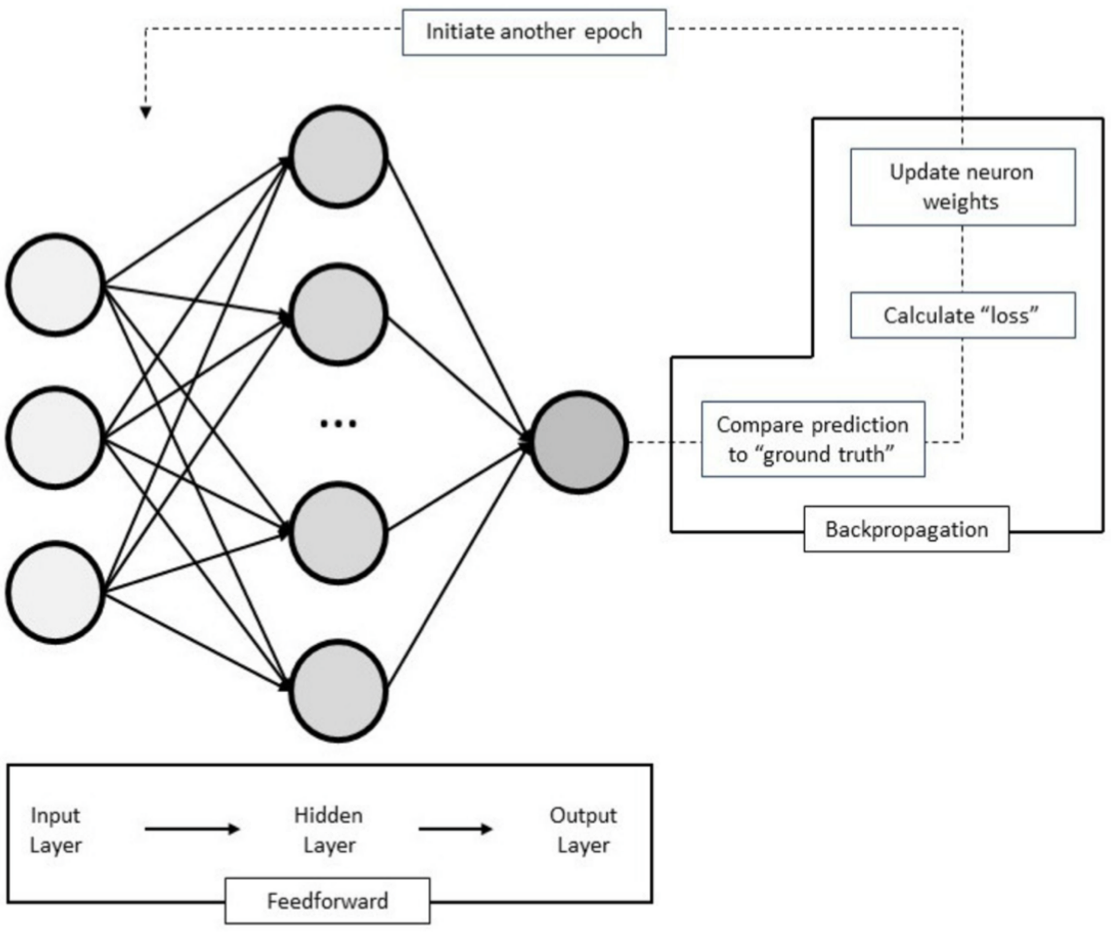
Cristian Sirbu

Report

Acquired Intelligence and Adaptive Behaviour

The field of supervised Machine Learning (ML) includes Artificial Neural Networks (ANN). ANNs have a structure similar to the human brain. Each node represents a neuron, and the connections between them function in the same way that brain synapses do. The significance of the represented feature to the value of the dependent variable is quantified by assigning a random weight to each neuron at the start. Through an iterative training process, these weights are constantly updated and ultimately optimized (epoch). The contribution of each neuron to the dependent variable, as well as the overall accuracy of the ANN algorithm prediction against a known value, are evaluated in each training loop. The foundation of supervised learning is the process of comparing network predictions to known values (the ground truth). The use of a loss function at the end of each epoch is required to enable this internal assessment and feedback process. In the diagram below, the feedforward, backpropagation training process is illustrated schematically.



