

[Open in app](#)[Get started](#)

Published in Towards Data Science



Aniruddha Karajgi

[Follow](#)Nov 4, 2020 · 14 min read · [Listen](#)[Save](#)

How Neural Networks Solve the XOR Problem

And why hidden layers are so important

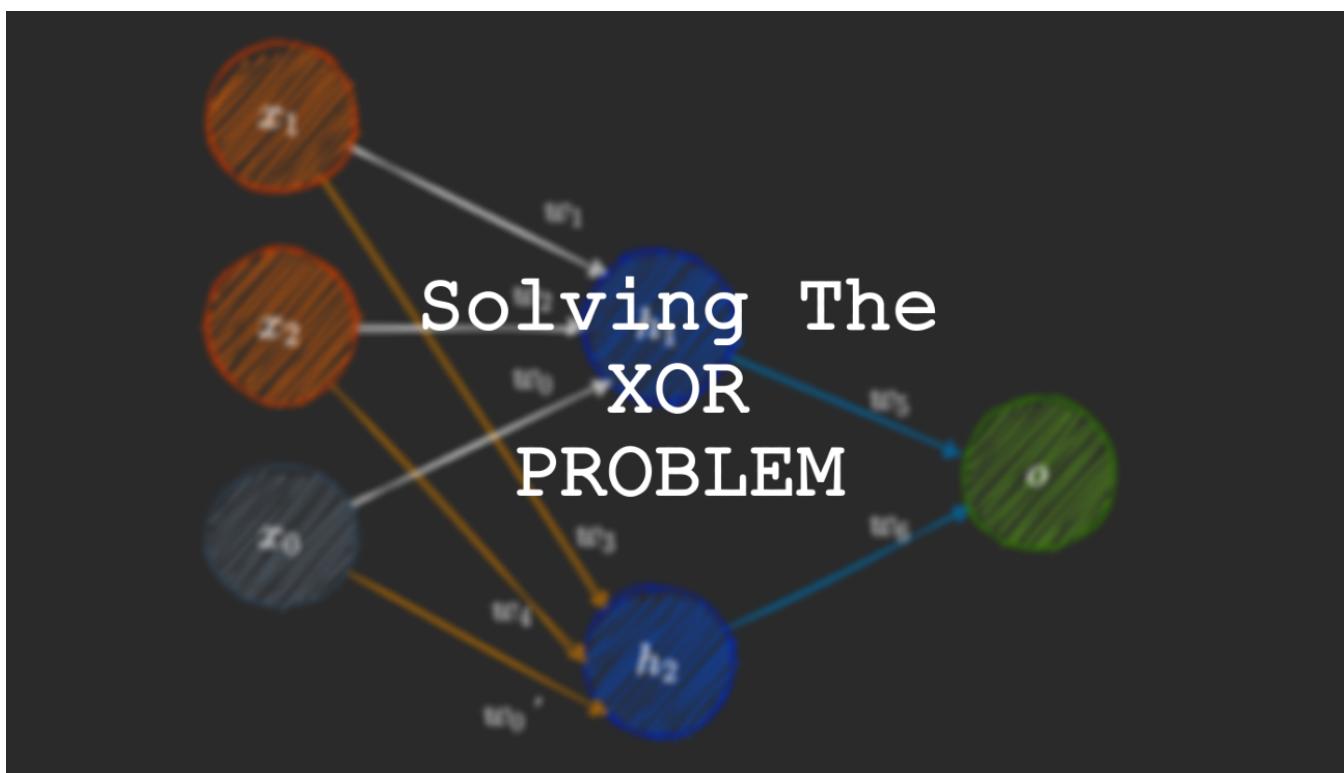


Image by Author

The **perceptron** is a classification algorithm. Specifically, it works as a linear binary classifier. It was invented in the late 1950s by Frank Rosenblatt.

The perceptron basically works as a threshold function — non-negative outputs are put into one class while negative ones are put into the other class.

Though there's a lot to talk about when it comes to neural networks and their variants



[Open in app](#)[Get started](#)

Table of Contents

The Perceptron

- Structure and Properties
- Evaluation
- Training algorithm

2d Xor problem

- The XOR function

Attempt #1: The Single Layer Perceptron

- Implementing the Perceptron algorithm
- Results
- The need for non-linearity

Attempt #2: Multiple Decision Boundaries

- Intuition
- Implementing the OR and NAND parts

The Multi-layered Perceptron

- Structure and Properties
- Training algorithm

Attempt #3: The Multi-layered Perceptron

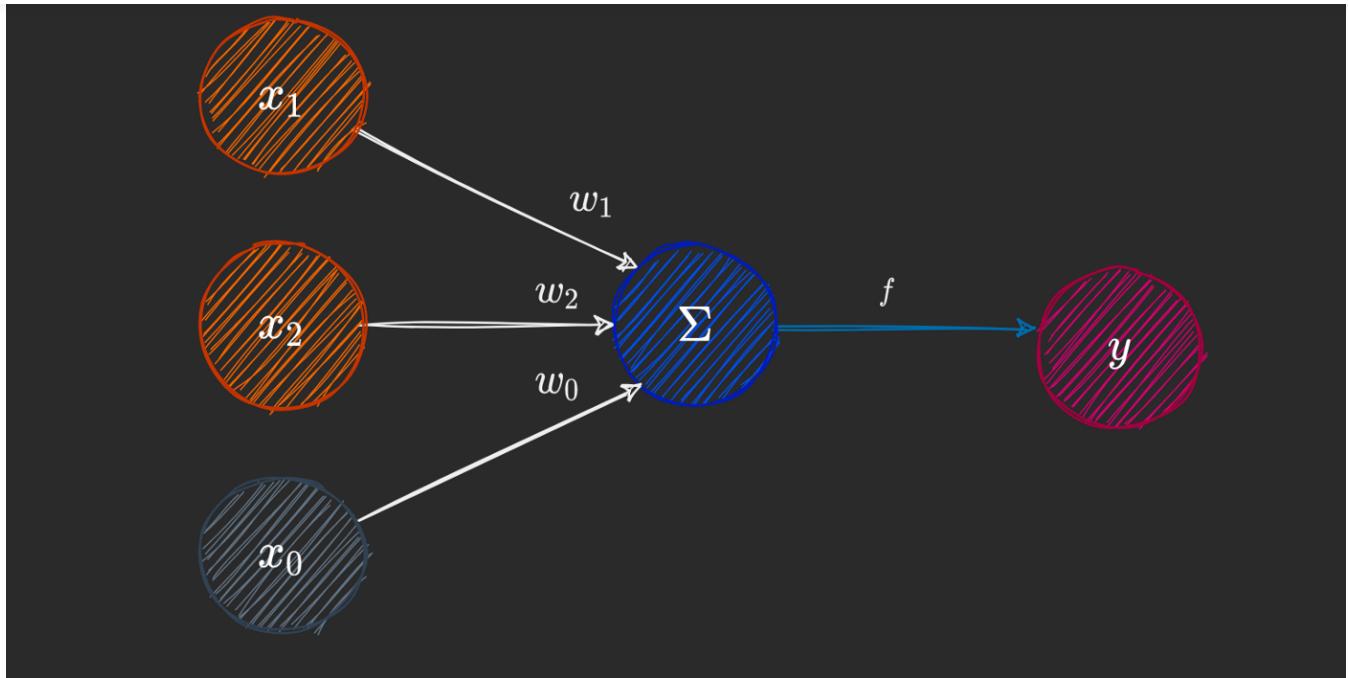
- Implementing the MLP
- Results

Structure and Properties

A perceptron has the following components:

- Input nodes
- Output node
- An activation function
- Weights and biases



[Open in app](#)[Get started](#)

A representation of a single-layer perceptron with 2 input nodes — Image by Author using [draw.io](#)

Input Nodes

These nodes contain the input to the network. In any iteration — whether testing or training — these nodes are passed the input from our data.

Weights and Biases

These parameters are what we update when we talk about “training” a model. They are initialized to some random value or set to 0 and updated as the training progresses. The bias is analogous to a weight independent of any input node. Basically, it makes the model more flexible, since you can “move” the activation function around.

Evaluation

The output calculation is straightforward.

- Compute the dot product of the input and weight vector
- Add the bias
- Apply the activation function.

This can be expressed like so:



[Open in app](#)[Get started](#)

This is often simplified and written as a dot- product of the weight and input vectors plus the bias.

$$y = f(w \cdot X + b)$$

Activation Function

This function allows us to fit the output in a way that makes more sense. For example, in the case of a simple classifier, an output of say -2.5 or 8 doesn't make much sense with regards to classification. If we use something called a sigmoidal activation function, we can fit that within a range of 0 to 1, which can be interpreted directly as a probability of a datapoint belonging to a particular class.

Though there are many kinds of activation functions, we'll be using a simple linear activation function for our perceptron. The linear activation function has no effect on its input and outputs it as is.

Classification

How does a perceptron assign a class to a datapoint?

We know that a datapoint's evaluation is expressed by the relation $wx + b$. We define a threshold (Θ) which classifies our data. Generally, this threshold is set to 0 for a perceptron.

