inzva DLSG

A Fancy Method for Function Approximation

Week1

Contents

1 Introduction

In this chapter, we finally introduce a simple neural network. We first define it in a mathematical sense. Then we will support our claims using illustration of them. We begin by defining an artificial neuron. Then we will define something called 'hidden layer'. It will contain stack of neurons inside. Then we will introduce algorithms called forward propagation, backward propagation and activations functions.

2 Fully Connected Layers

2.1 An Artificial Neuron

Let us define an artificial neuron. Assume we have input $\mathbf{x} \in \mathbb{R}^n$. Then we can define a function

$$f(x_1, ..., x_n) = \sum_{i=1}^{n} w_i x_i \tag{1}$$

where $w_i \in \mathbb{R}^n$ are learnable parameters and x_i's are indices of vector $\mathbf{x}$. We will investigate how we learn w_i's later but as now you can just imagine them as a real coefficients. Here f is a function that takes every indices of $\mathbf{x}$ as an input and outputs linear combination of the inputs. Then f is a function from n-dimensional space to one-dimensional space. If w_i's are 0, then our function will output zero. We can prevent this by adding a 'bias' term, which is crucial in machine learning. This bias term allows the function to shift the output, enabling the neural network to learn and approximate more complex patterns effectively. We redefine artificial neuron as:

$$f(x_1, ..., x_n) = \sum_{i=1}^{n} w_i x_i + b \tag{2}$$

where $b \in \mathbb{R}$. If we are lucky, we will have no problem with the artificial neuron we defined. Thus, an artificial neuron is a function f that takes an n-dimensional input $\mathbf{x}$, maps it to a real number and is in the form of (2). If we face any problems in the future, we can change the definition. For now, it seems fine. In Figure 1, we illustrate a single neuron with n input variables. This illustration will help us understand the topology of the network better. For a short period, we will omit the bias variable for educational purposes. Don't forget, it's still there.

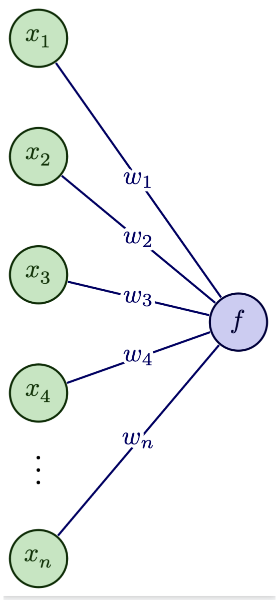


Figure 1: Hidden layer with one neuron

2.1.1 Defining Fully Connected Layer Inductively

Now, we will define what is called hidden layer. To do this we define another functions and we write them as linear system which you might be familiar if you take linear algebra class.

$$\begin{aligned} f_1(x_1, ..., x_n) &= \sum_{i=1}^{n} w_{1,i}x_i \\ f_2(x_1, ..., x_n) &= \sum_{i=1}^{n} w_{2,i}x_i \end{aligned} \tag{3}$$

Corresponding illustration of the system (3) would be like in Figure 2

Accordingly, we can generalize this induction to m neurons. We will stack m neuron in a layer. Look at the Figure 3 for the illustration. Thus, we will have corresponding linear system:

$$\begin{aligned} f_1 &= \sum_{i=1}^{n} w_{1,i}x_i \\ f_2 &= \sum_{i=1}^{n} w_{2,i}x_i \\ &\vdots \\ f_m &= \sum_{i=1}^{n} w_{m,i}x_i \end{aligned} \tag{4}$$

Then we would be finished with defining a layer. We just need a last part for the neural network, output layer. We will introduce the output layer in the same manner. From now on, when we write $w_{i,j}^{(n)}$ we mean the weights in the *nth* layer. For example, $w_{i,j}^{(1)}$ will be the weights between input layer and first hidden layer.