Hidden layers are important to the performance of neural networks, particularly in complex problems where accuracy and time complexity are the primary constraints. The process of determining the number of hidden layers and neurons in each hidden layer remains perplexing. Techniques that used fewer than three hidden layers suffered from a loss of accuracy, while architectures that used more than three hidden layers were found to be inefficient in terms of time complexity. In terms of time complexity and accuracy, implementing three numbers of hidden layers usually yields the best results.

When do we need hidden layers?

If the problem is linear there is not necessary to use hidden layers because using a linear regression is more effective. But, in case your data is not linear and its spread in a weird way then the linear regression is not going to work. This is exactly when you need hidden layers. Now, the question would be how many hidden layers would you need? The right answer for this question would be whatever gives you the best fit in solving the problem, just the right number of hidden layers. The numbers of hidden layers are proportional with the numbers of parameters in the data set. In the average an added hidden layer to the neural network contains 4 neurons but in case we use more neurons the loss curve looks better. Even if there are more than enough hidden layers the problem will be solved excellent but when it comes to testing data or other data sets the result would not be as good plus the time complexity will exceed which means we are overfitting by adding many hidden layers. From the readings and testing different data for this report I draw up this schema:

0 layers – Only capable of representing linear separable functions or decisions.

1 layer – Can approximate any function that contains a continuous mapping from one finite space to another. For example, if you plot x and y and you have one class clustered in the middle and another class somewhere near or around the previous class you can use a hidden layer to get decent results from that type of data spread.

2 layers – Can represent an arbitrary decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any accuracy.

More than 2 – Additional layers can learn complex representations but rare scenarios.

How many neurons a hidden layer should contain?

Too few neurons in the hidden layers will result in underfitting, which means that they cannot model complicated data sets while **too many neurons** may result in overfitting which means that the accuracy might be high but once we change the data set it will drop instantly.

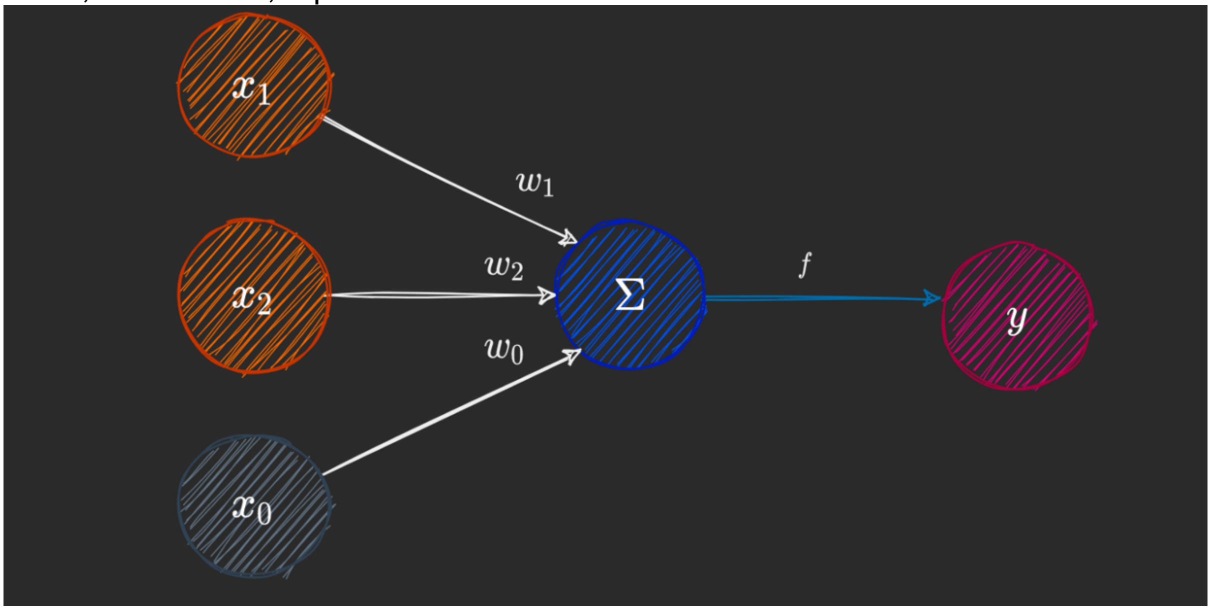
Here are some guidelines which might help you define how many nodes are necessary:

- In order to secure the ability of the network to generalize the number of nodes has to be kept as low as possible to avoid overfitting.
- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output.
- The number of hidden neurons should be less than twice the size of the input layer.

