

From where do Neural Networks derive their power ?

Neural Networks seem to be more powerful than the models obtained from Classical Machine Learning.

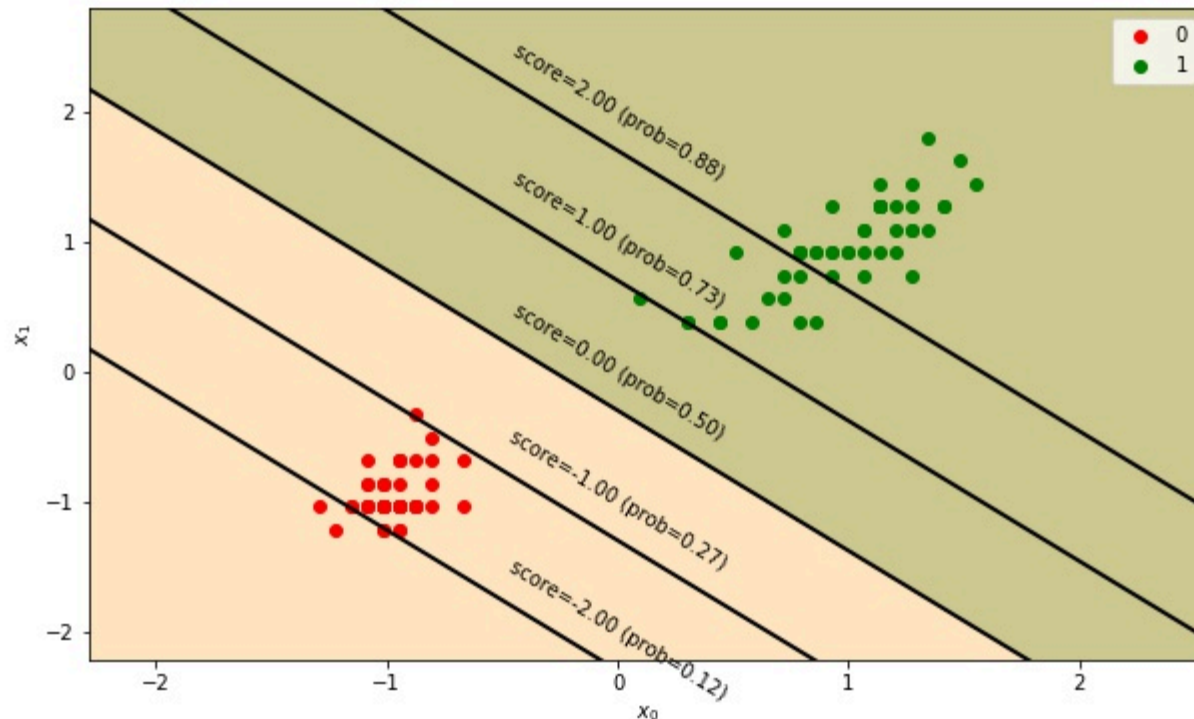
Why might that be ?

To be concrete: let us consider the Classification task.

A Classifier can be viewed as creating a decision boundary

- regions within feature space (e.g., \mathbb{R}^n) in which all examples have the same Class.

For example, a linear classifier like Logistic Regression creates linear boundaries