So points for which $wx + b$ is greater than or equal to 0 will belong to one class while the rest ($wx + b$ is negative) are classified as belonging to the other class. We can express this as:



[Open in app](#)[Get started](#)

$$\text{Class}_1: w \cdot X + b \geq 0$$

$$\text{Class}_2: w \cdot X + b < 0$$

Training algorithm

To train our perceptron, we must ensure that we correctly classify all of our train data. Note that this is different from how you would train a neural network, where you wouldn't try and correctly classify your entire training data. That would lead to something called overfitting in most cases.

We start the training algorithm by calculating the **gradient**, or Δw . Its the product of:

- the value of the input node corresponding to that weight
- The difference between the actual value and the computed value.

$$\Delta w = \text{node} \cdot (\text{actual} - \text{computed})$$

We get our new weights by simply incrementing our original weights with the computed gradients multiplied by the learning rate.

$$w_{\text{updated}} = w_{\text{old}} + lr \cdot \Delta w$$

A simple intuition for how this works: if our perceptron correctly classifies an input



[Open in app](#)[Get started](#)

The 2D XOR problem

In the XOR problem, we are trying to train a model to mimic a 2D XOR function.

The XOR function

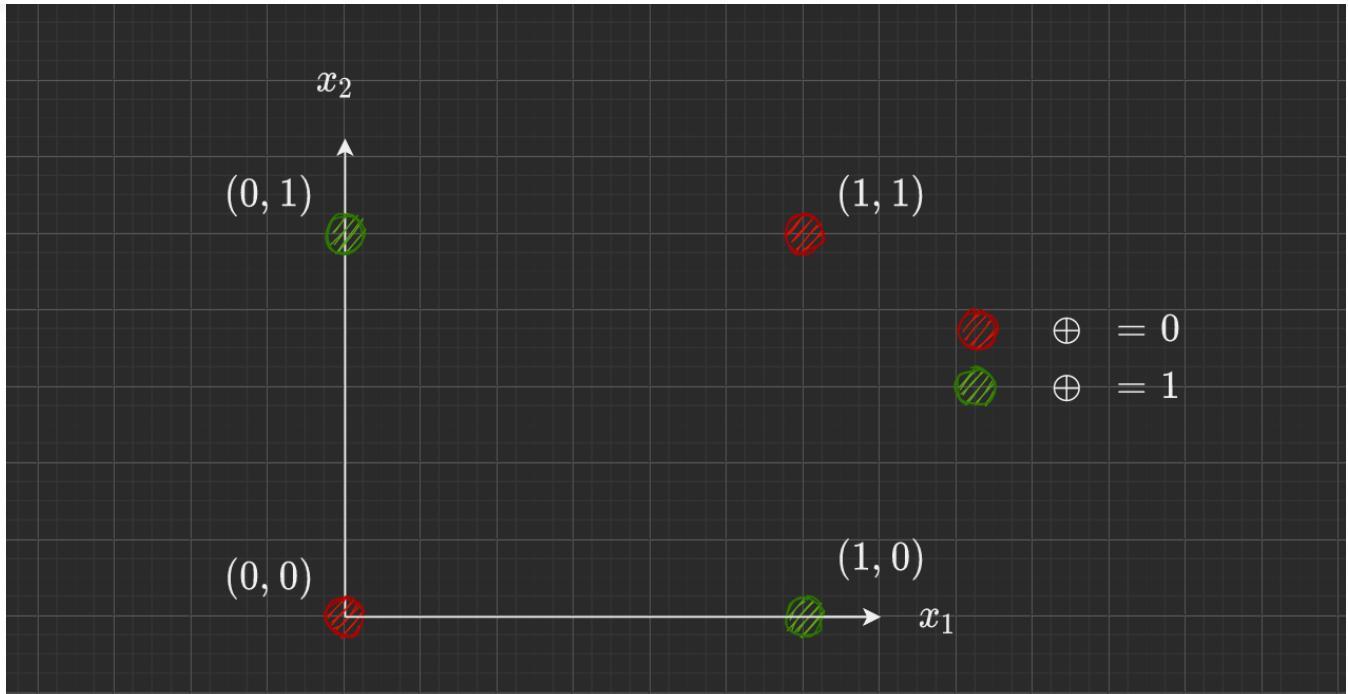
The function is defined like so:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

The XOR Truth table — Image by Author

If we plot it, we get the following chart. This is what we're trying to classify. The \oplus ("o-plus") symbol you see in the legend is conventionally used to represent the XOR boolean operator.



[Open in app](#)[Get started](#)

The XOR output plot — Image by Author using [draw.io](#)

Our algorithm —regardless of how it works — must correctly output the XOR value for each of the 4 points. We'll be modelling this as a classification problem, so `Class 1` would represent an XOR value of 1, while `Class 0` would represent a value of 0.

Attempt #1: The Single Layer Perceptron

Let's model the problem using a single layer perceptron.

Input data

The data we'll train our model on is the table we saw for the XOR function.

Data	Target
[0, 0]	0
[0, 1]	1
[1, 0]	1
[1, 1]	0

Implementation



[Open in app](#)[Get started](#)

Apart from the usual visualization (`matplotlib` and `seaborn`) and numerical libraries (`numpy`), we'll use `cycle` from `itertools`. This is done since our algorithm cycles through our data indefinitely until it manages to correctly classify the entire training data without any mistakes in the middle.

The data

We next create our training data. This data is the same for each kind of logic gate, since



[Open in app](#)[Get started](#)

The training function

Here, we cycle through the data indefinitely, keeping track of how many consecutive datapoints we correctly classified. If we manage to classify everything in one stretch, we terminate our algorithm.

If not, we reset our counter, update our weights and continue the algorithm.



[Open in app](#)[Get started](#)

[Open in app](#)[Get started](#)

To visualize how our model performs, we create a mesh of datapoints, or a grid, and evaluate our model at each point in that grid. Finally, we colour each point based on how our model classifies it. So the `class 0` region would be filled with the colour assigned to points belonging to that class.



[Open in app](#)[Get started](#)

To bring everything together, we create a simple `Perceptron` class with the functions we just discussed. We have some instance variables like the training data, the target, the number of input nodes and the learning rate.

Results

Let's create a perceptron object and train it on the XOR data.



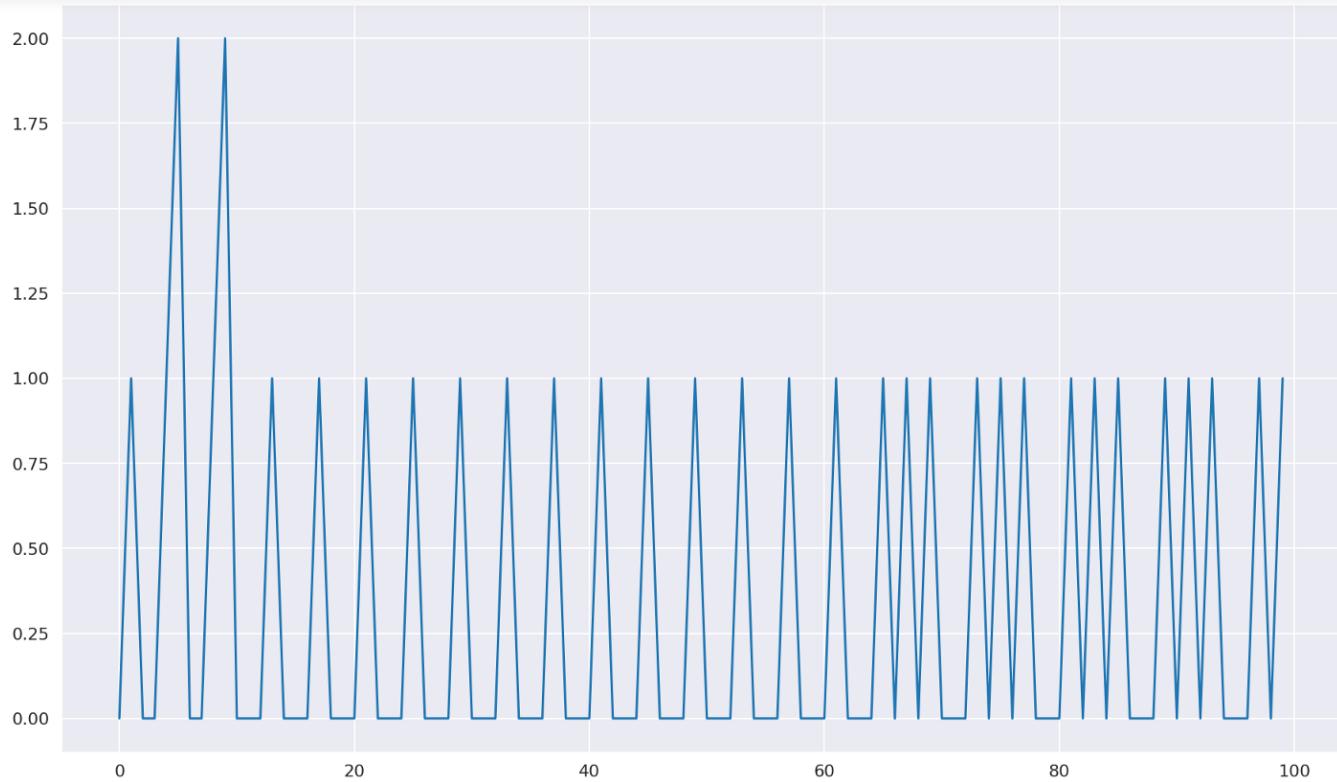
[Open in app](#)[Get started](#)

You'll notice that the training loop never terminates, since a perceptron can only converge on linearly separable data. Linearly separable data basically means that you can separate data with a point in 1D, a line in 2D, a plane in 3D and so on.