Do not forget, the selection of an architecture for your neural network will come down to a trial and error.

XOR Gate:

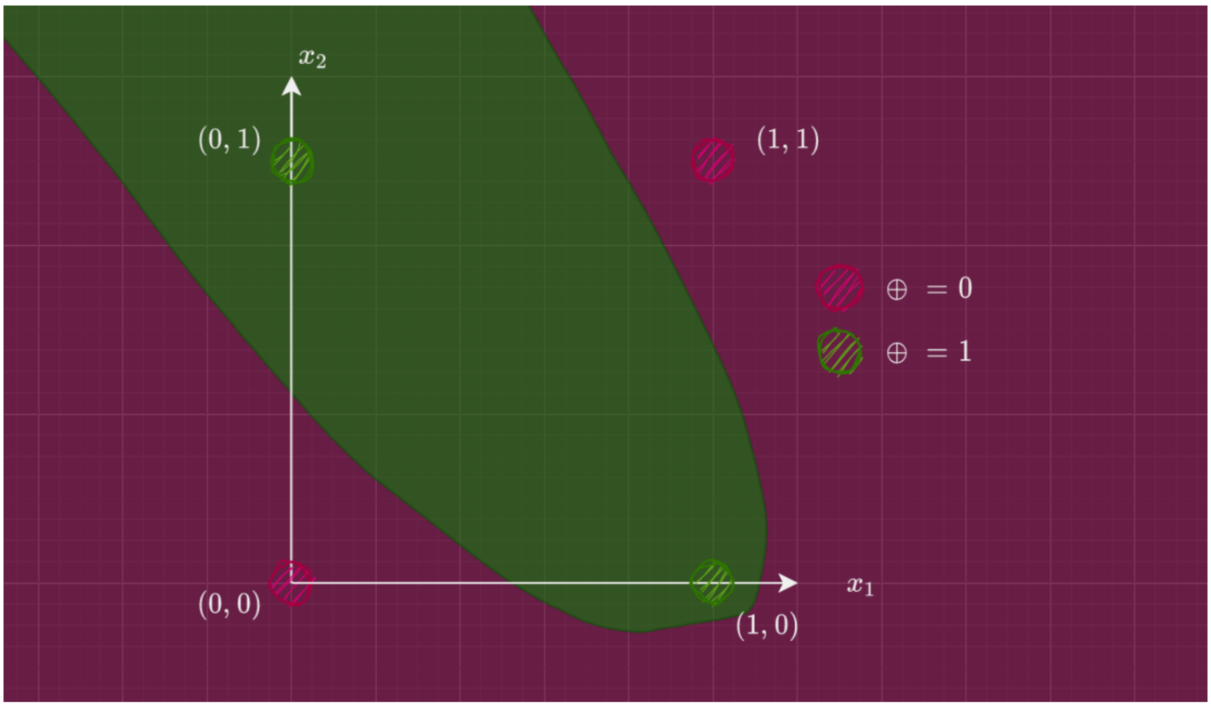
XOR Gate problem can be solved by one layer of neurons that's why we have to add **a hidden layer of neurons or a non-linear decision boundary**. Trying to solve the problem with a single layer you'll notice that the training loop never terminates, since a perceptron can only converge on linearly separable 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 can't imitate the XOR function.

A single perceptron will clearly not suffice because the classes aren't linearly separable. This boils down to the fact that there isn't going to be a single linear decision boundary that will work.

Non-linearity enables more complicated decision boundaries to be used. This could be one possible decision threshold for our XOR data.



We already know that the imitating the XOR function would require a non-linear decision boundary. So, let's first break down the XOR function into its AND and OR counterparts.

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

We can add notA and notB to the equation because they equal to 0:

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

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

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

Boolean laws tells us that notA + notB = not(AB) =>

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

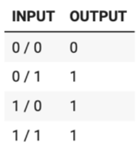
The XOR function can be condensed into two parts: a NAND and an OR. If we can calculate these separately, we can just combine the results, using an AND gate.