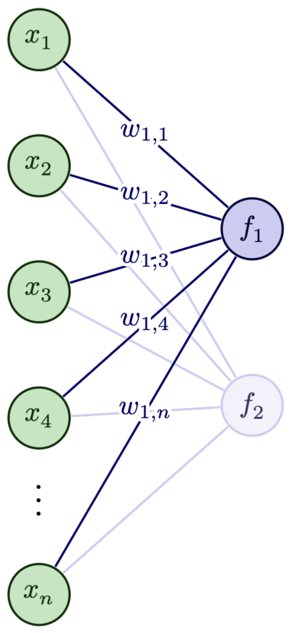


Figure 2: Hidden layer with two neurons

Then the mathematical equation that represents Figure 4 will be the following:

$$
\begin{aligned}
f_j &= \sum_{i=1}^{n} w_{j,i}^{(1)} x_i \\
and \\
\hat{y} &= \sum_{i=1}^{m} w_{1,i}^{(2)} f_i
\end{aligned}
\tag{5}
$$

Let us gather what we have using the last neural network we just defined. We have an input $\mathbf{x} = (x_1, x_2, ..., x_n)$ which serves as an **input layer**. We have $f_i, i = 1, 2, ..., m$ and computed by $f_j = \sum_{i=1}^{n} w_{j,i}^{(1)} x_i$ which serves as a **hidden layer**. We also have $\hat{y} = \sum_{i=1}^{m} w_{1,i}^{(2)} f_i$ which serves as an **output layer**. This neural network is called **fully connected neural network**. It has 2 layers. We do not count the input layer conventionally. Since we know the basics of linear algebra and what we have here is just a bunch of linear systems we can write the neural network in the matrix form. We will use the following notation hereupon:

- $\mathbf{W}^{[k]}$ weights matrix at the *kth* layer.
- $\mathbf{b}^{[k]}$ bias variable at the *kth* layer.
- $\mathbf{z}^{[k]}$ hidden variables at the *kth* layer.
- $\mathbf{a}^{[k]}$ hidden variables at the *kth* layer after we apply component-wise activation function. (We have not introduced activation functions and their importance yet.)

What we have just learned about neural networks is not entirely right. We missed a few important parts. Our network is linear, which means it can only compute or approximate well to the family of linear functions. However, in most cases, linear

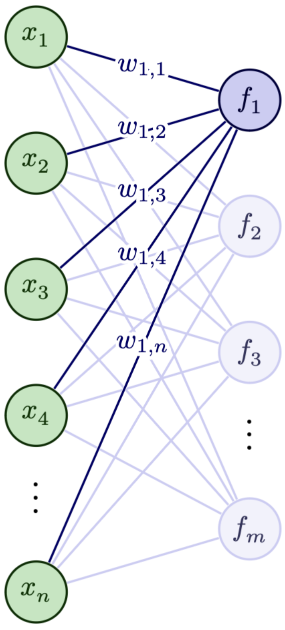


Figure 3: Hidden layer with m neuron

functions are not good enough. We want our neural network to be able to approximate non-linear functions too. For that, we will need activation functions. We redefine the artificial neuron.

$$f(x_1, ..., x_n) = \sigma\big(\sum_{i=1}^{n} w_i x_i + b\big) \tag{6}$$

where σ is a nonlinear differentiable function. [1] [2]You may know σ as the sigmoid function from your earlier studies of machine learning. It is indeed. However, we have plenty of activation functions and we use the σ letter for all of them. The most common ones are:

[1]Refer this link for the definition of a differentiable function

[2]Activation functions do not have to necessarily be differentiable everywhere. Refer this discussion

