Lecture 8
Neural Networks, Part 1

Alice Gao

November 2, 2021

Contents

1 Learning Goals

By the end of the lecture, you should be able to

- Describe motivations for using a neural network model.
- Describe the simple mathematical model of a neuron.
- Describe desirable properties of an activation function. Give examples of activation functions and their properties.
- Distinguish feed-forward and recurrent neural networks.
- Learn a perceptron that represents a simple logical function.
- Determine the logical function represented by a perceptron.
- Explain why a perceptron cannot represent the XOR function.
- Construct a 3-layer neural network that represents the XOR function.

2 Introduction to Artificial Neural Networks

There is currently a lot of hype around neural networks. How did we get to this point?

2.1 History and background

First, a few terms. What are artificial intelligence, machine learning, and deep learning, and what are the relationships between these?

- Artificial Intelligence: building machines that behave intelligently.
- Machine Learning: letting computers learn without being explicitly programmed. A branch of AI. Primarily uses statistical methods to achieve learning.
- Deep Learning: developing hierarchical networks that mimic the human brain. A branch of ML.

You might be surprised that deep learning has been around for decades, but only recently (in the late 2000's) become very successful. This was mainly due to more powerful computers and the availability of lots of data.

There are two important events you should know about for deep learning.

In 2012, a deep learning algorithm called AlexNet won the ImageNet challenge. This was a supervised learning application.

In the same year, Google Brain made a breakthrough by successfully using deep learning for an unsupervised learning application. After processing over 10 million images from YouTube videos, one node of the neural network developed a very strong affinity for cats and was able to recognize when an image had a cat in it. This experiment has been called the infamous Cat Experiment.

Hopefully, this gives you some insight as to why deep learning is so popular now.

2.2 Learning complex relationships

Consider some applications. Maybe we have an image and we want to recognize what's in the image: image interpretation. Or maybe we want speech recognition: we're developing something like Siri, which can help us perform tasks by talking to it. Or maybe we want a machine to translate text for us.

These are all applications where the data is complex. In particular, the relationship between inputs and outputs is complex.

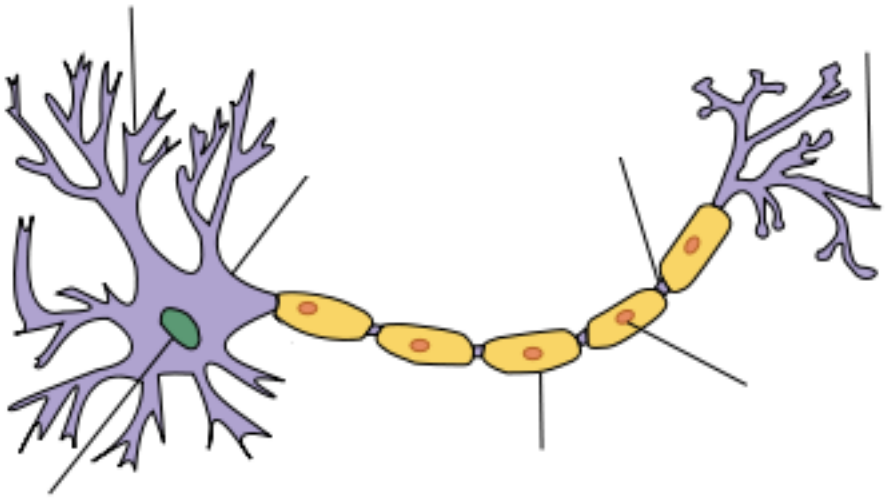
How can we build a model to learn such complex relationships?

We already know that humans can learn these complex relationships very well. So if humans can do it, is it possible to build a model that mimics the human brain?

This was the original motivation for artificial neural networks.

2.3 Human brains

Thanks to Neuroscience, now we have some understanding of the structure of the human brain. The brain is a set of densely connected neurons.



Each neuron is a very simple unit, but it still has many components.