A perceptron can only converge on linearly separable data. Therefore, it isn't capable of imitating the XOR function.

Remember that a perceptron must correctly classify the entire training data in one go. If we keep track of how many points it correctly classified consecutively, we get something like this.



[Open in app](#)[Get started](#)

The value of correct_counter over 100 cycles of training — Image by Author

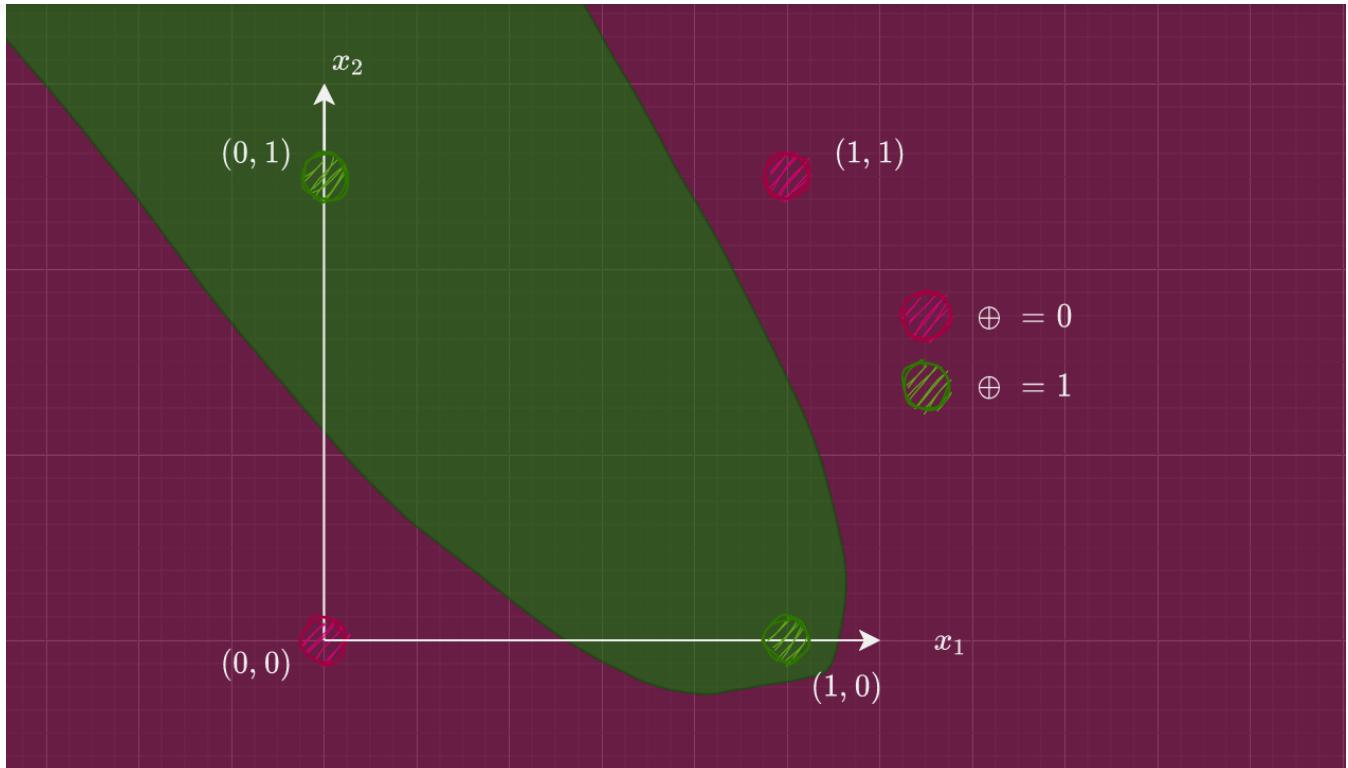
The algorithm only terminates when `correct_counter` hits 4 — which is the size of the training set — so this will go on indefinitely.

The Need for Non-Linearity

It is clear that a single perceptron will not serve our purpose: the classes aren't linearly separable. This boils down to the fact that a single linear decision boundary isn't going to work.

Non-linearity allows for more complex decision boundaries. One potential decision boundary for our XOR data could look like this.




[Open in app](#)
[Get started](#)


A potential non-linear decision boundary for our XOR model — Image by Author using [draw.io](#)

The 2d XOR problem — Attempt #2

We know that the imitating the XOR function would require a non-linear decision boundary.

But why do we have to stick with a single decision boundary?

The Intuition

Let's first break down the XOR function into its AND and OR counterparts.

The XOR function on two boolean variables A and B is defined as:

$$\text{XOR}(A, B) = A \cdot \overline{B} + B \cdot \overline{A}$$



[Open in app](#)[Get started](#)

$$XOR(A, B) = A \cdot \overline{B} + B \cdot \overline{A} + (A \cdot \overline{A} + B \cdot \overline{B})$$

Let's rearrange the terms so that we can pull out \overline{A} from the first part and \overline{B} from the second.

$$XOR(A, B) = (A \cdot \overline{A} + A \cdot \overline{B}) + (B \cdot \overline{A} + B \cdot \overline{B})$$

Simplifying it further, we get:

$$XOR(A, B) = (A + B) \cdot (\overline{A} + \overline{B})$$

Using DeMorgan's laws for boolean algebra: $\sim A + \sim B = \sim(AB)$, we can replace the second term in the above equation like so:

$$XOR(A, B) = (A + B) \cdot (\overline{AB})$$

Let's replace A and B with x_1 and x_2 respectively since that's the convention we're using in our data.

$$XOR(x_1, x_2) = (x_1 + x_2) \cdot (\overline{x_1 x_2})$$



[Open in app](#)[Get started](#)

Modelling the OR part

We'll use the same Perceptron class as before, only that we'll train it on OR training data.

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

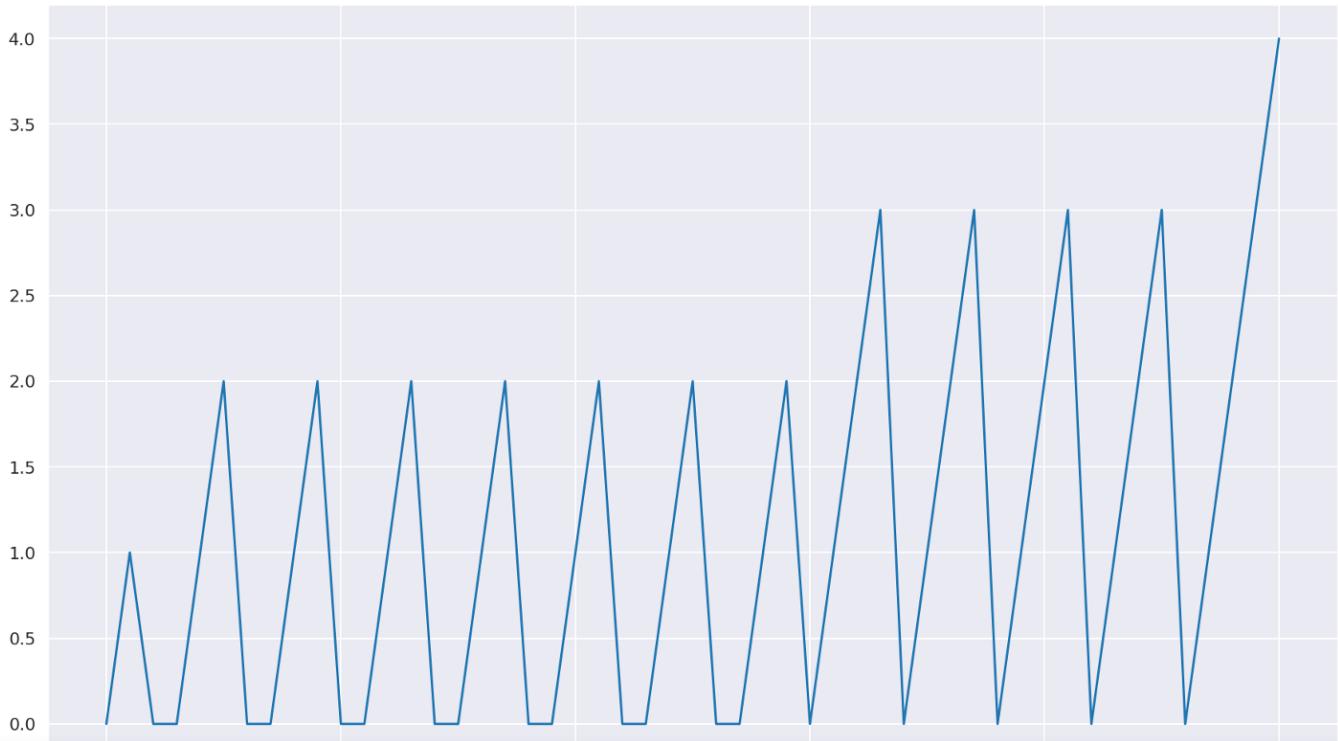
The OR truth table — Image by Author using [draw.io](#)