Cristian Sirbu

Report

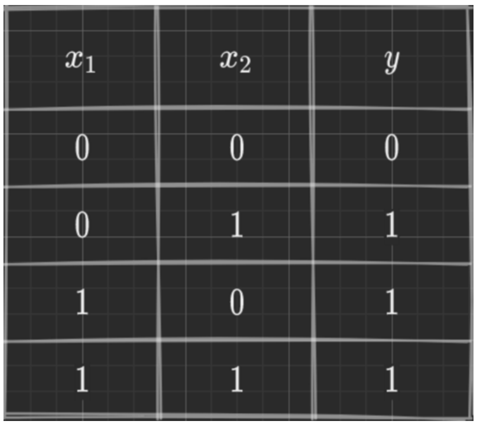
Let’s call the OR section of the formula part I, and the NAND section as part II.

The OR gate is defined as follows:



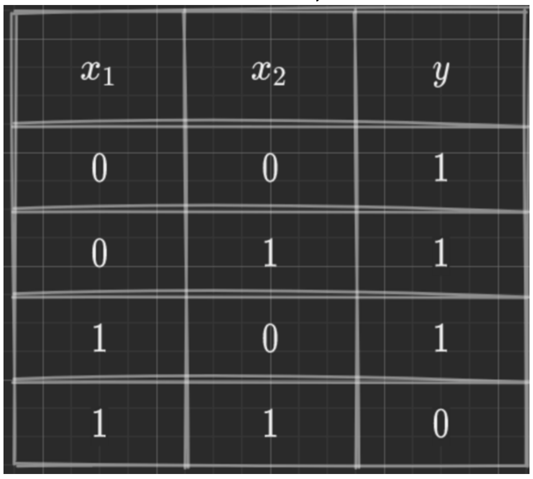
INPUT	OUTPUT
0 / 0	0
0 / 1	1
1 / 0	1
1 / 1	1

```
# Define our weights and biases (todo)
W = torch.tensor([[2.], [2.]])
b = torch.tensor([[-1.]])
assert list(W.size()) == [2, 1], f"Weights are incorrect size
({W.size()})"
assert list(b.size()) == [1, 1], f"Biases are incorrect size
({b.size()})"

# Activation function
def f(inp):
    inp[inp >= 0] = 1
    inp[inp < 0] = 0
    return inp

# Loop over each example
for i in range(input_data.size(0)):
    # Get example `i` (and unsqueeze to [1, 2] and [1, 1])
    x = input_data[i].unsqueeze(0)
    y = output_data[i].unsqueeze(0)

    # Predict output
    y_hat = f(torch.mm(x, W) + b)

    # Check predictions are correct
    print(f"Prediction {y_hat}, desired output {y}")
    assert (y == y_hat), f"{y_hat} does not equal {y}"
```



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

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:

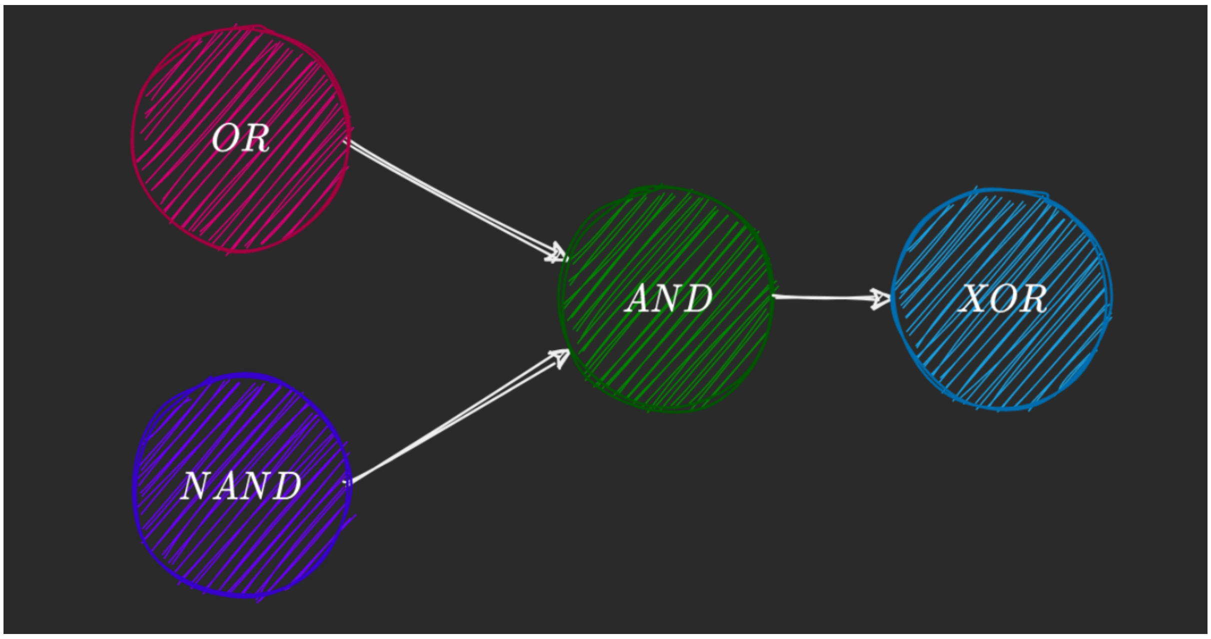


x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	0

Bringing everything together

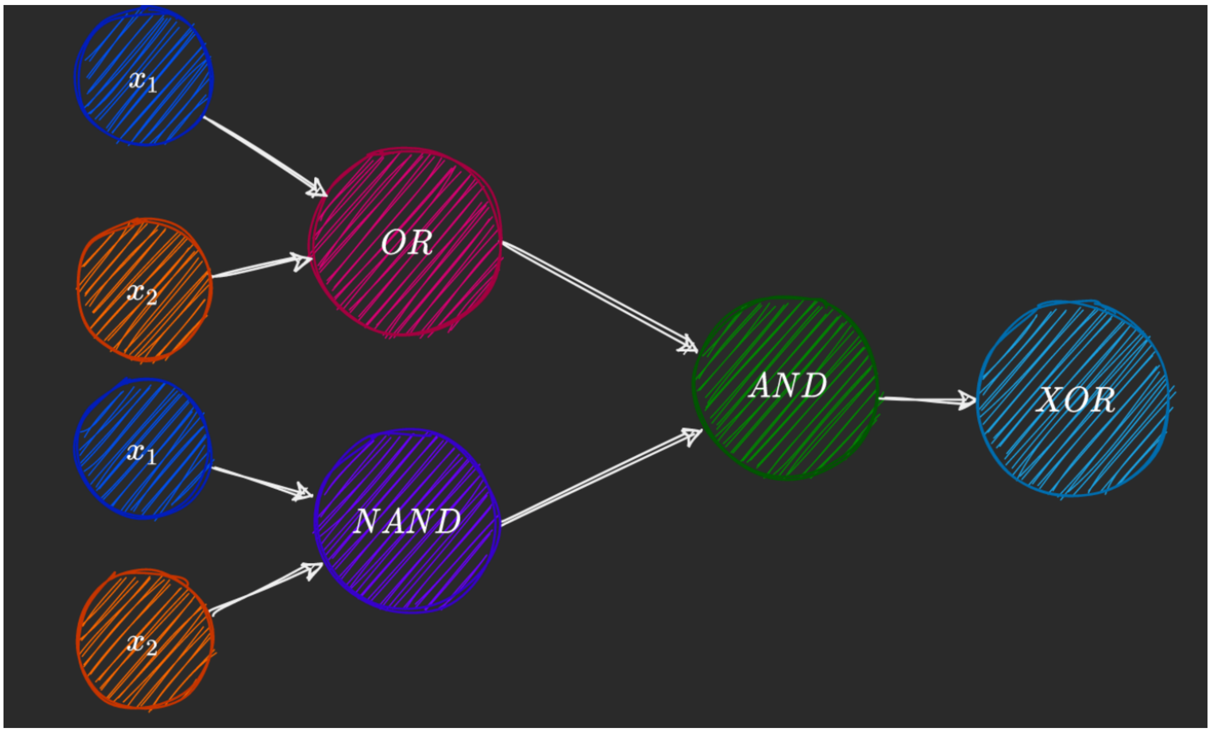
Two things are clear from this:

- we are performing a logical AND on the outputs of two logic gates (where the first one is an OR and the second one a NAND)
- and that both functions are being passed the same input (x1 and x2).