- Dendrites: receive inputs from other neurons (left)
- Soma: controls activity of the neuron (centre)
- Axon: sends outputs to other neurons (right)
- Synapse: links neurons

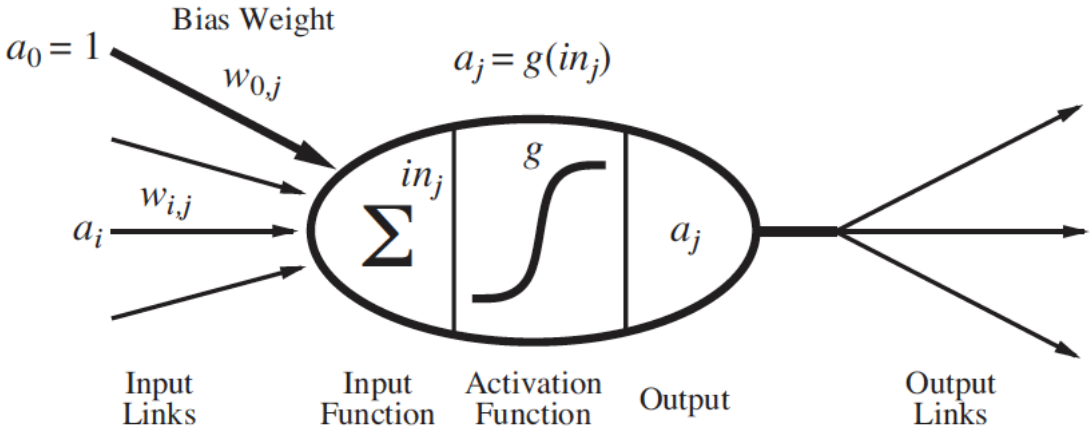
Don't worry about these terms; this is just to motivate our mathematical computational model for the neuron. Here, a neuron takes input, does some (usually very simple) computation, and decides on the strength of its output, if any.

As a computational model, this contrasts with the other models we've talked about, because it is much simpler. For example, if we combine multiple decision trees into what's called a random forest, each model is very complex. But the model of the human brain is very different: it has a large number of simple components, rather than a small number of complex components. No one model is inherently better than another, though.

2.4 A simple mathematical model of a neuron

McCulloch and Pitts first proposed this model of a neuron in 1943. This is a very simple model; actual neurons have much more complex behaviour.

This model says that a neuron is a linear classifier: it "fires" when a linear combination of its inputs exceeds some threshold.



Neuron j computes a weighted sum of its input signals a_i, where w_{ij} is a bias weight:

$$in_j = \sum_{i=0}^{n} w_{ij} a_i.$$

Neuron j then applies an activation function g to the weighted sum to derive the output:

$$a_j = g(in_j) = g\left(\sum_{i=0}^{n} w_{ij} a_i\right).$$

Notice that the input a_0 is always set to 1; this is called a bias or dummy input. Remember, our model is a linear classifier: we want to represent a linear function, but our inputs are all variables, so we need the possibility of having a constant term. This $a_0 = 1$ lets us have that constant term.

2.5 Desirable properties of the activation function

The activation function is very important in our model. Let's look at three of the desirable properties an activation function should have.

- It should be non-linear.

  Combining linear functions will not produce a non-linear function, but complex relationships are often non-linear. Using a non-linear activation function allows us to model non-linear relationships by interleaving linear functions (weighted sums of inputs) with non-linear functions (activation functions).

- It should mimic the behaviour of real neurons.

  If the weighted sum of input signals is large enough, the neuron fires. Otherwise, it does not fire. (No need for a hard threshold, can fire an output signal of some amount)

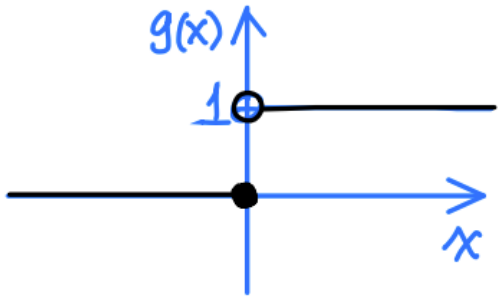
- It should be differentiable almost everywhere.