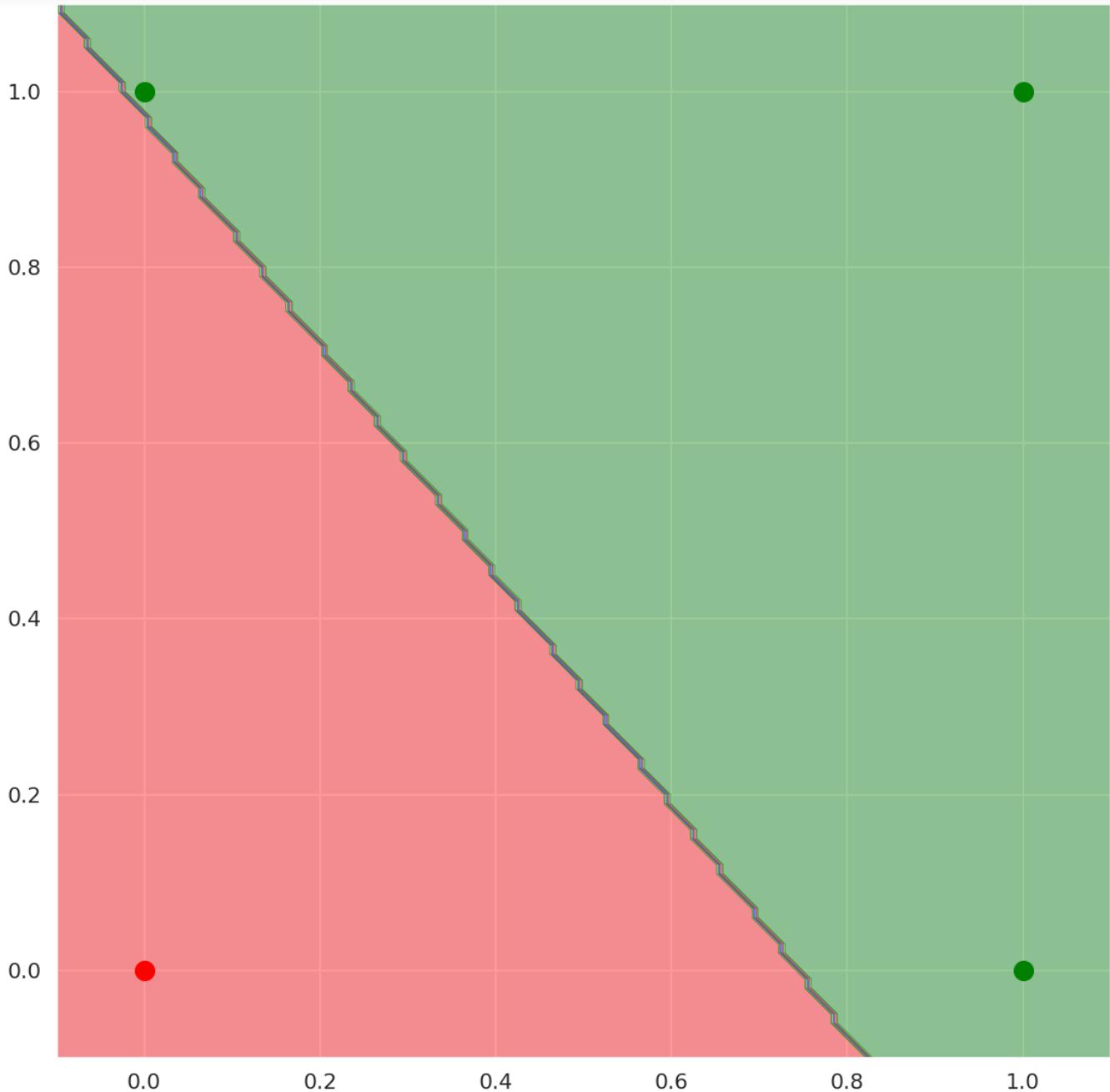


[Open in app](#)[Get started](#)

This converges, since the data for the OR function is linearly separable. If we plot the number of correctly classified consecutive datapoints as we did in our first attempt, we get this plot. It's clear that around iteration 50, it hits the value 4, meaning that it classified the entire dataset correctly.

`correct_counter` measures the number of consecutive datapoints correctly classified by our Perceptron



[Open in app](#)[Get started](#)

The Output plot of our OR perceptron — Image by Author

Modelling the NAND part

Let's move on to the second part. We need to model a NAND gate. Just like the OR part, we'll use the same code, but train the model on the NAND data. So our input data would be:

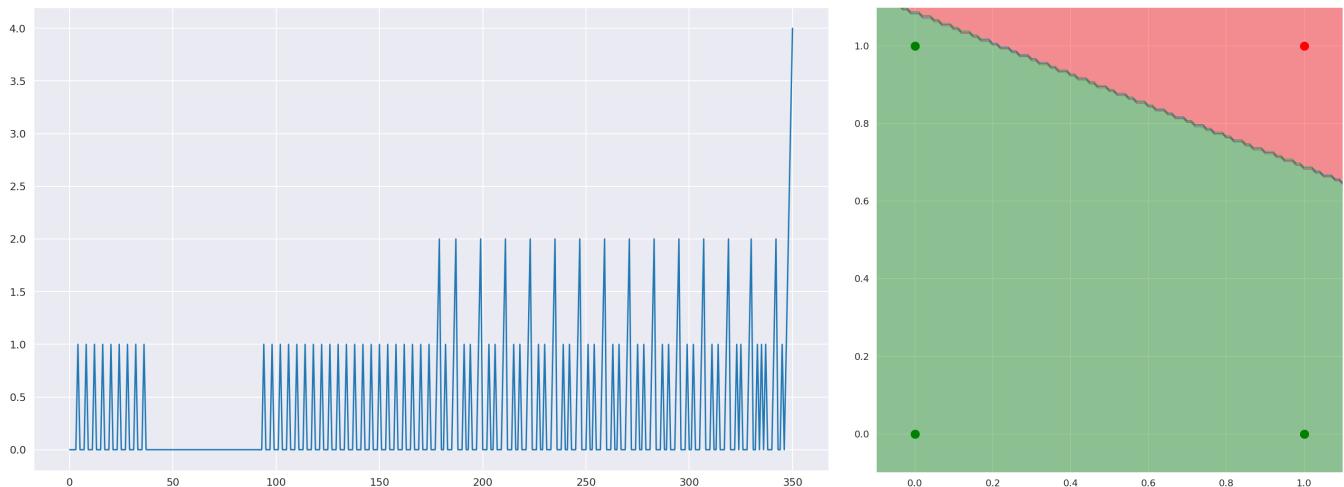


[Open in app](#)[Get started](#)

x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	0

The NAND Truth table — Image by Author using [draw.io](#)

After training, the following plots show that our model converged on the NAND data and mimics the NAND gate perfectly.



Decision boundary and correct_counter plots for the NAND perceptron — Image by Author

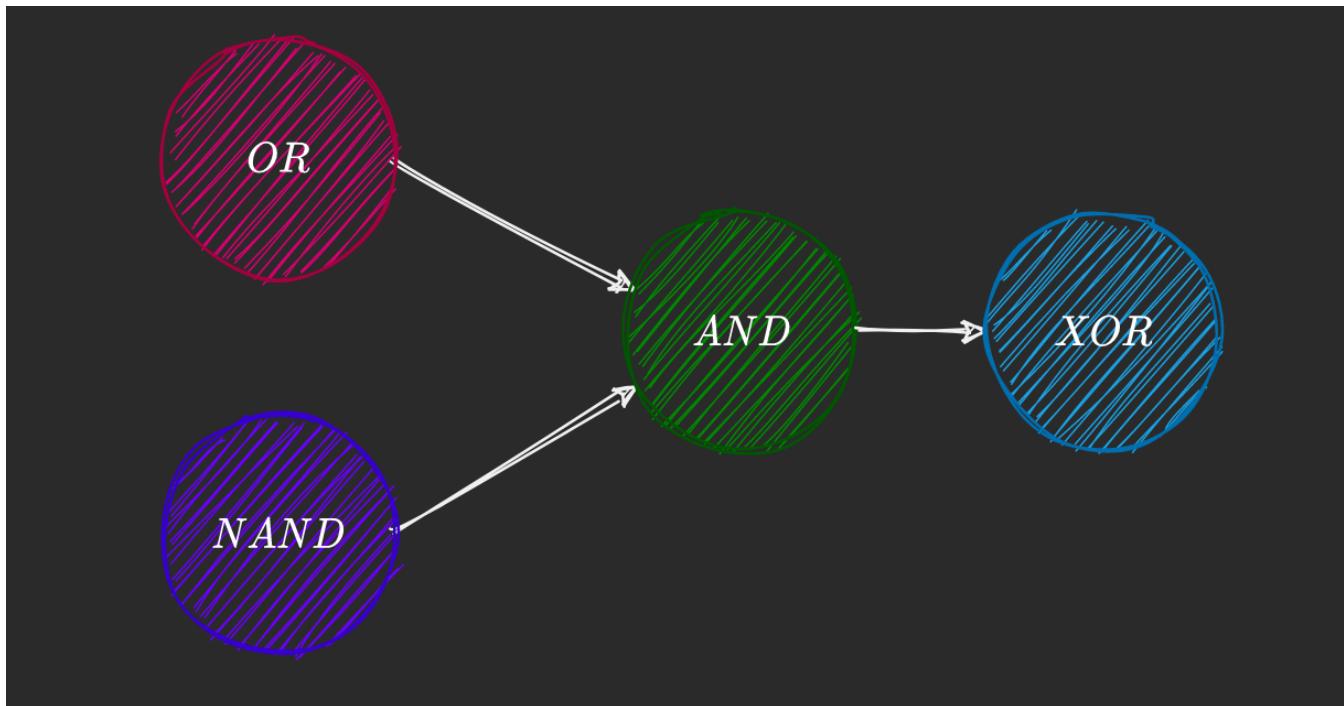
Bringing everything together



[Open in app](#)[Get started](#)