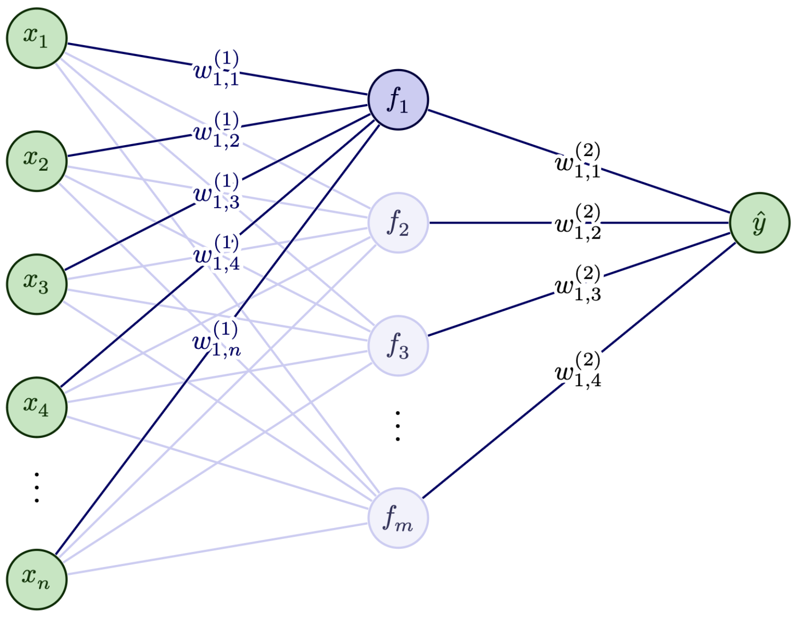
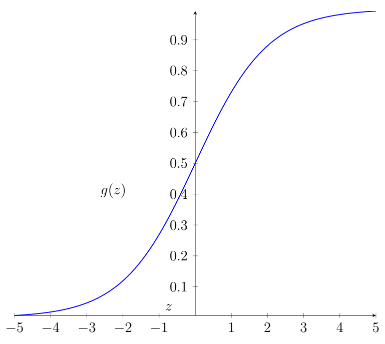
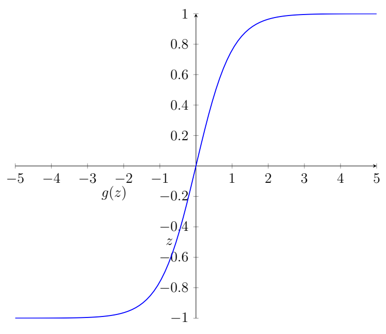


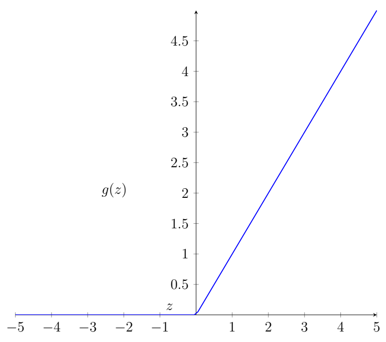
Figure 4: A shallow neural network



(a) Sigmoid $\rightarrow \frac{1}{1+e^{-z}}$



(b) Tanh $\rightarrow \frac{e^z-e^{-z}}{e^z+e^{-z}}$



(c) ReLU $\rightarrow \max(0, z)$

Figure 5: Activation Functions

Activation functions are a critical component of neural networks in deep learning. As we state above, they introduce non-linearity into the network, enabling it to learn and model complex patterns in data. Without activation functions, a neural network would simply be a linear regression model, incapable of capturing intricate relationships. Common activation functions (Figure 5) include the Sigmoid(a), Tanh(b), and ReLU(c). The Sigmoid function maps input values to a range between 0 and 1, the Tanh function maps inputs to a range between -1 and 1, and the ReLU function outputs zero for negative inputs and the input itself for positive values.

2.2 Forward Propagation

Now suppose we have the following neural network

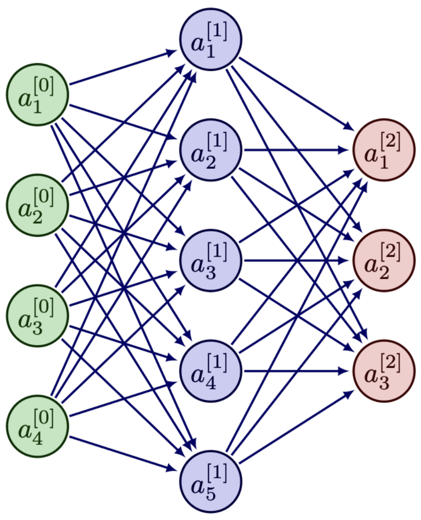


Figure 6: A shallow neural network

The forward propagation equations in this network can be written using the vector and matrix forms as follows:

$$\begin{aligned} \mathbf{x} &= \mathbf{a}^{[0]} \\ \mathbf{z}^{[1]} &= \mathbf{W}^{[1]}\mathbf{a}^{[0]} + \mathbf{b}^{[1]} \\ \mathbf{a}^{[1]} &= \sigma(\mathbf{z}^{[1]}) \\ \mathbf{z}^{[2]} &= \mathbf{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]} \\ \mathbf{a}^{[2]} &= \sigma(\mathbf{z}^{[2]}) \end{aligned} \tag{7}$$

In this setting, $\mathbf{W}^{[1]}$ is a matrix with shape 5×4. $\mathbf{W}^{[2]}$ is a matrix with shape 3×5. $\mathbf{z}^{[1]}$ is a vector with shape 5×1, $\mathbf{a}^{[1]}$ has the same shape as $\mathbf{z}^{[1]}$ and $\mathbf{b}^{[2]}$ has the shape 3×1. Other shapes left to the curious readers as an exercise. It is clear that the shape of weight matrices are directly related to the number of neuron between layers.