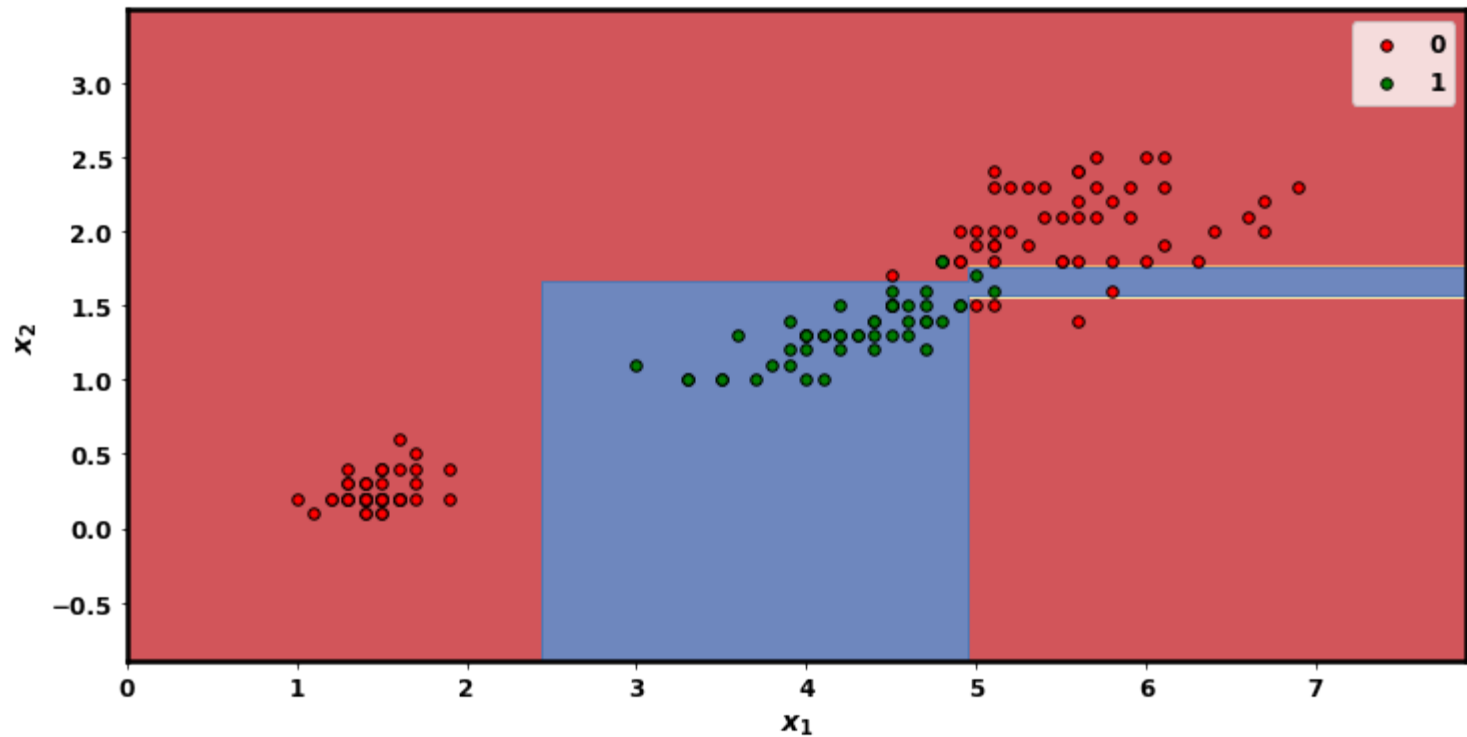
The boundaries of Decision Trees are more complex, but are perpendicular to one feature axis

- due to the nature of the question that labels a node n of the tree

$$\mathbf{x}_j^{(i)} < t_{n,j}$$

```
In [5]: X_2c, y_2c = bh.make_iris_2class()

fig, ax = plt.subplots(figsize=(12,6))
_= bh.make_boundary(X_2c, y_2c, depth=4, ax=ax)
```



As we will see:

- the shape of decision boundaries (and functions, for Regression tasks) created by Neural Networks can be much more complex
- the complexity is obtained due to the non-linear activation functions

A Neural Network computes a function

The model that solves Regression task defines a function F

- from features to prediction

$$F : \mathbb{R}^n \mapsto \mathbb{R}$$

Similarly, a model that solves a Classification task defines a function F

- from features
- to a vector of probabilities (one element for each of the $||C||$ possible class labels)

$$F : \mathbb{R}^n \mapsto \mathbb{R}^{||C||}$$

Training a model

- causes a function F to be defined
- that tries to *replicate* the training examples $\langle \mathbf{X}, \mathbf{y} \rangle$
 - the quality of the replication is defined by the Loss
- the trained model can compute F for *any* input feature vectors
 - not necessarily training

If there is some true mathematical function F' you want your model to replicate

- the more representative your training examples $\langle \mathbf{X}, \mathbf{y} \rangle$ are of true F'
- the closer the model's F will be to your desired F'

So the best a model can do is replicate the training data without loss.