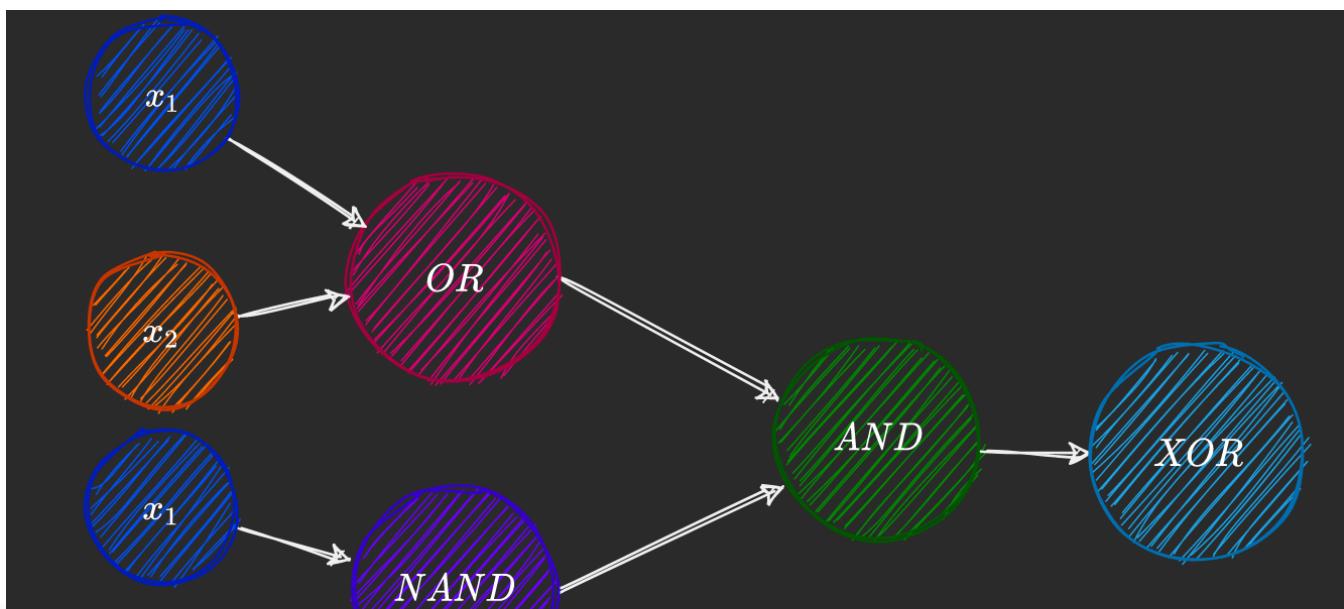
- and that both functions are being passed the same input (x_1 and x_2).

Let's model this into our network. First, let's consider our two perceptrons as black boxes.



The plan for our model — Image by Author using [draw.io](#)

After adding our input nodes x_1 and x_2 , we can finally implement this through a simple function.

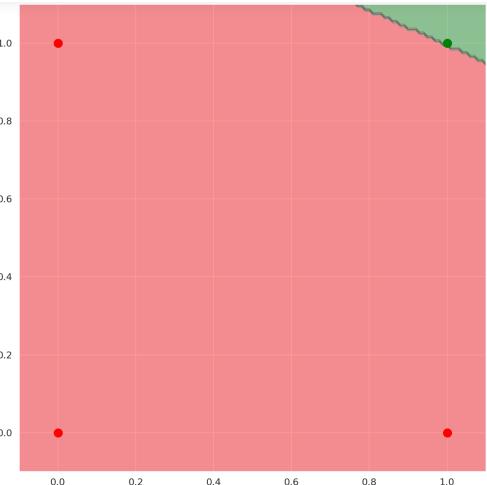
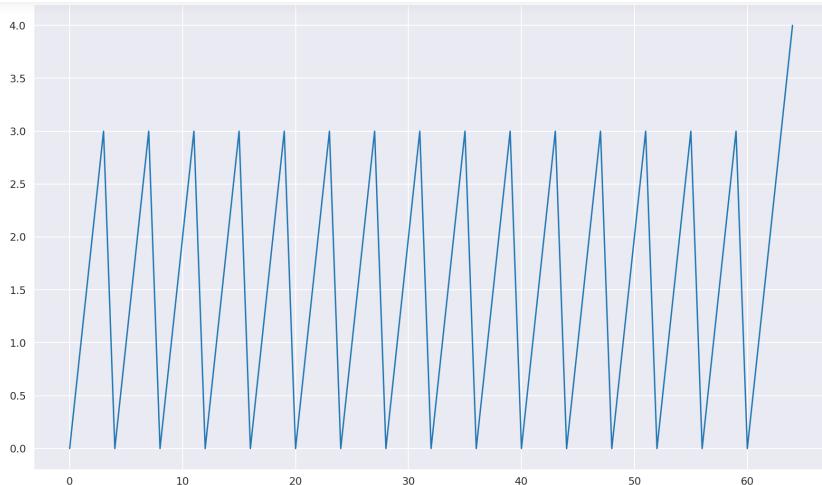


[Open in app](#)[Get started](#)

Adding input nodes — Image by Author using [draw.io](#)

Finally, we need an AND gate, which we'll train just we have been.



[Open in app](#)[Get started](#)

The correct_count and output plots of our AND perceptron. — Image by Author

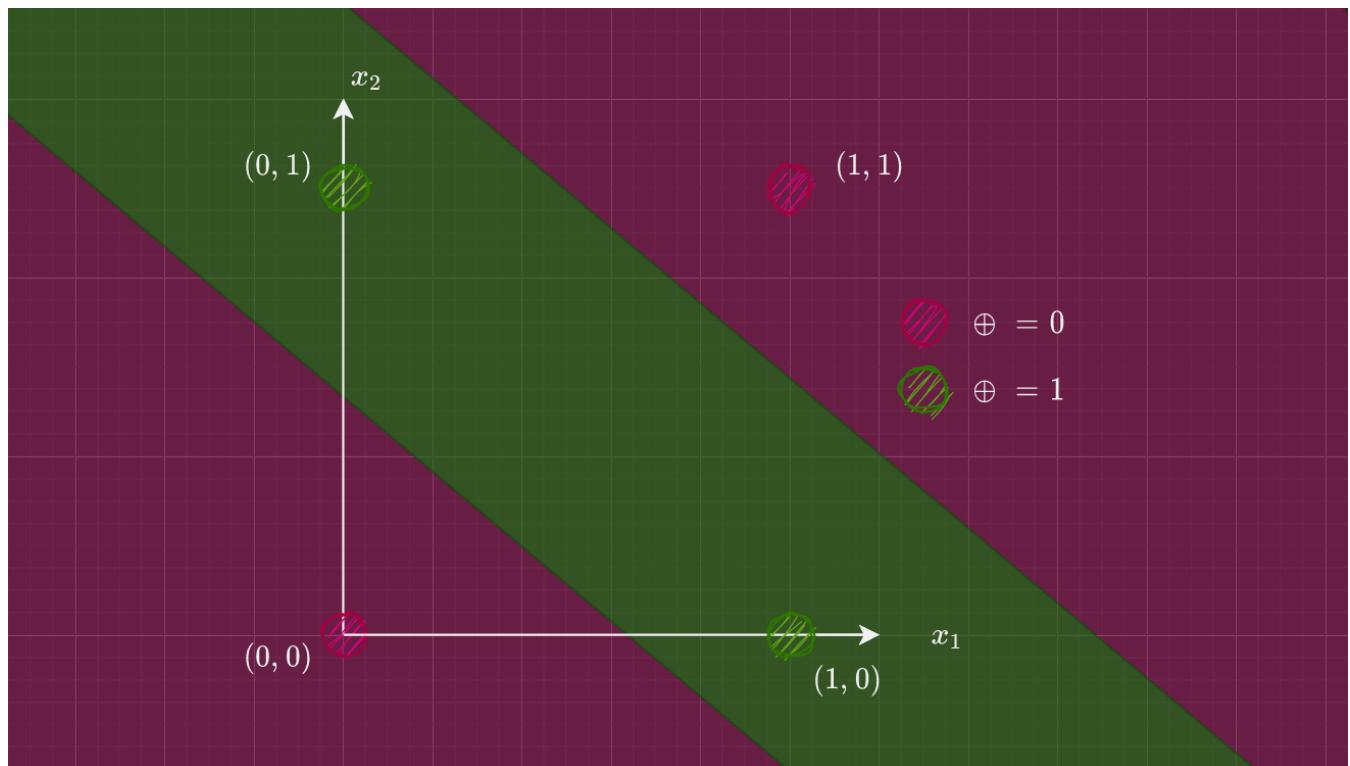
What we now have is a model that mimics the XOR function.