  We want to learn a neural network using gradient descent or other algorithms, which require the activation function to be differentiable.

2.6 Common activation functions

- Step function: $g(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$.

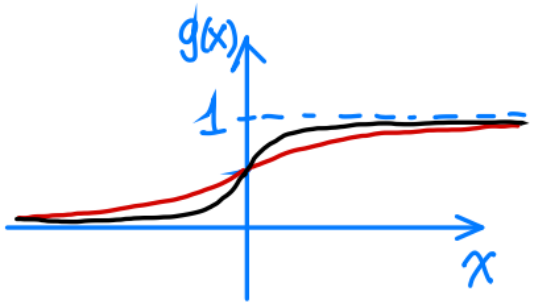
  Simple to use, but not differentiable and not used in practice. Useful for explaining concepts.



- Sigmoid function: $g(x) = \dfrac{1}{1 + e^{-kx}}$.

  Can approximate the step function (red: smaller k, worse approximation; black: larger k, better approximation). Gives clear, bounded prediction. Differentiable.

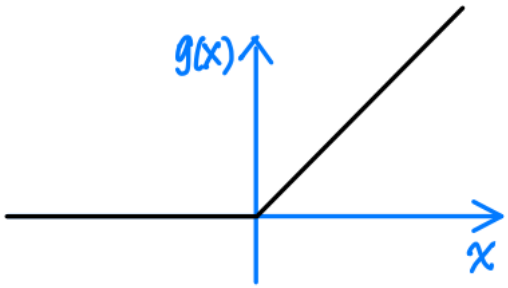
  Preferred for a long time, but has "vanishing gradient" problem (gradient is very small for extreme values of x) and is computationally expensive.



- Rectified linear unit (ReLU): $g(x) = \max(0, x)$.

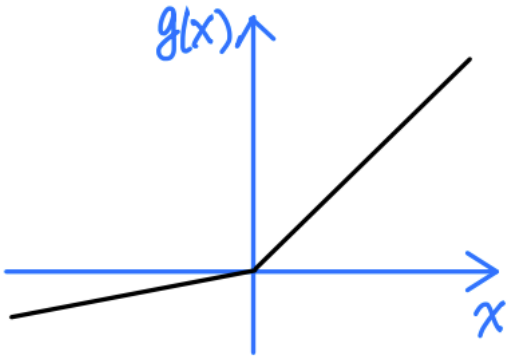
  Computationally efficient. Non-linear and differentiable.

  "Dying ReLU" problem: gradient is 0 for negative x or x approaching 0.



- Leaky ReLU: $g(x) = \max(0.1x, x)$.

  Like a ReLU, but enables learning for negative input values.

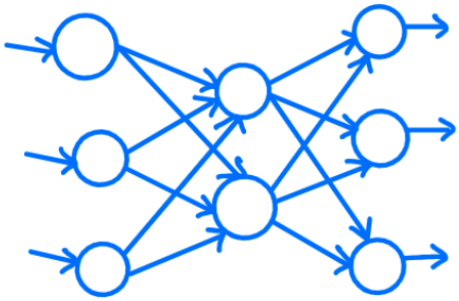


3 Introduction to Perceptrons

3.1 Feed-forward v.s. recurrent neural networks

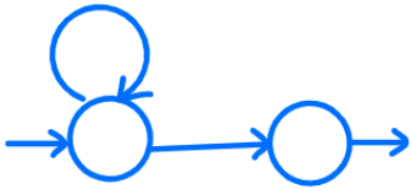
Let me discuss two types of neural networks: feed-forward neural network and recurrent neural network. See a 2-layer feed-forward network below. It is a directed acyclic graph. The key property is that the network has no loops. Think about the numbers of flowing through the network. They come in as input values, get transformed through the edges and

the nodes, and finally go out as output values. These values only flow in one direction. They never go backwards.