We can rewrite equations in a more open form to make you persuade.

$$
\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} a_1^{[0]} \\ a_2^{[0]} \\ a_3^{[0]} \\ a_4^{[0]} \end{bmatrix}
$$

$$
\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_{1,1}^{[1]} & w_{1,2}^{[1]} & w_{1,2}^{[1]} & w_{1,2}^{[1]} \\ w_{2,1}^{[1]} & w_{2,2}^{[1]} & w_{2,3}^{[1]} & w_{2,4}^{[1]} \\ w_{3,1}^{[1]} & w_{3,2}^{[1]} & w_{3,3}^{[1]} & w_{3,4}^{[1]} \\ w_{4,1}^{[1]} & w_{4,2}^{[1]} & w_{4,3}^{[1]} & w_{4,4}^{[1]} \\ w_{5,1}^{[1]} & w_{5,2}^{[1]} & w_{5,3}^{[1]} & w_{5,4}^{[1]} \end{bmatrix} \begin{bmatrix} a_1^{[0]} \\ a_2^{[0]} \\ a_3^{[0]} \\ a_4^{[0]} \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \\ b_5^{[1]} \end{bmatrix} \tag{8}
$$

$$
\begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \sigma\left(\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} \right) = \begin{bmatrix} \sigma(z_1^{[1]}) \\ \sigma(z_2^{[1]}) \\ \sigma(z_3^{[1]}) \\ \sigma(z_4^{[1]}) \end{bmatrix}
$$

This is it for the first layer. Curious readers can try to write equations in the second layer using matrix and vector forms as an exercise. We now want to generalize it and find a formula for kth layer.

Let us calculate $a_1^{[k]}$ in the Figure 7.

$$
\begin{aligned}
a_1^{[k]} &= \sigma(w_{1,1}a_1^{[l]} + w_{1,2}a_2^{[l]} + ... + w_{1,n}a_n^{[l]}) \\
a_1^{[k]} &= \sigma\left(\sum_{i=1}^{n} w_{1,i}a_i^{[l]} \right)
\end{aligned} \tag{9}
$$

Then if we stack $w_{i,j}$'s one under the other while considering i is the ith neuron in the layer k and j stands for the jth neuron in the layer $k-1$. We would end up with following matrix-vector form for the forward propagation from layer $k-1$ to layer k

$$
\begin{bmatrix} a_1^{[k]} \\ a_2^{[k]} \\ \vdots \\ a_m^{[k]} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{bmatrix} \begin{bmatrix} a_1^{[k-1]} \\ a_2^{[k-1]} \\ \vdots \\ a_n^{[k-1]} \end{bmatrix} + \begin{bmatrix} b_1^{[k]} \\ b_2^{[k]} \\ \vdots \\ b_m^{[k]} \end{bmatrix} \right) \tag{10}
$$

Considering n is the number of neuron in layer $k-1$ and m is the number of neurons in layer k we have the following equation and it is equivalent mathematical expression of (10).

$$
\underset{m\times 1}{\mathbf{a}^{[k]}} = \sigma\left(\underset{m\times n}{\mathbf{W}^{[k]}} \underset{n\times 1}{\mathbf{a}^{[k-1]}} + \underset{m\times 1}{\mathbf{b}^{[k]}} \right) \tag{11}
$$

2.3 Backward Propagation

We proceed with **backward propagation**, also called **backprop**. It allows us to compute the gradients of our learnable parameters efficiently. Then, we use a gradient-based algorithm like **stochastic gradient descent** to update the weights. In this way, we achieve learning.