We will explore the questions

- is exact replication possible ?
- what is the role of the non-linear activation functions in the replication

The power of non-linear activation functions

In our introduction to Neural Networks, we identified non-linear activation functions as a key ingredient.

Let's examine, in depth, why this is so.

Many activation functions behave like a binary "switch"

- Converting the scalar value computed by the dot product
- Into a True/False answer
- To the question: "Is a particular feature present" ?

By changing the "bias" from 0, we can move the threshold of the switch to an arbitrary value.

This allows us to construct a *piece-wise* approximation of a function

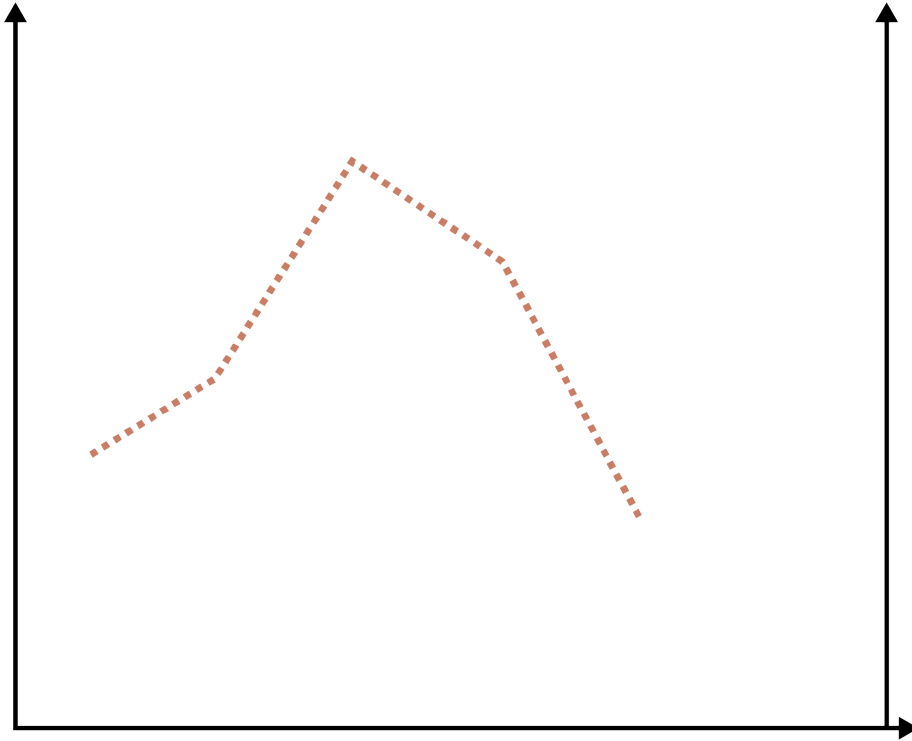
- The switch, in the region in which it is active, defines one piece
- Changing the bias/threshold allows us to relocate the piece

Consider the following function f :