In a feed-forward network, the outputs are a function of its inputs. If you know the inputs, you can determine the outputs.

On the other hand, a recurrent network can have loops. See an example below. We have taken an output from a node and feed the value back into the node as an input. What's the advantage of having a loop? You may have learned this in a circuit design course. Loops in a circuit can give it memory. The circuit can remember some information while it's doing computations. Similarly, a recurrent network has memory.



The complication is that, the output value is no longer a function of its inputs. The output may also depend on what the network remembers about the historical inputs. Historical inputs determine the current state, which influences the output.

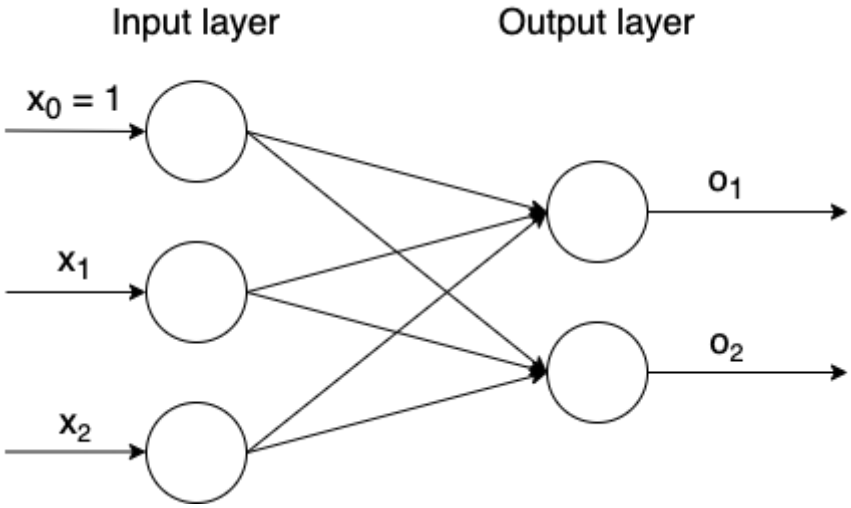
Arguably, a recurrent network is a better model of the human brain, because we have memory. Unfortunately, because of the loops, it's also more difficult to train a recurrent network and to interpret their meanings.

In this course, we will focus on feed-forward networks. If you are interested in learning more about other types of neural networks, I'd recommend taking CS 480 on Intro to Machine Learning and CS 479 on Neural Networks.

3.2 Perceptrons

Let's look at a perceptron. A perceptron is a single-layer feed-forward neural network. In the picture below, it looks like there are two layers, the input layer and the output layer. However, the input layer does not have neurons. It simply contains the input values. The

output layer has two neurons. Each neuron takes the input values, calculates the weighted sum, and feeds the result through some activation function to produce the output values.



A perceptron can represent simple logical functions, such as AND, OR, and NOT. This was a big deal when it was discovered. At the time, much of AI was focusing on developing systems to perform formal logical reasoning. Representing logical functions is the first step towards this goal. Many people thought perceptrons would eventually allow us to develop a powerful logical deduction system, which is capable of solving any problem.

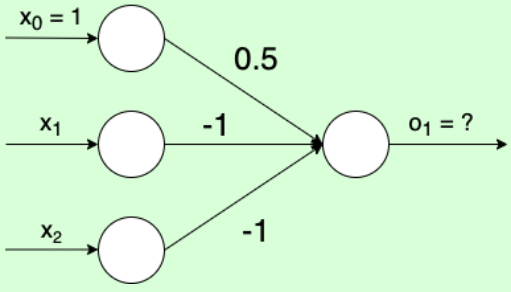
I told you that a perceptron can represent simple logical functions. How does it work?

Before we look at the mathematics, let's take another look at the perceptron on the previous slide. This network appears to have two outputs, but two outputs are independent. We can split up the network into two independent networks, each containing one perceptron. And, we can learn the weights in each network separately. Because of this, I am going to focus on discussing a single perceptron at a time.

