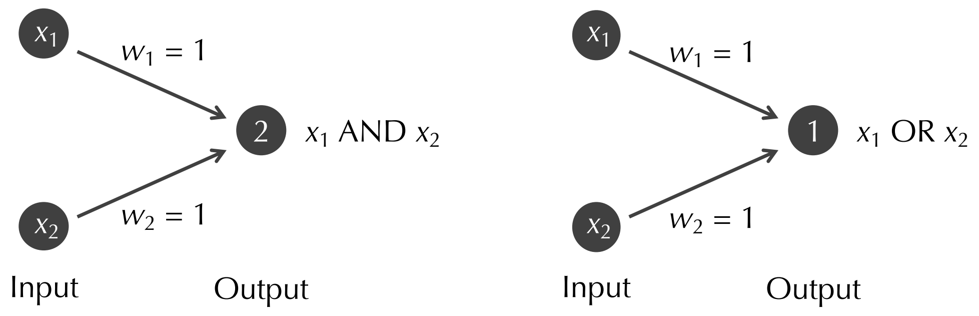

Figure 1: Perceptrons computing the binary operations AND (left) and OR (right). We draw these perceptrons so that the inputs are shown in the nodes on the left and are both connected to a perceptron on the right, whose output is the desired operation. For each of the two perceptrons, the weight w_i is assigned to the edge connecting x_i to the perceptron, and the threshold value θ labels the node corresponding to the perceptron.

We also saw that we can generalize these two functions to perform AND and OR on an arbitrary number of n binary inputs x_i by setting all weights w_i equal to 1 and θ equal to n and 1, respectively.

However, we struggled with replicating the exclusive or (XOR) of two binary variables with a perceptron and saw that this is because it wasn't linearly separable.

We will be able to compute XOR if we *add* a neuron. The intuition for this is that binary functions can be built using other binary functions as building blocks. In this case, x_1 XOR x_2 is equivalent to (x_1 OR x_2) AND (!x_1 OR !x_2). (Here, ! denotes "not", so that !x_1 is 1 when x_1 is 0 and !x_1 is 0 when x_1

is 1.

Exercise: What perceptron might we use to implement $!x_1$ OR $!x_2$?

To replicate $!x_1$ OR $!x_2$, we want the neuron to fire when $x_1 + x_2 \leq 1$, or when $-x_1 - x_2 \geq -1$. So the neuron we are looking for has $w_1 = -1$, $w_2 = -1$, and $\theta = -1$.

Exercise: We then want to take the output of the neurons x_1 OR x_2 along with $!x_1$ OR $!x_2$ and combine them via an AND. How might we do this?

The answer to the preceding exercise is to pass the same inputs (x_1 and x_2) into the two separate perceptrons, and then *combine* these outputs into a new AND perceptron that will then produce the desired output (Figure 2).

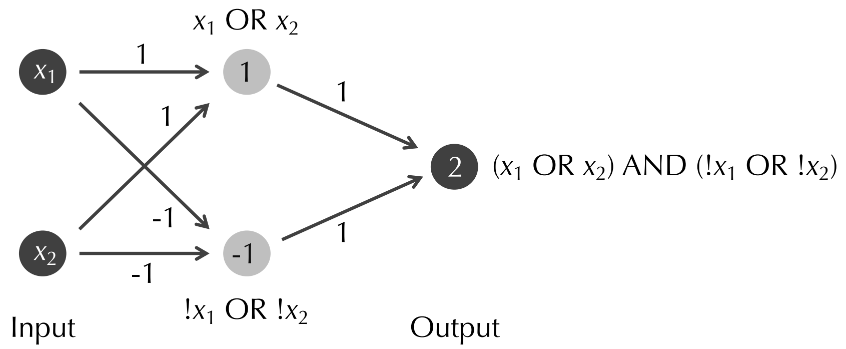

Figure 2: A one-layer neural network of perceptrons that computes the function (x_1 OR x_2) AND ($!x_1$ OR $!x_2$), which is equivalent to x_1 XOR x_2. The first layer consists of two perceptrons computing each of x_1 OR x_2 and $!x_1$ OR $!x_2$. The output perceptron then performs the AND operation of the results.