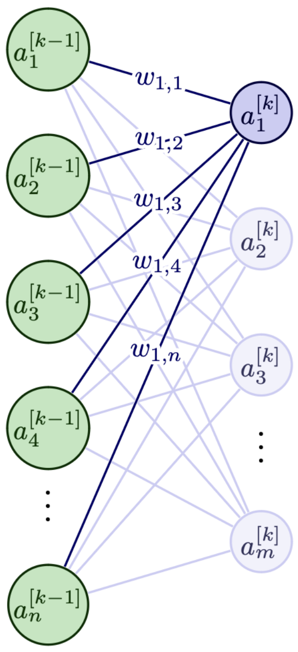


Figure 7: Forward propagation from layer $k-1$ to layer k

Learning is essentially equivalent to finding one of the minima of a function. This function is called the **cost function**, which is also interchangeable with the terms **objective function** and **loss function**. These terms can be confusing, but here's the key distinction:

- Objective function and cost function are synonymous in our context. They represent the function we aim to minimize during training.
- The loss function calculates the error for a single training example. When we sum the loss function over all examples, we get the cost function.

However, don't get hung up on terminology for too long. The core concept is captured by the following equation:

$$\mathcal{C}_\theta = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}_\theta(y_i, \hat{y}_i) \tag{12}$$

Here, you might encounter a common loss function called MSE (mean squared error):

$$\mathcal{L}_\theta(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2. \tag{13}$$

We will compute the gradients of the cost function with respect to our learnable parameters. As you know, our learnable parameters are w_{ij}'s and b_i's.

During training, the network first performs a forward pass which we have just covered and where the input data propagates through the network layers to generate an output which we called $\hat{y}$ or $\mathbf{a}^{[L]}$. Then we continue with backpropagation: the error between the predicted and actual output is calculated. This error is then propagated backwards through the network, layer by layer. At each layer, the gradient of the error with respect to the weights and biases is calculated using the chain rule and the activation function used in that layer.

As we did in forward propagation we start with simple examples. Assume you have a simple neural network in the Figure 8.

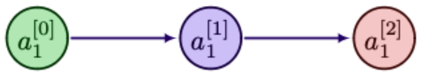


Figure 8: A neural network with one neurons in all layers

Then our forward propagation equations are the following:

$$\begin{aligned} z^{[1]} &= w^{[1]}a^{[0]} + b^{[1]} \\ a^{[1]} &= \sigma(z^{[1]}) \\ z^{[2]} &= w^{[2]}a^{[1]} + b^{[2]} \\ a^{[2]} &= \sigma(z^{[2]}) \end{aligned} \tag{14}$$

and the function we want to minimize is

$$\mathcal{L}(y, a^{[2]}) = (y - a^{[2]})^2 \tag{15}$$

We want to compute $\frac{\partial \mathcal{L}(y,a^{[2]})}{\partial w^{[2]}}, \frac{\partial \mathcal{L}(y,a^{[2]})}{\partial b^{[2]}}, \frac{\partial \mathcal{L}(y,a^{[2]})}{\partial w^{[1]}}, \frac{\partial \mathcal{L}(y,a^{[2]})}{\partial b^{[1]}}$.
Let us compute $\frac{\partial \mathcal{L}(y,a^{[2]})}{\partial w^{[2]}}$

$$\begin{aligned} \mathcal{L}(y, a^{[2]}) &= (y - a^{[2]})^2 \\ &= (y - \sigma(z^{[2]})^2 \\ &= (y - (w^{[2]}a^{[1]} + b^{[2]}))^2 \end{aligned} \tag{16}$$

and the derivate

$$\frac{\partial((y - (w^{[2]}a^{[1]} + b^{[2]}))^2)}{\partial w^{[2]}} = 2(y - \sigma(z^{[2]})) \cdot (\sigma' \cdot (w^{[2]}a^{[1]} + b^{[2]}) \cdot a^{[1]}) \tag{17}$$

We used the chain rule to compute this derivative. In this way, any derivative we need can be computed. While taking derivatives, you can draw a tree diagram below to keep track of which variable is a function of which.

$$\begin{tikzcd}
 & a^{[2]} \arrow[d, dash] & \\
 & z^{[2]} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
w^{[2]} & a^{[1]} \arrow[d, dash] & b^{[2]} \\
 & z^{[1]} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
w^{[1]} & a^{[0]} & b^{[1]}
\end{tikzcd}$$