Report

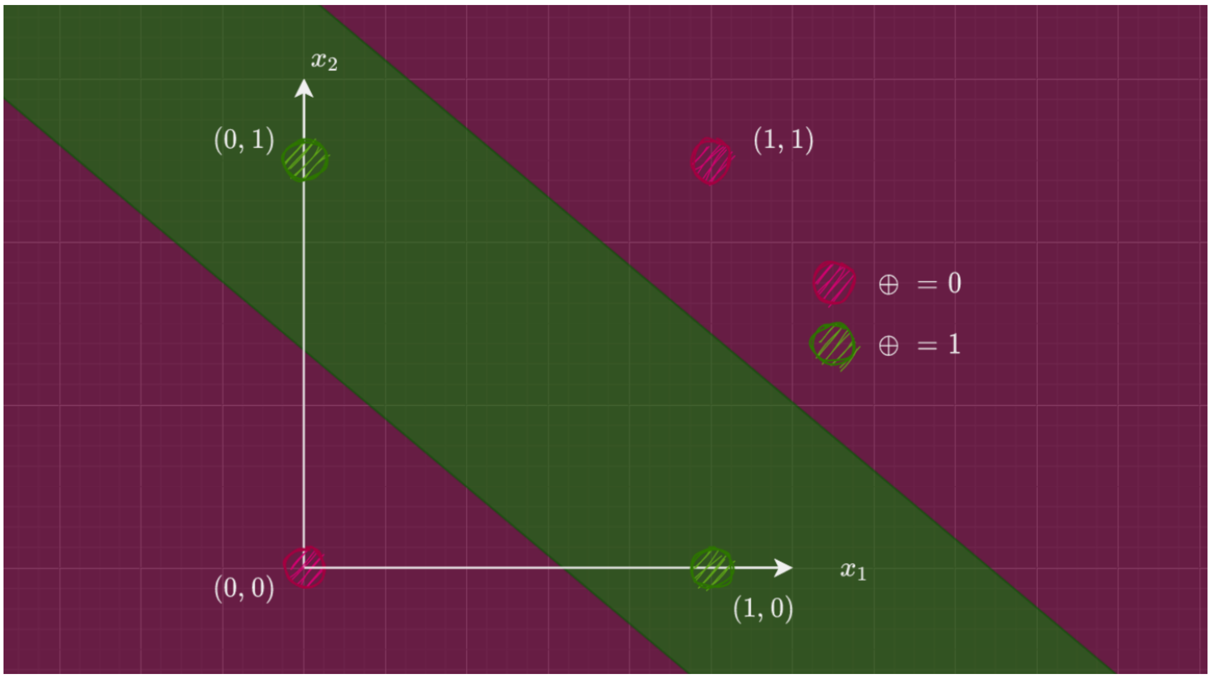
That’s our model of perceptron, and by adding our input nodes we get a model with 2 perceptron’s which looks like this:



This is how the classifier looks like for this model :

```
def XOR(x1, x2):
    """
    Return the boolean XOR of x1 and x2
    """

    x = [x1, x2]
    p_or = Perceptron(train_data, target_or)
    p_nand = Perceptron(train_data, target_nand)
    p_and = Perceptron(train_data, target_and)

    p_or.train()
    p_nand.train()
    p_and.train()

    return p_and.classify([p_or.classify(x),
                            p_nand.classify(x)])
```

If we plot the decision boundaries from our model we get something like this:

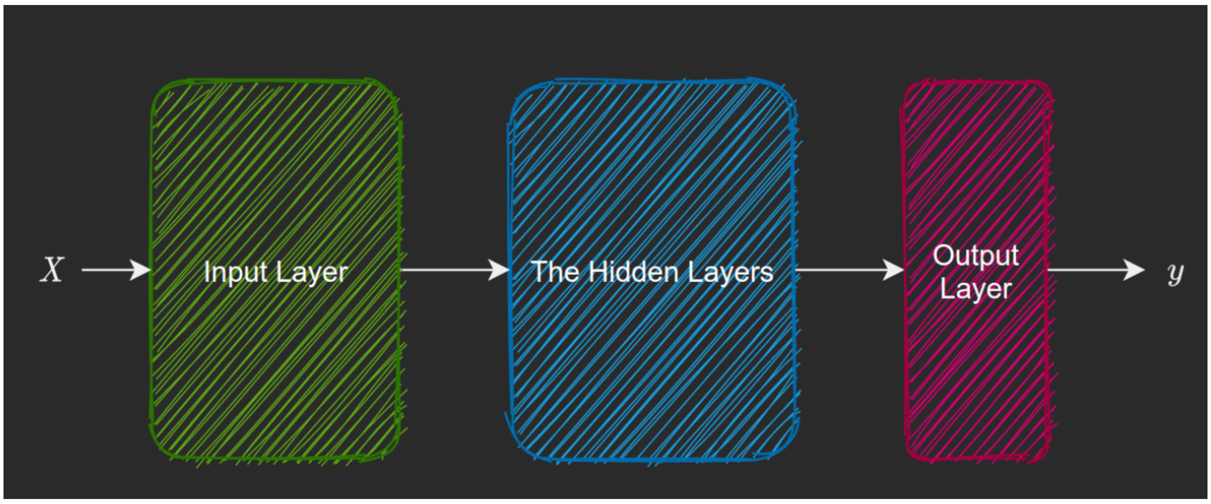


And we see that it covers the full range of possibilities, unlike linear decision boundaries.

Another way to solve the XOR gate is with a 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.

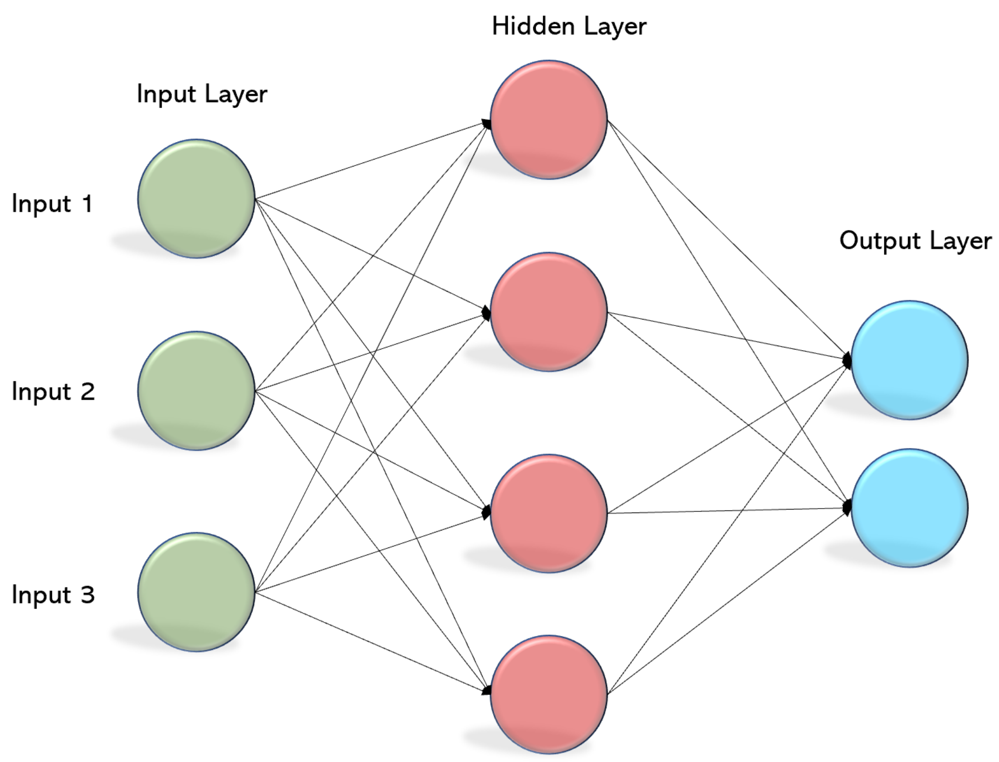
The hidden layer allows for non-linearity. A node in the hidden layer isn't too different to an output node: nodes in the previous layers connect to it with their own weights and biases, and an output is computed, generally with an activation function.



Activation functions should be differentiable so that backpropagation can be used to update a network's parameters.