Function to approximate

$f(\mathbf{x})$

\mathbf{x}



This function is

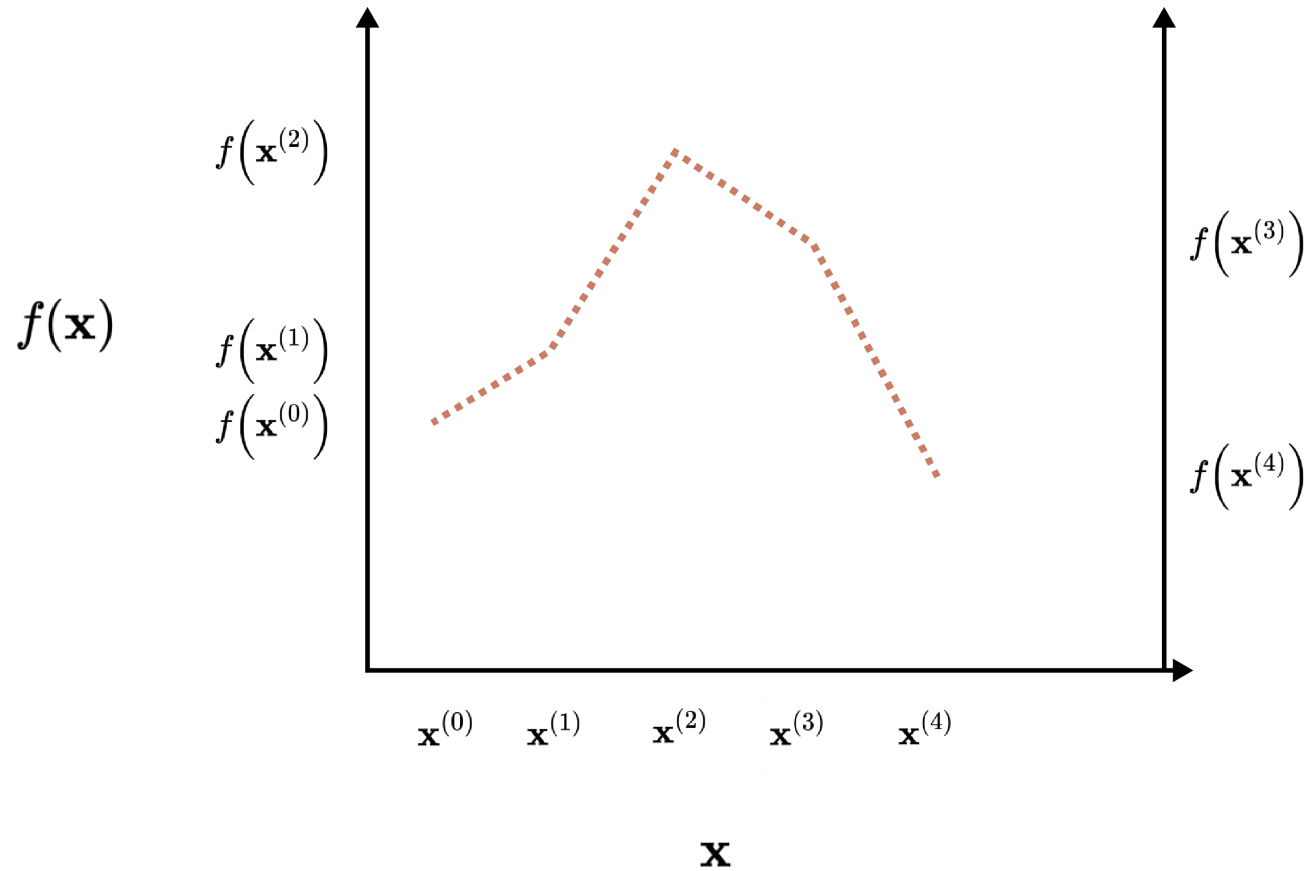
- Not continuous
- Define over set of discrete examples

$$\langle \mathbf{X}, \mathbf{y} \rangle = [\mathbf{x}^{(i)}, \mathbf{y}^{(i)} | 1 \leq i \leq m]$$