Thus by looking at the tree we can write:

$$\frac{\partial \mathcal{L}(y, a^{[2]})}{\partial w^{[2]}} = \underbrace{\frac{\partial \mathcal{L}(y, a^{[2]})}{\partial a^{[2]}}}_{2(y-a^{[2]})} \cdot \underbrace{\frac{\partial a^{[2]}}{\partial z^{[2]}}}_{\sigma'(z^{[2]})} \cdot \underbrace{\frac{\partial z^{[2]}}{\partial w^{[2]}}}_{a^{[1]}} \tag{18}$$

$$\frac{\partial \mathcal{L}(y, a^{[2]})}{\partial b^{[2]}} = \underbrace{\frac{\partial \mathcal{L}(y, a^{[2]})}{\partial a^{[2]}}}_{2(y-a^{[2]})} \cdot \underbrace{\frac{\partial a^{[2]}}{\partial z^{[2]}}}_{\sigma'(z^{[2]})} \cdot \underbrace{\frac{\partial z^{[2]}}{\partial b^{[2]}}}_{1} \tag{19}$$

$$\frac{\partial \mathcal{L}(y, a^{[2]})}{\partial w^{[1]}} = \underbrace{\frac{\partial \mathcal{L}(y, a^{[2]})}{\partial a^{[2]}}}_{2(y-a^{[2]})} \cdot \underbrace{\frac{\partial a^{[2]}}{\partial z^{[2]}}}_{\sigma'(z^{[2]})} \cdot \underbrace{\frac{\partial z^{[2]}}{\partial a^{[1]}}}_{w^{[2]}} \cdot \underbrace{\frac{\partial a^{[1]}}{\partial z^{[1]}}}_{\sigma'(z^{[1]})} \cdot \underbrace{\frac{\partial z^{[1]}}{\partial w^{[1]}}}_{a^{[0]}} \tag{20}$$

We have just seen the calculations for a very small neural network with only two weights and one neuron. Naturally, with more neurons, the calculations become more complex, requiring Jacobian matrices and function gradients to compute the derivatives. However, these details are beyond the scope of our scope. See pages 200-223 in [2] for more details.

Backpropagation is a powerful algorithm implemented in many software packages. While you can often call it with a single line of code, it's important to understand its core function: calculating the changes in learnable parameters. This allows us to find a local minimum in the objective function, which we aim to minimize. The process of approaching this minimum is handled by optimization algorithms, which we'll explore in the coming weeks.

3 Neural Networks as Universal Approximators

Neural networks, particularly deep neural networks, have demonstrated remarkable capabilities as universal approximators. This concept, rooted in the **universal approximation theorem** [1][3][4]. Basically, it states that a sufficiently large neural network can approximate any continuous function to an arbitrary degree of accuracy, given appropriate weights and architecture. The theorem statement as follows

Theorem (Universal function approximation)[5]
Let $\sigma \in C(\mathbb{R}, \mathbb{R})$ be a non-polynomial activation function. For every $n, m \in \mathbb{N}$, every compact subset $K \subseteq \mathbb{R}^n$, every function $f \in C(K, \mathbb{R}^m)$ and $\epsilon > 0$, there exist $k \in \mathbb{N}$, $\mathbf{A} \in \mathbb{R}^{k \times n}$, $\mathbf{b} \in \mathbb{R}^k$, and $\mathbf{C} \in \mathbb{R}^{m \times k}$ such that

$$\sup_{x \in K} \|f(x) - g(x)\| < \epsilon,$$

where $g(x) = \mathbf{C}\sigma(\mathbf{A}x + \mathbf{b})$.

In summary, a feedforward network with just one layer can represent any function, but this layer might need to be extremely large and may not learn or generalize well. In many cases, using deeper models (more layers) can decrease the number of units needed to represent the function and also reduce the generalization error.

So, this theorem does not guarantee that we will successfully train our network and that it will be able to represent the desired function. It simply states: "*Your network has the **ability** to represent the function, but **learning** it is another job.*"

References

[1] CYBENKO, G. V. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems 2* (1989), 303–314.

[2] GOODFELLOW, I., BENGIO, Y., AND COURVILLE, A. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[3] HORNIK, K., STINCHCOMBE, M., AND WHITE, H. Multilayer feedforward networks are universal approximators. *Neural Networks 2*, 5 (1989), 359–366.

[4] LESHNO, M., LIN, V. Y., PINKUS, A., AND SCHOCKEN, S. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks 6*, 6 (1993), 861–867.

[5] PINKUS, A. Approximation theory of the mlp model in neural networks. *Acta Numerica 8* (1999), 143–195.