Back to our question. How can we use a perceptron to represent a logical function? Let's look at our first question below.

Problem: Consider the following perceptron where the activation function is the step function ($g(x) = 1$ if $x > 0$, $g(x) = 0$ if $x \leq 0$.) Which of the following logical functions does the perceptron compute?

(A) $x_1 \wedge x_2$
(B) $\neg(x_1 \wedge x_2)$
(C) $x_1 \vee x_2$
(D) $\neg(x_1 \vee x_2)$



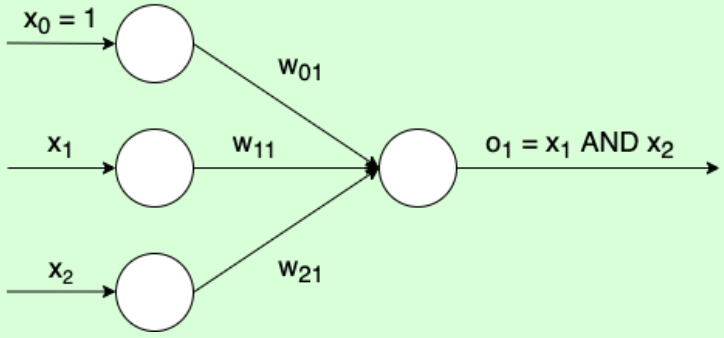
Solution: The correct answer is (D). This perceptron is computing a negation of the OR function. The easiest way to derive this answer is to draw a truth table. With two inputs, there should be four rows in the table.

x_1	x_2	o_1
0	0	1
0	1	0
1	0	0
1	1	0

See above for the complete truth table. Next, we need to stare at this truth table and figure out the function that o1 is representing. It looks like a negated version of the OR function. The OR function would be true if at least one of x1 and x2 is 1, and this looks like the opposite of OR.

I found this example interesting. Although we have real numbers on the edges, we can use them to compute binary outputs. Looking at this example, you might be wondering, how did I come up with these weights in the first place? Let's explore this in the next question.

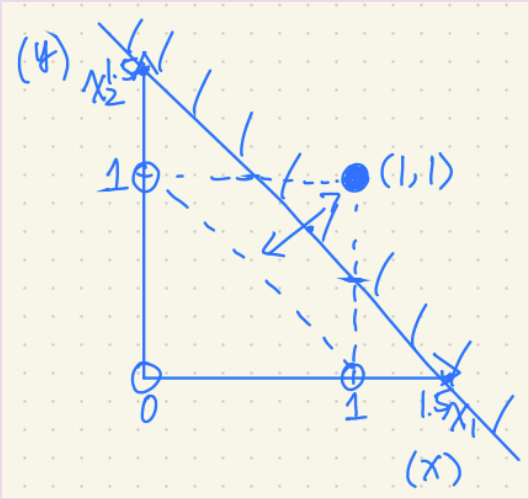
Problem: Consider the following perceptron where the activation function is the step function ($g(x) = 1$ if $x > 0$, $g(x) = 0$ if $x \leq 0$). What should the weights w_{01}, w_{11}, and w_{21} be such that the perceptron represents an AND function?



x_1	x_2	o_1
0	0	0
0	1	0
1	0	0
1	1	1

Solution: There are many correct answers. Here is one of them. The key insight is that a perceptron is a linear classifier. Given the data points, we need to find a line that separates the positive examples from the negative examples. Then, we can derive an inequality that selects the side with the positive examples. This inequality will tell us the weights in the perceptron.

Let's draw the inputs in the x_1x_2-plane here, shade in the inputs that correspond to an output of 1, and leave the others empty. You'll notice that there is a rather large gap between our shaded and unshaded inputs. Any line going through this gap would work, but we'll pick one of the nicest ones: this line with x_1 and x_2 intercepts of 1.5 and a slope of -1.



Our line is $x_2 = -x_1 + 1.5$, which we can rewrite as $x_1 + x_2 - 1.5 = 0$.