This example provides a simple depiction of the power of combining neurons into a simple form of a **neural network**, in which neuron outputs get fed as inputs into other neurons. Figure 2 illustrates a "one-layer" neural network, meaning that there is a "hidden" layer of neurons that are neither the input variables nor the output neurons. We henceforth use gray to denote hidden layers and black to denote input variables and output neurons.

Furthermore, it is not difficult to see now that we can use the AND and OR perceptrons as building blocks to produce any **binary function**, i.e., a function that takes binary variables x_i as input and returns 1 or 0. The other way of saying this is that binary functions are **universal** for neural networks of perceptrons with binary input variables.

Yet as we will see in the next two sections, neural networks become much more powerful when we allow their inputs and outputs to be arbitrary real numbers. For example, we will next demonstrate that not just binary functions, but *any continuous function* can be represented by a very simple one-layer neural network of perceptrons if we generalize these perceptrons to take any real input x and to produce a more varied output than just 1 or 0.

2. Universality of Neural Networks

Before beginning, we should review some mathematics that you may have seen but likely haven't seen applied in a practical context.

We say that a function $f : \mathbb{R} \to \mathbb{R}$ is **continuous** at a point c if $f(c)$ exists and for every $\epsilon > 0$, there exists some $\delta > 0$ such that whenever $|x - c| < \delta$, $|f(x) - f(c)| < \epsilon$. A visualization of continuity at a point is given in Figure 3. A function is continuous on an open interval (including all of $\mathbb{R}$) if it is continuous at every point in that interval.

We will consider functions that are continuous on closed intervals.[1]. In part, this is because of the following fundamental result, which dictates that continuous functions on closed intervals cannot head toward infinity. (Contrast with a function like $f(x) = 1/(x-4)$ on the interval $(0, 4)$.). We present this theorem without proof, which is beyond the scope of the class.

Extreme Value Theorem. *Any continuous function $f : [a, b] \to \mathbb{R}$ is bounded on this interval. In particular, there are some values x_1 and x_2 in the interval such that $f(x_1) \leq f(x) \leq f(x_2)$ for all $x \in [a, b]$.*

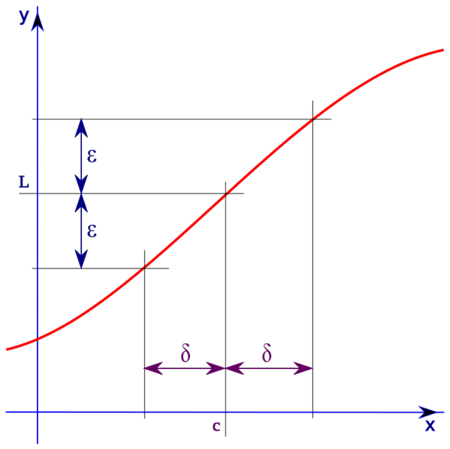

Figure 3: When a function is continuous at point c, for any choice of $\epsilon > 0$ we can find a $\delta > 0$ such that the function "fits inside" the rectangle of width 2δ and height 2ϵ centered at c. Image courtesy Hi:Te, Wikimedia Commons user.

When we say that we can replicate any function with a neural network, what we mean is that we can perform better and better approximations of a function $f : [a, b] \to \mathbb{R}$ by the output of one-layer finite neural networks (whose general form is shown in Figure 4).

Before getting ahead of ourselves, we should define what we mean by approximating a function using some other function. A function $f : \mathbb{R} \to \mathbb{R}$ is **ϵ-approximable** by the output $O(x)$ of another function with one input variable over an interval $[a, b]$ if for every $x \in [a, b]$, $|f(x) - O(x)| < \epsilon$. Furthermore, given a family $\mathcal{O} = O_1, O_2, \ldots, \ldots$ of functions, we say that f is **approximable** by $\mathcal{O}$ if for any $\epsilon > 0$, there exists some function $O_i \in \mathcal{O}$ such that f is ϵ-approximable by O_i.

Finally, the output of a neuron as a function of its input is called an **activation function**. Until now we have used the **binary activation function** in which the output of a neuron is 1 if the

[1]The definition of continuity on a closed interval $[a, b]$ is messier because it requires us to deal with the fact that the set of points such that $|x - a| < \delta$ or $|x - b| < \delta$ do not all lie within the interval for any δ; however, this case can be handled in turn.