If we were to implement our XOR model, it would look something like this:



[Open in app](#)[Get started](#)

If we plot the decision boundaries from our model, — which is basically an AND of our OR and NAND models — we get something like this:

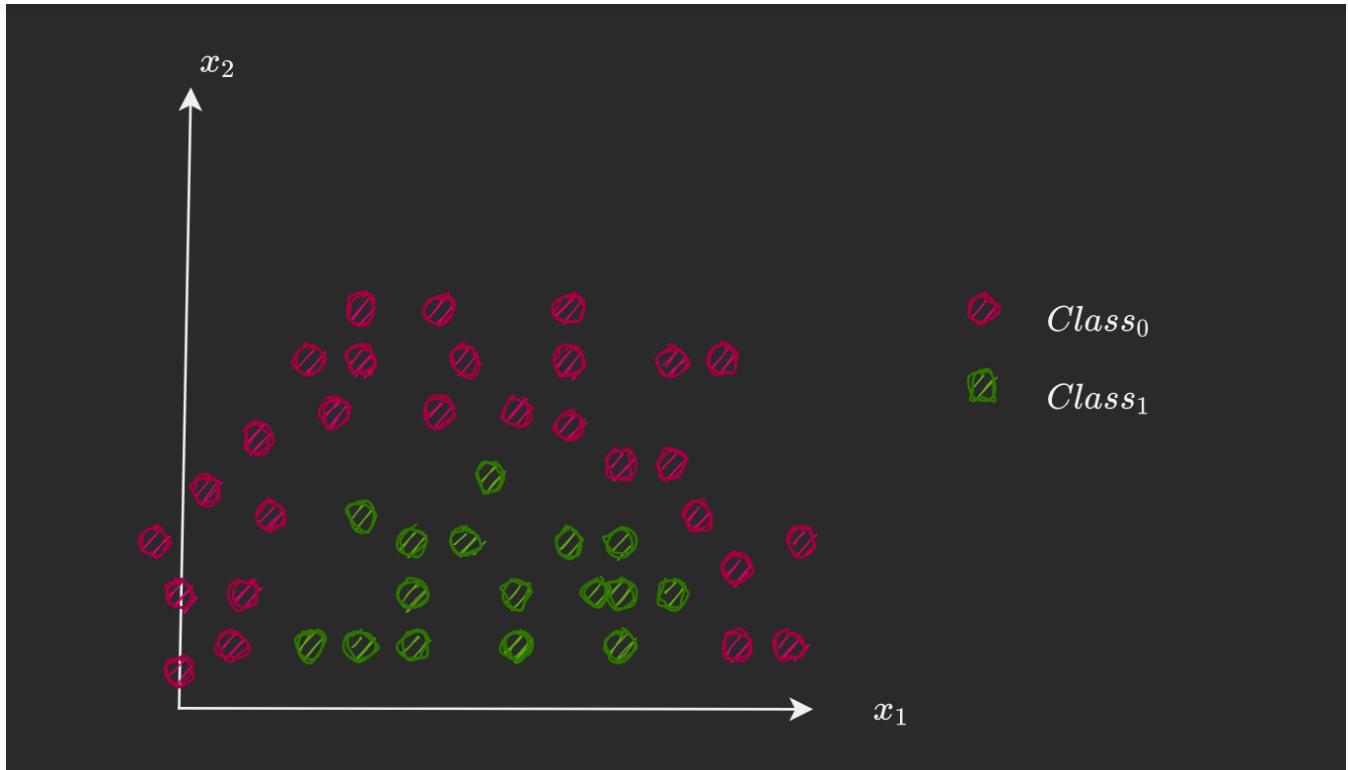


The Output plot of our 2nd Attempt, showing a correct classification on our XOR data— Image by Author using [draw.io](#)

Out of all the 2 input logic gates, the XOR and XNOR gates are the only ones that are not linearly-separable.

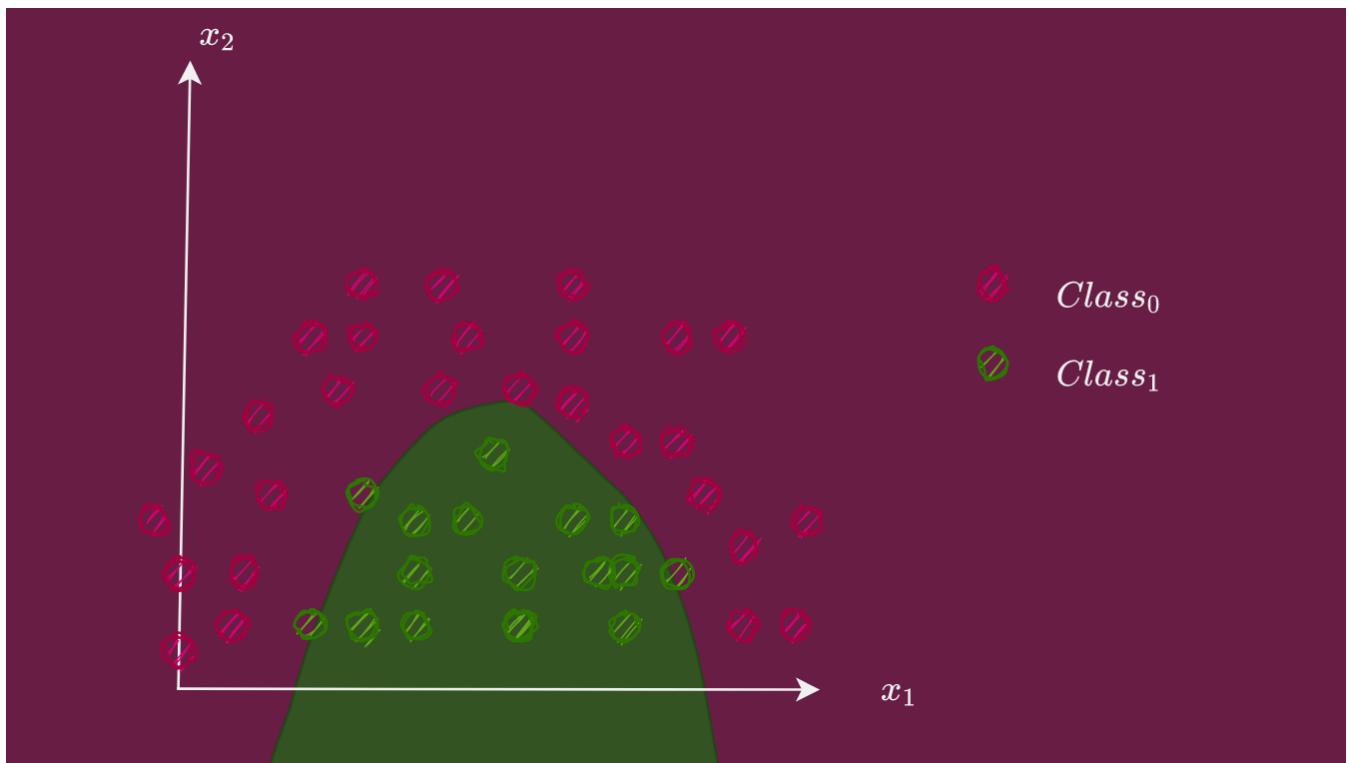
Though our model works, it doesn't seem like a viable solution to most non-linear classification or regression tasks. It's really specific to this case, and most problems can't be split into just simple intermediate problems that can be individually solved and then combined. For something like this:



[Open in app](#)[Get started](#)

A binary classification problem in two dimensions — Image by Author using [draw.io](#)

A potential decision boundary could be something like this:



A potential decision boundary that fits our example — Image by Author using [draw.io](#)



[Open in app](#)[Get started](#)

The Multi-layered Perceptron

The overall components of an MLP like input and output nodes, activation function and weights and biases are the same as those we just discussed in a perceptron.