However, we want to classify all inputs on one side as "firing" our neuron, so we need this to be an inequality that is positive for inputs on one side. Here, that means the top-right side of the line. That is, our $(1, 1)$ input should be positive when we substitute it into the left-hand side of the inequality. We see that $1 + 1 - 1.5 = 0.5 > 0$, so the inequality we need is $x_1 + x_2 - 1.5 > 0$.

If instead we got a negative value for our $(1, 1)$ input, the inequality would flip and we would negate the coefficients.

Finally, we read the weights off of the coefficients: $w_{01} = -1.5$, $w_{11} = 1$, $w_{21} = 1$.

For a general logical function, you can draw the inputs and attempt to place a line which classifies the inputs correctly. The coefficients of the line tell the weights, but you may have to adjust the signs.

4 Limitations of Perceptrons

4.1 The First AI Winter

From the 1950s to the 1960s, research into perceptrons seemed very promising. At the time, people believed AI could be solved if computers could be made to perform formal logical reasoning. Recall that we can use perceptrons to represent simple logical functions, like AND, OR, or NOT, and that these logical functions are the building blocks of a logical deduction system.

Unfortunately, in the late 1960s, some limitations of perceptrons were discovered. Marvin Minsky, the founder of the MIT AI lab, and Seymour Papert, the director of the lab at the time, were skeptical about perceptrons. They studied perceptrons and found a significant limitation. They recorded their findings in a book called "Perceptrons: An Introduction to Computational Geometry".

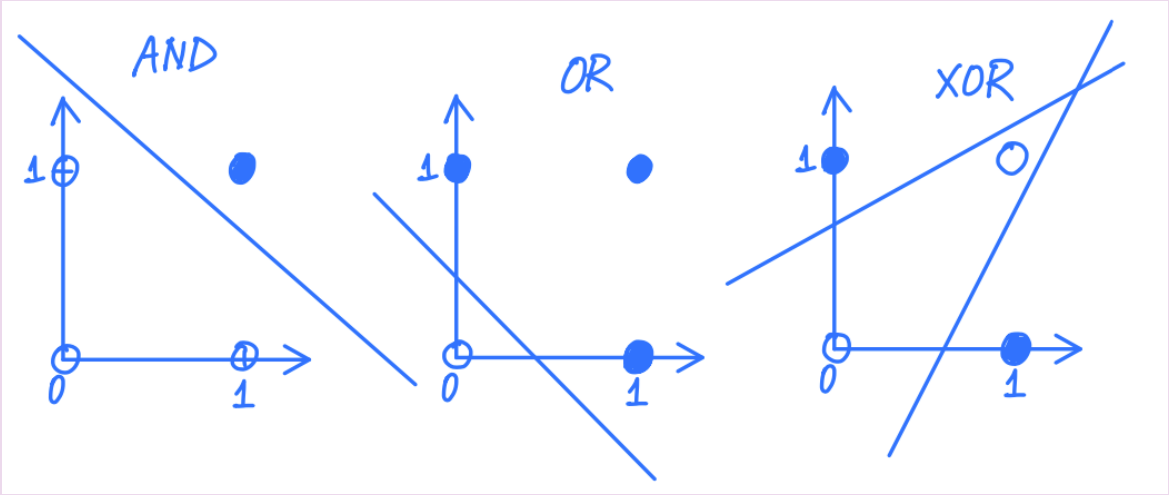
In the book, Minsky and Papert showed that it is not possible to represent an XOR function using perceptrons only - a deeper network with at least two layers is needed. Recall that the XOR of two inputs is true whenever the two inputs are different. They are either 1 and 0 or 0 and 1.

This fact by itself is not a problem. If a deeper network is needed, then we can construct one and train it to learn our target function. Unfortunately, at the time, nobody knew how to train a neural network with at least two layers. People only knew how to train a perceptron - a single-layer network.