Cristian Sirbu

Report



For our problem, we will be using two layers, which means we have two sets of weights and biases. Our first step will be to construct a two layer neural network, and then we will cover how to learn the weights and biases in the following section.

Note that we still have the same number of inputs (2), the same amount of outputs (1). But what is the output for the first layer, and what is the input for the second layer? This is what we will refer to as the number of hidden nodes, and you can pick any number you like.

$$h = f(W1 \times x + b1)$$
$$y = W2 \times h + b2$$

```
out = torch.sigmoid(torch.mm(x, W) + b)
```

We will use torch.sigmoid as a activation function. For the output generation process we will use Backpropagation, because is a way to update the weights and biases of a model starting from the output layer all the way to beginning. The principle behind it is that each parameter changes in proportion to how much it affects the network's output.

Backpropagation is a method of updating a model's weights and biases depending on their gradients with respect to the error function, from the output layer to the first layer.

```
# Provides extra neural network functions
import torch.nn as nn
# Provides optimizers
import torch.optim as optim
```

Cristian Sirbu

Report

```
# number of epochs
num_epochs = 10000

# Define our data for XOR problem
input_data = torch.tensor([[0., 0.], [0., 1.], [1., 0.], [1., 1.]])
output_data = torch.tensor([[0.], [1.], [1.], [0.]])

# Define weights & biases for first layer (todo)
W_1 = nn.Parameter(torch.rand(2,4))
b_1 = nn.Parameter(torch.rand(1,4))
# Define weights & biases for second layer (todo)
W_2 = nn.Parameter(torch.rand(4,1))
b_2 = nn.Parameter(torch.rand(1,1))

# Setup our loss function
loss_fn = nn.MSELoss()

# Setup our optimizer
optimizer = optim.SGD([W_1, W_2, b_1, b_2], lr=0.01)

# Define our predict function (todo)
def predict(x, W_1, W_2, b_1, b_2):
    h = torch.sigmoid(torch.mm(x,W_1) + b_1)
    output = (torch.mm(h,W_2) + b_2)
    return output

# Training loop
for epoch in range(num_epochs):
    for i in range(input_data.size(0)):
        # Get example `i` (and unsqueeze to [1, 2] and [1, 1])
        x = input_data[i].unsqueeze(0)
        y = output_data[i].unsqueeze(0)

        # Clear gradients (todo)
        optimizer.zero_grad()
        # Predict outputs (todo)
        p = predict(x, W_1, W_2, b_1, b_2)
        # Calculate loss (todo)
        loss = loss_fn(p,y)
        # Calculate gradients (todo)
        loss.backward()
        # Update weights (todo)
        optimizer.step()

    # Test our network
    if epoch % 1000 == 0:
        print(f"Testing network @ epoch {epoch}")
        for i in range(input_data.size(0)):
            # Make a prediction
            x = input_data[i].unsqueeze(0)
            y = output_data[i].unsqueeze(0)
            y_hat = predict(x, W_1, W_2, b_1, b_2)
            # Print result
            print("Input:{} Target: {} Predicted:[{}]
Error:[{}]".format(
                x.data.numpy(),
                y.data.numpy(),
                np.round(y_hat.data.numpy(), 4),
                np.round(y.data.numpy() - y_hat.data.numpy(), 4)
            ))
```

The code above is a neural network with 2 neurons and a hidden layer of other 2 neurons witch solves the XOR problem.

As more layers or nodes are added, the decision boundaries become more complex. However, this could lead to a phenomenon known as overfitting, in which a model achieves extremely high accuracies on training data but fails to generalise.

The Tensorflow Neural Net playground is a useful tool for trying out different network topologies and seeing the results.

The Mean Squared loss function was the loss function we employed in our MLP model. Though this is a popular loss function, it makes assumptions about the data (such as that it is gaussian) and isn't necessarily convex when dealing with classification problems. It was employed here to make it easier to grasp how a perceptron works, although there are superior alternatives, such as binary cross-entropy loss, for classification problems.

Cristian Sirbu

Report

Citation:

- Karajgi, A. (2021, July 21). *How Neural Networks Solve the XOR Problem | By Aniruddha Karajgi | Towards Data Science.* Medium. https://towardsdatascience.com/how-neural-networks-solve-the-xor-problem-59763136bdd7.