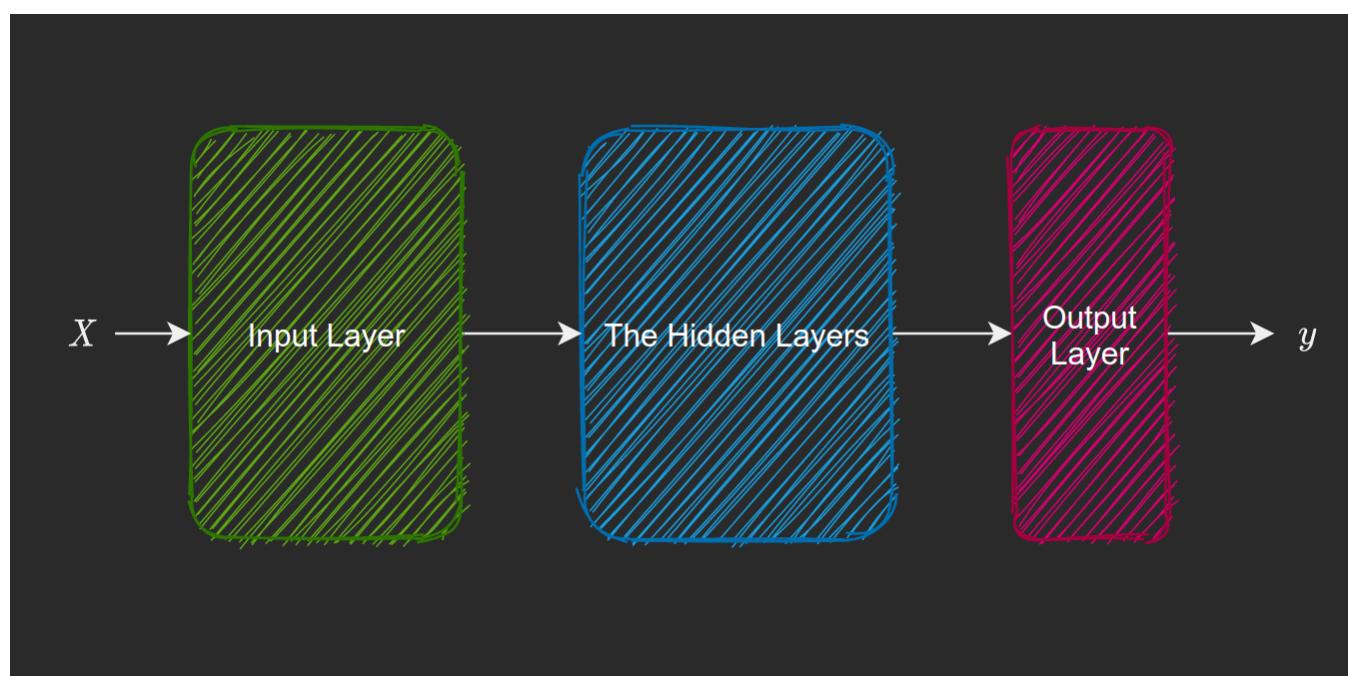
The biggest difference? An MLP can have hidden layers.

Hidden layers

Hidden layers are those layers with nodes other than the input and output nodes.

An MLP is generally restricted to having a single hidden layer.

The **hidden layer allows for non-linearity**. A node in the hidden layer isn't too different to an output node: nodes in the previous layers connect to it with their own weights and biases, and an output is computed, generally with an activation function.



The general structure of a multi-layered perceptron — Image by Author using [draw.io](#)



[Open in app](#)[Get started](#)

the `sigmoid` and the `tanh` functions, which are some of the most popular non-linear activation functions.

Activation functions should be differentiable, so that a network's parameters can be updated using backpropagation.

Training algorithm

Though the output generation process is a direct extension of that of the perceptron, updating weights isn't so straightforward. Here's where backpropagation comes into the picture.

Backpropagation is a way to update the weights and biases of a model starting from the output layer all the way to the beginning. The main principle behind it is that each parameter changes in proportion to how much it affects the network's output. A weight that has barely any effect on the output of the model will show a very small change, while one that has a large negative impact will change drastically to improve the model's prediction power.

Backpropagation is an algorithm for update the weights and biases of a model based on their gradients with respect to the error function, starting from the output layer all the way to the first layer.

The method of updating weights directly follows from derivation and the chain rule.

There's a lot to cover when talking about backpropagation. It warrants its own article. So if you want to find out more, have a look at this excellent article by [Simeon Kostadinov](#).

Understanding Backpropagation Algorithm



[Open in app](#)[Get started](#)

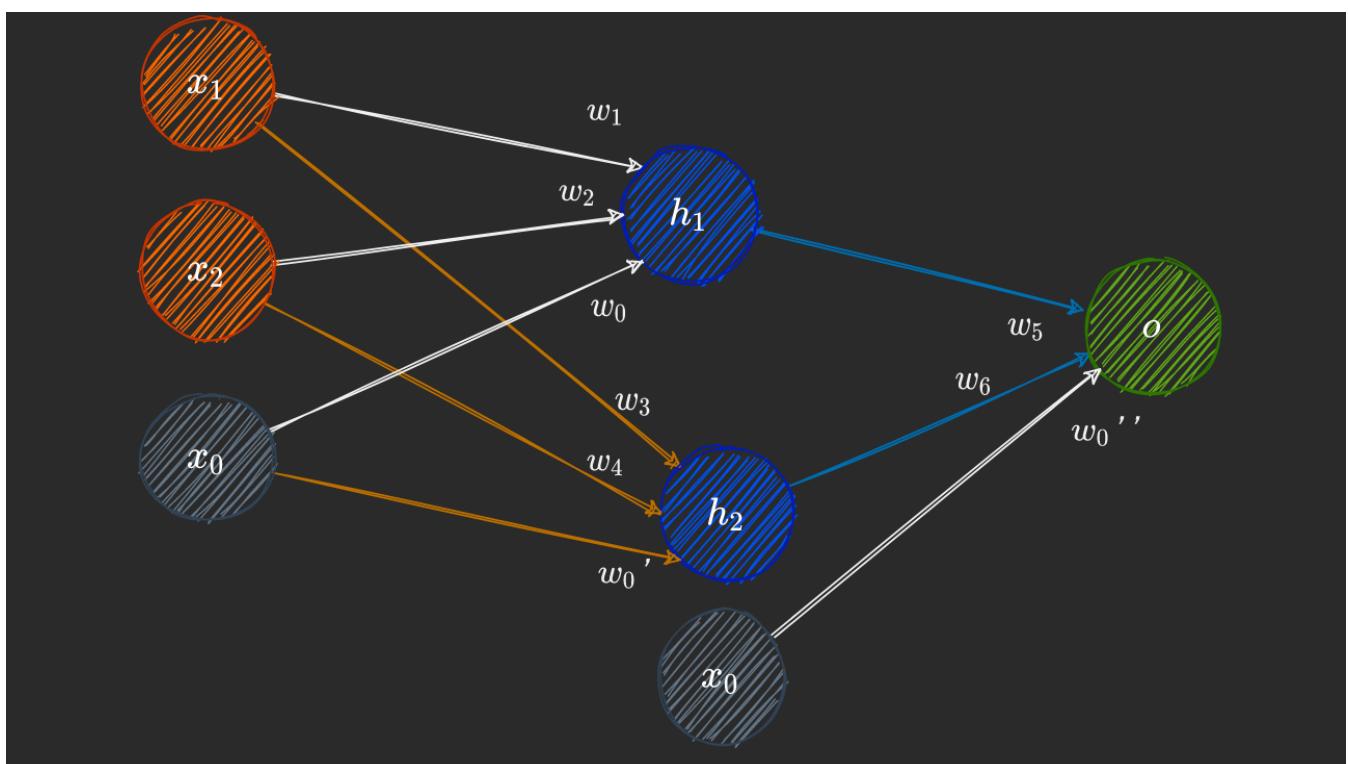
Attempt #3: the Multi-layered Perceptron

The architecture

There are no fixed rules on the number of hidden layers or the number of nodes in each layer of a network. The best performing models are obtained through trial and error.

The architecture of a network refers to its general structure — the number of hidden layers, the number of nodes in each layer and how these nodes are inter-connected.

Let's go with a single hidden layer with two nodes in it. We'll be using the sigmoid function in each of our hidden layer nodes and of course, our output node.



[Open in app](#)[Get started](#)

The libraries used here like NumPy and pyplot are the same as those used in the Perceptron class.

The training algorithm

The algorithm here is slightly different: we iterate through the training data a fixed number of times — `num_epochs` to be precise. In each iteration, we do a forward pass, followed by a backward pass where we update the weights and biases as necessary. This is called backpropagation.



[Open in app](#)[Get started](#)

Here, we define a sigmoid function. As discussed, it's applied to the output of each hidden layer node and the output node. Its differentiable, so it allows us to comfortably perform backpropagation to improve our model.

Its derivative is also implemented through the `_delsigmoid` function.

The forward and backward pass

In the forward pass, we apply the $wx + b$ relation multiple times, and applying a



[Open in app](#)[Get started](#)

In the backward pass, implemented as the `update_weights` function, we calculate the gradients of each of our 6 weights and 3 biases with respect to the error function and update them by the factor `learning_rate * gradient`.

Finally, the `classify` function works as expected: Since a sigmoid function outputs values between 0 and 1, we simply interpret them as probabilities of belonging to a particular class. Hence, outputs greater than or equal to 0.5 are classified as belonging to `Class 1` while those outputs that are less than 0.5 are said to belong to `Class 0`.



[Open in app](#)[Get started](#)

Let's bring everything together by creating an MLP class. All the functions we just discussed are placed in it. The `plot` function is exactly the same as the one in the Perceptron class.