These two results combined together suggested that pursuing perceptrons may be a dead end. This discovery had a huge impact on the research towards artificial neural networks, and it is believed that the book led to the first AI winter. There was a freeze to funding on neural networks. It also became challenging to publish any paper on perceptrons, since it was not thought to be a promising research direction.

Problem: Why can't a perceptron represent XOR?

Solution: Intuitively, a perceptron is a linear classifier (given that we use the step function as the activation function), but XOR is not linearly separable. In the drawing below, you can see that AND and OR are linearly separable, but there is no way to linearly separate the shaded outputs of XOR.

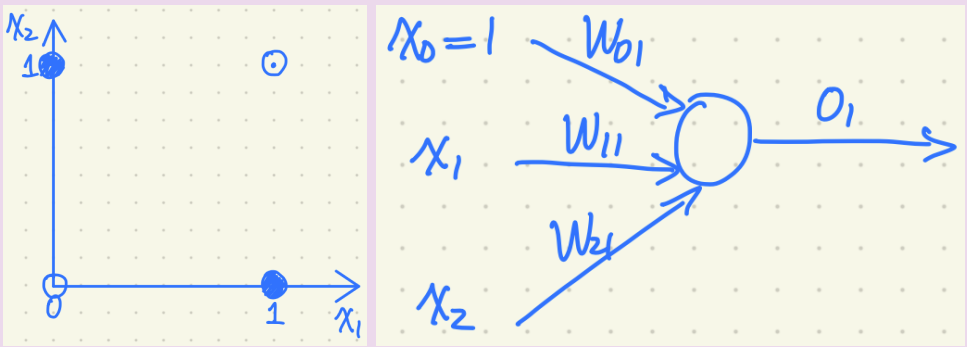


Let's prove this rigorously.

Proof. Assume towards a contradiction that we can represent the XOR function using a perceptron. The activation function is the step function

$$g(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}.$$

Below is the graph of our data points again and a drawing of the perceptron for reference.



Since we know how our perceptron classifies the four input points, we can use the activation function to derive four inequalities:

$$\begin{aligned} w_{21} \cdot 1 + w_{11} \cdot 0 + w_{01} &> 0 \\ w_{21} \cdot 0 + w_{11} \cdot 1 + w_{01} &> 0 \\ w_{21} \cdot 1 + w_{11} \cdot 1 + w_{01} &\leq 0 \\ w_{21} \cdot 0 + w_{11} \cdot 0 + w_{01} &\leq 0 \end{aligned}$$

Simplifying and rewriting these, we have

$$w_{21} + w_{01} > 0 \implies w_{21} > -w_{01} \quad (1)$$
$$w_{11} + w_{01} > 0 \implies w_{11} > -w_{01} \quad (2)$$
$$w_{21} + w_{11} + w_{01} \leq 0 \implies w_{21} + w_{11} \leq -w_{01} \quad (3)$$
$$w_{01} \leq 0 \quad (4)$$

Adding (1) and (2) gives

$$w_{21} + w_{11} > -2w_{01} \quad (5)$$

From (4), we know that $-2w_{01} \geq -w_{01}$. Then we can put (3) and (5) together to get

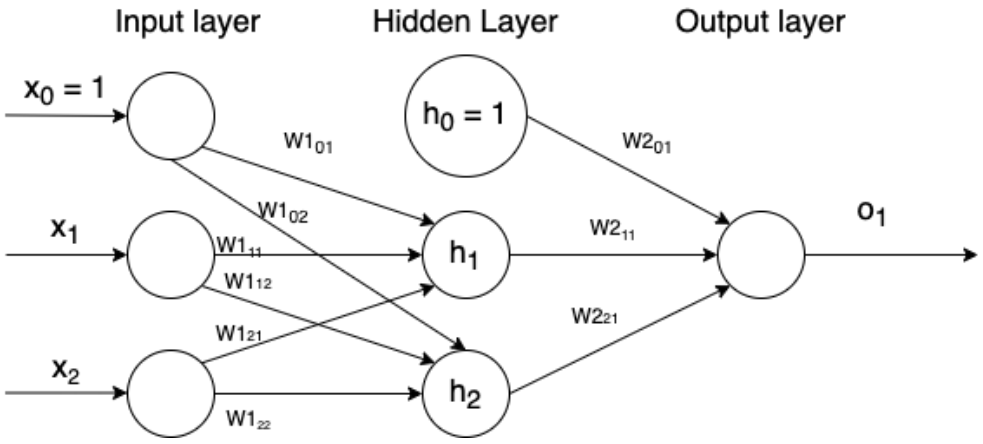
$$w_{21} + w_{11} > -2w_{01} \geq -w_{01} \geq w_{21} + w_{11}$$