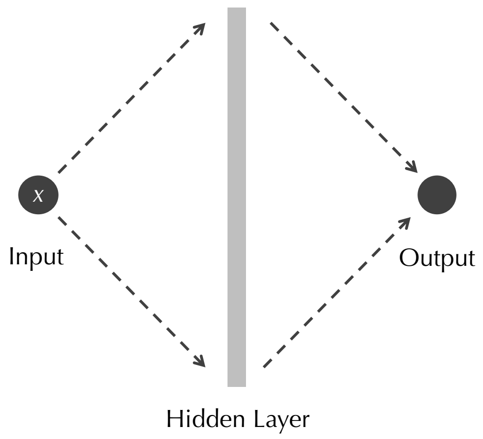

Figure 4: A depiction of a generalized one-layer neural network with a single input variable and a single output perceptron. Every continuous function on a closed interval can be approximated by a neural network of this form.

input is at least some threshold θ and 0 otherwise. We are going to also use an **identity activation function** (denoted id_θ) in which the output of the neuron is equal to its input whenever this input exceeds the threshold:

$$\mathrm{id}_\theta(x) = \begin{cases} x & \text{for } x \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

There are many activation functions used, but the assumption is typically that the same activation function is used for every node of a neural network. We will relax this assumption very slightly and consider 1-layer neural networks whose hidden layer has binary activation functions and whose output layer uses the identity activation function.

We now know enough to state the main result of the section.

Universality Approximation Theorem. *Any continuous function $f : \mathbb{R} \to \mathbb{R}$ is approximable by the outputs of a family of finite 1-layer neural networks.*

The rest of this section will consist of a proof of the above statement, which will proceed in two main thrusts. First, we will show that every continuous function is approximable by a family of very simple functions called "step functions". Second, we will show that we can produce step functions as outputs of neural networks. The combination of these two facts, stated below as theorems, will yield the result.

Theorem 1. *Every continuous function $f : [a, b] \to \mathbb{R}$ is approximable by a family of step functions.*

Theorem 2. *Any step function is the output of some 1-layer neural network.*

A **tower function** is a function $T_{c,d,k} : \mathbb{R} \to \mathbb{R}$ that is defined as follows given constants c, d, and k (where $c \leq d$).

$$T_{c,d,k}(x) = \begin{cases} k & \text{for } c \leq x < d \\ 0 & \text{otherwise} \end{cases}$$

Next, we define a **partition** of an interval $[a, b]$ as an increasing finite sequence of real numbers $\mathbf{c} = (c_1, \ldots, c_{n+1})$ taken from the interval.

A **step function** is a sum of disjoint tower functions. More precisely, it is a function $S_c : [a, b] \to \mathbb{R}$ defined by a partition $\mathbf{c} = (c_1, \ldots, c_{n+1})$ of $[a, b]$ and a collection of parameters $\mathbf{k} = (k_1, \ldots, k_n)$ such that

$$S_{\mathbf{c},\mathbf{k}}(x) = T_{c_1,c_2,k_1}(x) + T_{c_2,c_3,k_2}(x) + \cdots + T_{c_n,c_{n+1},k_n}(x)$$

and also $S_{\mathbf{c},\mathbf{k}}(b) = k_n$. In other words, if a number x lies between c_i and c_{i+1}, then $S_{\mathbf{c},\mathbf{k}}(x) = k_i$.

Before proving Theorem 1, we need one additional fact about continuous functions on a closed interval. A real-valued function is **uniformly continuous** if for any $\epsilon > 0$ there exists some $\delta > 0$ such that for *every* c, if $|x - c| < \delta$, then $|f(x) - f(c)| < \epsilon$. This is a stronger condition than continuity on an interval, but it must be true for all continuous functions on a closed interval.

Lemma 1. *Any continuous function $f : \mathbb{R} \to \mathbb{R}$ is uniformly continuous.*

Exercise: Provide a function that is continuous on an open interval but not uniformly continuous. Justify your answer.

We are now ready to prove Theorem 1.