Results

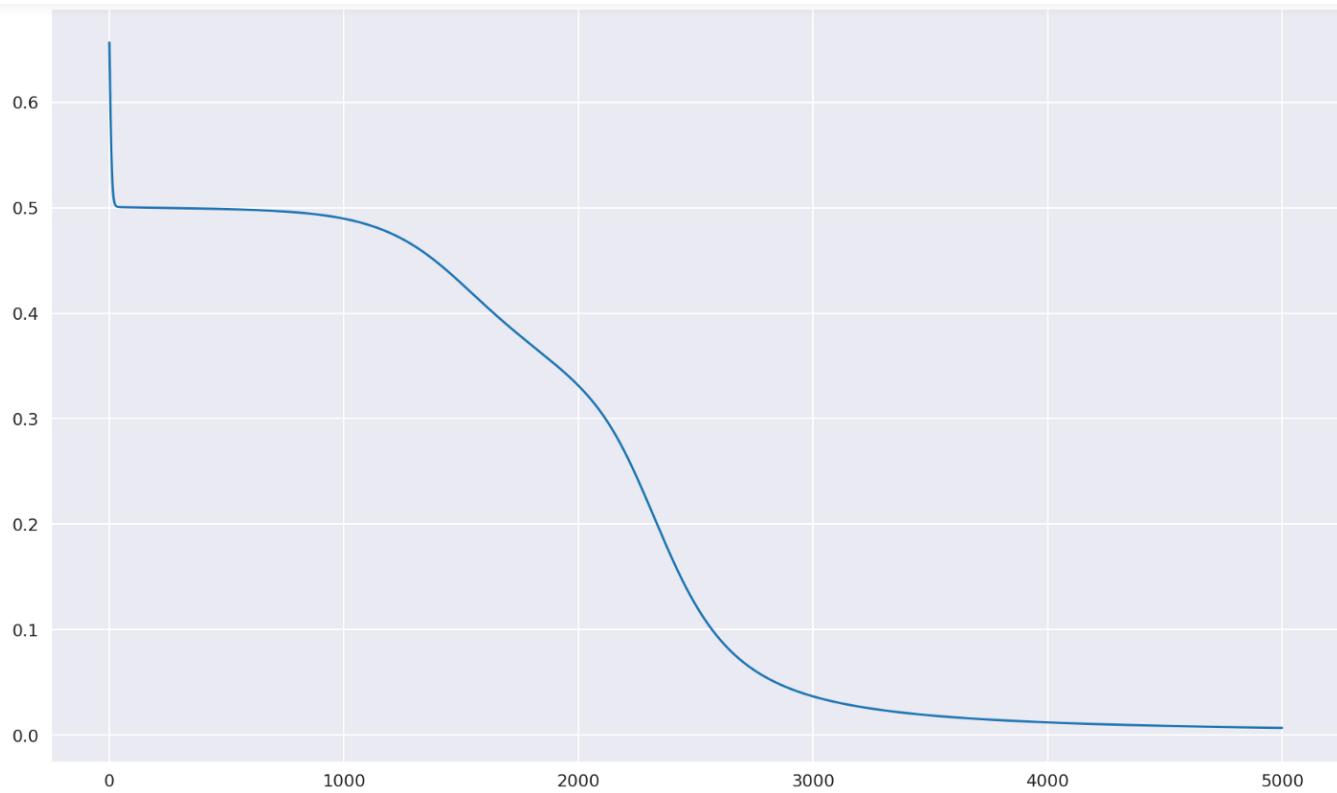
Let's train our MLP with a learning rate of `0.2` over `5000` epochs.



[Open in app](#)[Get started](#)

If we plot the values of our loss function, we get the following plot after about 5000 iterations, showing that our model has indeed converged.

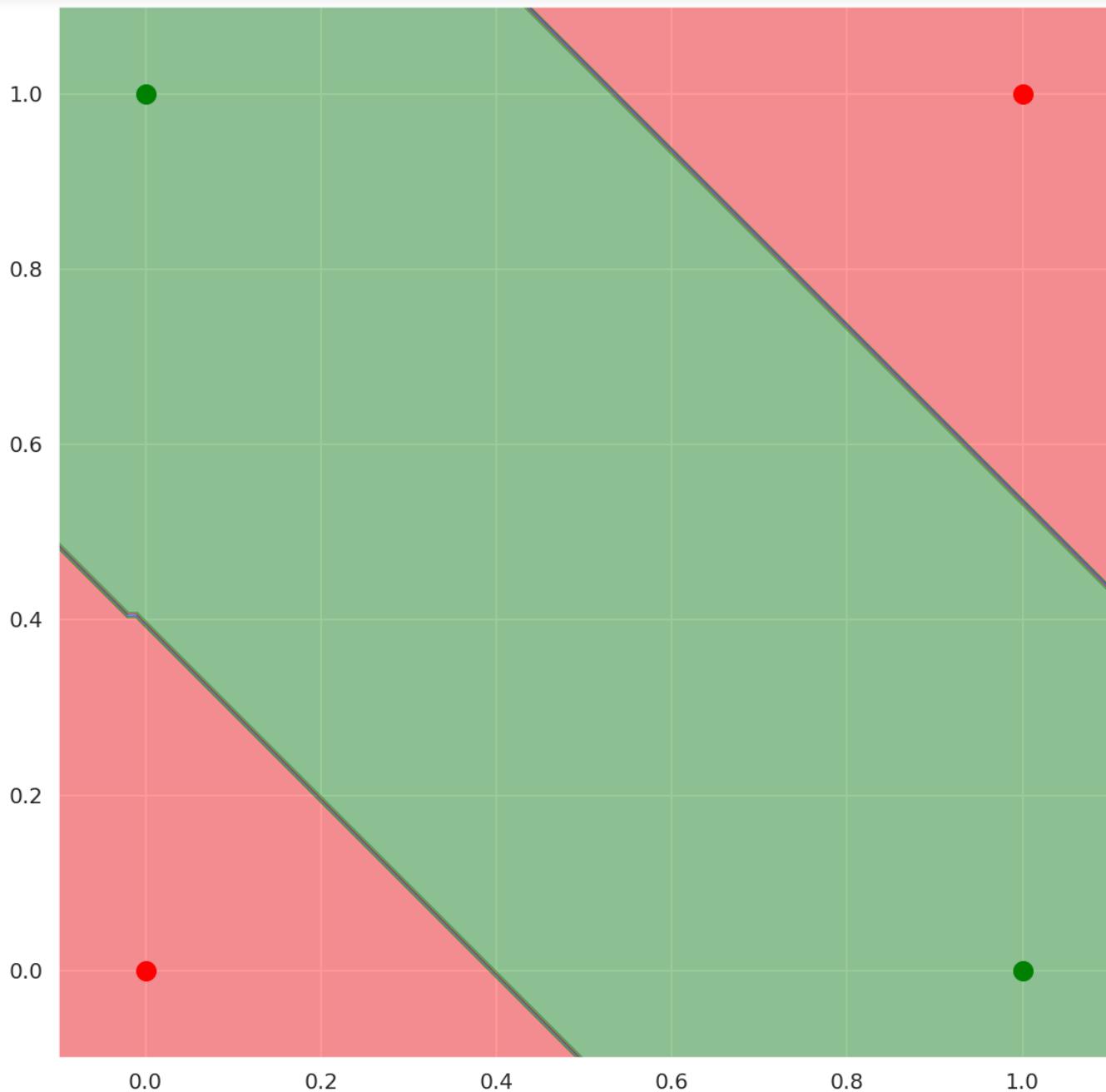


[Open in app](#)[Get started](#)

The Loss Plot over 5000 epochs of our MLP — Image by Author

A clear non-linear decision boundary is created here with our generalized neural network, or MLP.



[Open in app](#)[Get started](#)

The Decision Boundary plot, showing the decision boundary and the classes — Image by Author

Note #1: Adding more layers or nodes

Adding more layers or nodes gives increasingly complex decision boundaries. But this could also lead to something called overfitting — where a model achieves very high accuracies on the training data, but fails to generalize.

A good resource is the Tensorflow Neural Net playground, where you can try out different network architectures and view the results.



[Open in app](#)[Get started](#)playground.tensorflow.org

Note #2: Choosing a loss function

The loss function we used in our MLP model is the Mean Squared loss function. Though this is a very popular loss function, it makes some assumptions on the data (like it being gaussian) and isn't always convex when it comes to a classification problem. It was used here to make it easier to understand how a perceptron works, but for classification tasks, there are better alternatives, like **binary cross-entropy loss**.

How to Choose Loss Functions When Training Deep Learning Neural Networks - Machine Learning Mastery

Deep learning neural networks are trained using the stochastic gradient descent optimization algorithm. As part of the...

[machinelearningmastery.com](https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/)

The Code

You'll find the entire code from this post [here](#).

Polaris000/BlogCode/xorperceptron.ipynb

The sample code from this post can be found [here](#).

[github.com](https://github.com/Polaris000/BlogCode/xorperceptron.ipynb)

Conclusion

Neural nets used in production or research are never this simple, but they almost



[Open in app](#)[Get started](#)

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. [Take a look.](#)

[Get this newsletter](#)

[About](#) [Help](#) [Terms](#) [Privacy](#)

Get the Medium app