which is a contradiction. □

4.2 XOR as a 2-layer neural network

Now that we know we cannot represent XOR with a single-layer neural network, how can we represent XOR? After all, XOR is a simple function, and we should be able to represent it.

It turns out the simplest network we need looks like the following. It may appear that we have three layers: the input layer, the hidden layer, and the output layer. However, the input layer is not a real layer, since it is not performing any computation. It simply contains the input values. The middle layer is called a hidden layer since it is not visible, whereas the input and output are the two visible layers. In a larger network, any layer in the middle is a hidden layer.



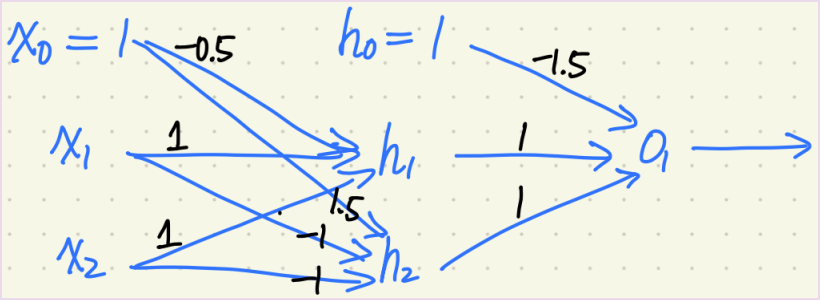
Problem: Give weights such that the network represents the XOR function, using the step function as the activation function.

You may not know where to start, but try to reason about this mathematically. You may also have to recall some knowledge from a previous logic class: try to break down the XOR function into functions we do know how to represent, then put them together.

Solution: Let us channel our inner logician—perhaps recalling CS 245. The XOR of two inputs x_1 and x_2 can be expressed as

$$o_1 = ((x_1 \vee x_2) \wedge (\neg(x_1 \wedge x_2))) .$$

We can represent all of these simpler AND, OR, and NOT components with perceptrons. This should be sufficient for you to figure out the weights, but I will provide one solution anyway by way of the diagram below.



To verify these weights are correct, here are the formulas and truth tables for h_1, h_2, and o_1.

$h_1 = g(x_1 + x_2 - 0.5)$ $\quad$ $h_2 = g(-x_1 - x_2 + 1.5)$ $\quad$ $o_1 = g(h_1 + h_2 - 1.5)$

x_1	x_2	h_1
0	0	0
0	1	1
1	0	1
1	1	1

x_1	x_2	h_2
0	0	1
0	1	1
1	0	1
1	1	0

h_1	h_2	o_1
0	0	0
0	1	0
1	0	0
1	1	1

$h_1 = (x_1 \vee x_2)$ $\quad$ $h_2 = \neg(x_1 \wedge x_2)$ $\quad$ $o_1 = (h_1 \wedge h_2)$

Indeed, $o_1 = (h_1 \wedge h_2) = ((x_1 \vee x_2) \wedge (\neg(x_1 \wedge x_2))) = (x_1$ `XOR` $x_2)$.

5 Practice Problems

1. What are some desirable properties of an activation function?
2. What is the difference between a feed-forward and a recurrent neural network? What is the advantage and the disadvantage of using each type of network?
3. Why is it not possible to represent the XOR function using a perceptron (with the step function as the activation function)?