Proof. Let $\epsilon > 0$ be given. Because f is continuous on a closed interval, it must be uniformly continuous by Lemma 1, and so we know that there must be some $\delta > 0$ such that whenever $|x_1 - x_2| < \delta$ for points $x_1, x_2 \in [a, b]$, $|f(x_1) - f(x_2)| < \epsilon$.

Let $n = \lceil \frac{b-a}{\delta} \rceil$ and consider the partition $c = (c_1, \ldots, c_{n+1})$ of $[a, b]$ defined as follows. For $\gamma = \frac{b-a}{n}$, set $c_i = a + (i - 1)\gamma$. Note that $c_0 = a$, $c_{n+1} = b$, and that $c_{i+1} - c_i = \gamma \leq \delta$.

Now let k_i be defined as the function value of the midpoints $m_i = \dfrac{c_i + c_{i+1}}{2}$ of the intervals $[c_i, c_{i+1}]$:

$$k_i = f(m_i)$$

Let $S_{\mathbf{c},\mathbf{k}} : [a, b] \to \mathbb{R}$ be the step function of $\mathbf{c}$ and $\mathbf{k} = (k_1, \ldots, k_n)$. For any point $x \in [c_i, c_{i+1}]$, $|x - m_i| \leq \gamma/2 < \delta$. So by the uniform continuity of f, we know that $|f(x) - f(m_i)| = |f(x) - k_i| = |f(x) - S_{c,k}(x)| < \epsilon$, which is what we set out to show.

□

Now that we have shown that continuous functions are approximable using step functions, we will turn to proving Theorem 2. First, before we try to use a neural network to compute a step function, can we use it to compute a single tower function?

Specifically, we want a one-layer neural network that takes a single input x and returns a single output that is equal to the tower function $T_{c,d,k}(x)$. This network is shown in Figure 5. The top neuron in the hidden layer fires 1 when $x \geq c$ and 0 otherwise; the bottom neuron fires 1 when $x \geq d$ and 0 otherwise.

We then assign weights of $+k$ to the output of the top neuron and $-k$ to the output of bottom neuron, so that the input to the output neuron will be $+k$ when $c \leq x < d$ and 0 otherwise. We can then assign a strict threshold value of k to this neuron so that if $c \leq x < d$ then the k will be passed through to the output; thus, the output will be $T_{c,d,k}(x)$ as desired.

Now the challenge is to build a step function. Say that we have the interval $[d, e]$ and want to consider the tower function $T_{d,e,k'}$. Then we can add two more input neurons as shown in Figure 6.

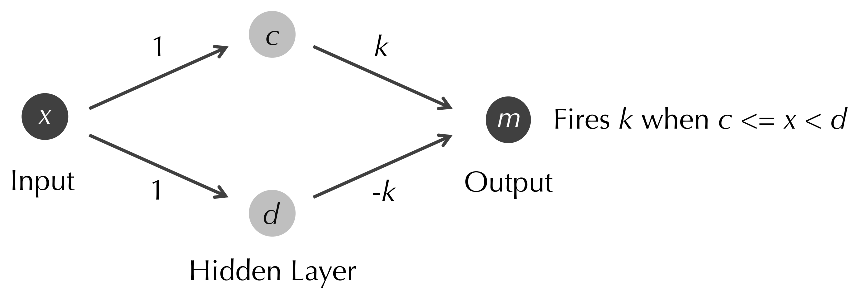

Figure 5: A one-layer neural network with two hidden neurons computing the tower function that returns k when $c \leq x < d$. We make the slight generalization to the previous model of neuron output and assume that the output neuron fires k instead of 1.

The output will be k when $c \leq x < d$ and k' when $d \leq x < e$, producing a step function with two "steps".

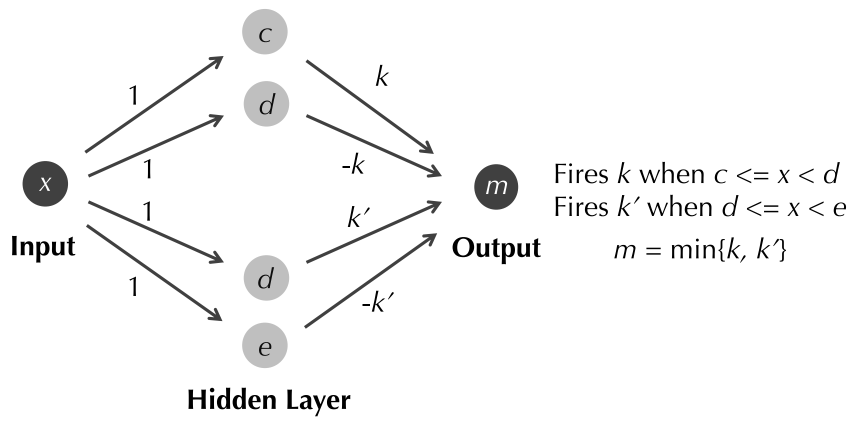

Figure 6: A one-layer neural network implementing a step function comprised of two tower functions for the adjacent intervals $[c, d]$ and $[d, e]$.

In this manner, we can continue adding two neurons at a time for any partition of $[a, b]$, thus building a neural network whose output is any finite step function. This proves Theorem 2 and therefore the Universality Approximation Theorem. $\square$

Before proceeding, a few remarks. First, the activation functions that we have considered thus far compare $\sum_{1 \leq i \leq n} w_i \cdot x_i$ and compare it against some threshold parameter θ. This leads to a discontinuous activation function with very simplistic output. In practice, we instead take the sum $\sum_{1 \leq i \leq n} w_i \cdot x_i - \theta$ and pass this linear function as input to an activation function α that is the same at every neuron. Perhaps the most common activation function is the **logistic function**,

$$\sigma(x) = \frac{1}{1 + e^{-x}}\,.$$

We make this remark because the Universality Approximation Theorem holds for *any* activation function that is continuous, bounded, and non-constant. The proof of Theorem 2 proceeds similarly, but with way uglier details. In particular, instead of replicating tower functions exactly, we build neural networks of the form shown in Figure 5 whose output *approximate* tower functions.

It is beyond the scope of the course, but we can show that this theorem generalizes to the case in which we desire f to take multi-dimensional input and return multi-dimensional output, as illustrated in Figure 7.

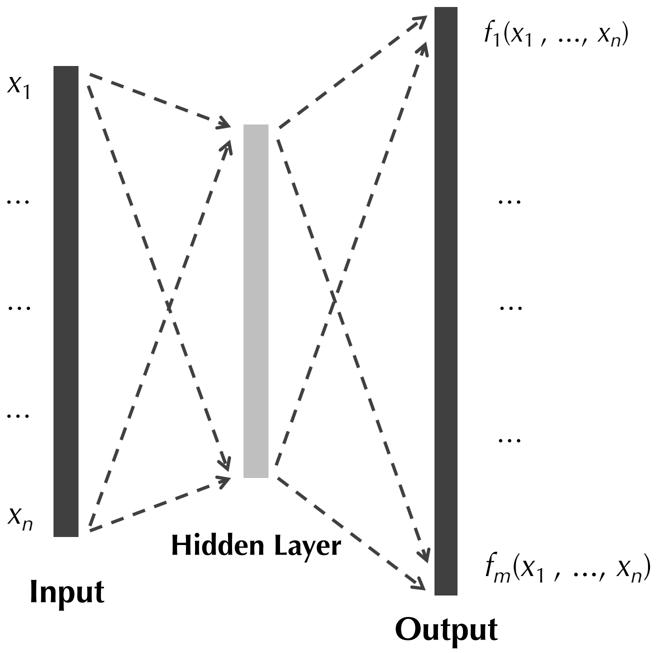

Figure 7: A depiction of a one-layer neural network in which inputs $\mathbf{x}$ are n-dimensional vectors $(x_1, \ldots, x_n)$ and we wish to approximate an m-dimensional function $f(\mathbf{x})$ whose values can therefore be represented as a vector of m separate functions $(f_1(\mathbf{x}), \ldots, f_m(\mathbf{x}))$. Dashed edges indicate that every node in one layer is connected (via a weighted edge) to every node in the next layer.

Finally, we note that the Universality Approximation Theorem is important because just about every question we can ask in computer science relates to a problem of writing an algorithm to determine some desired output(s) on a collection of input(s). If a neural network can approximate any continuous function, then it is a very powerful object. In the next section, we turn neural networks toward the problem of classification.