For ease of presentation, we will assume the examples are sorted in increasing value of $\mathbf{x}^{(i)}$:

$$\mathbf{x}^{(0)} < \mathbf{x}^{(1)} < \dots \mathbf{x}^{(m)}$$

Function to approximate, defined by examples \mathbf{x}

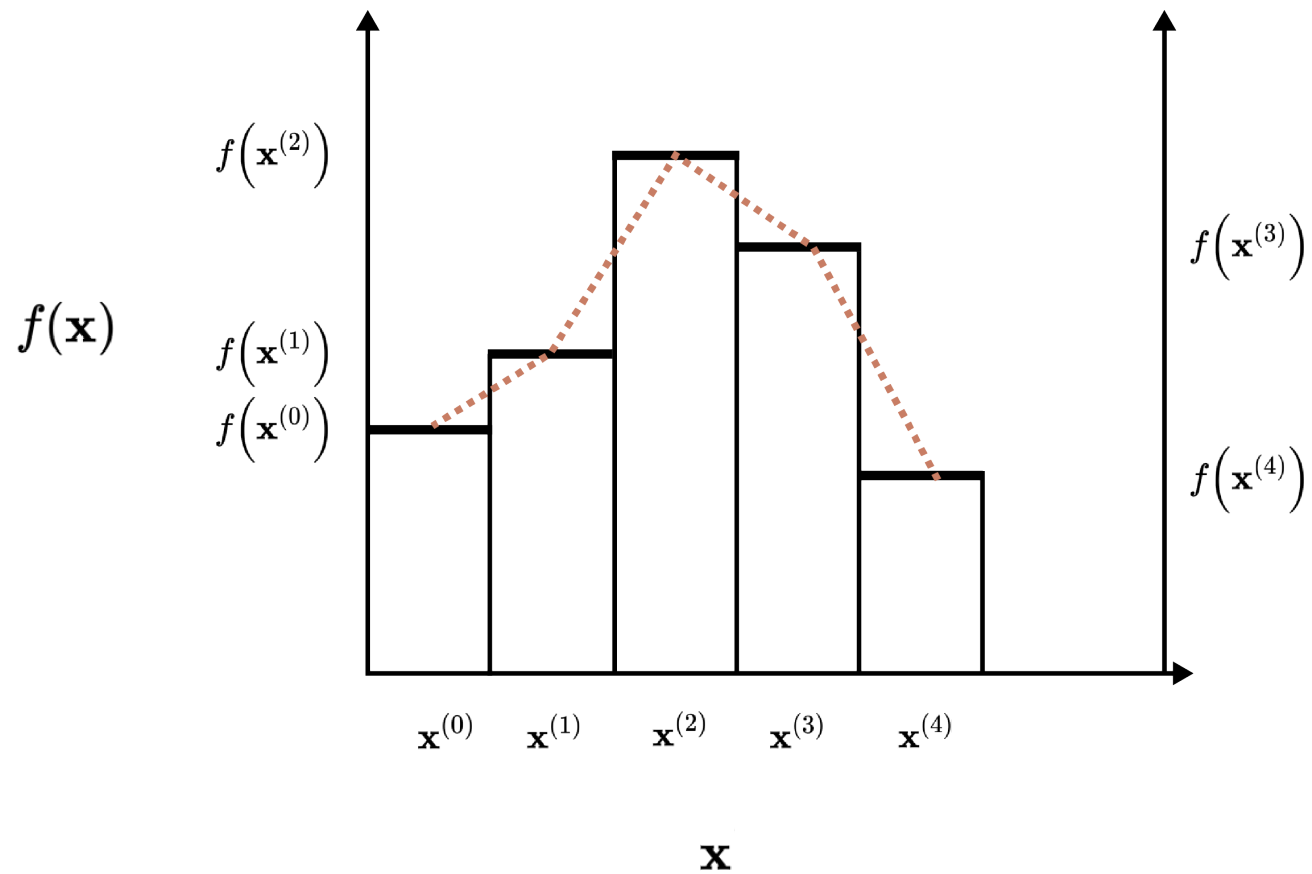


>

We can replicate the discrete function

- By a sequence of *step functions*
- Which create a piece-wise approximation of the function f

Piece-wise function approximation by step functions



We will show how to construct a Step Function using

- ReLU activation with 0 threshold

Once we have a step, we can place the center of the step anywhere along the x axis

- By adjusting the threshold of the ReLU

We start off by constructing a Binary Switch

- a function of input \mathbf{x}
- that outputs either 0 or 1
- depending on whether $x \geq \mathbf{x}^{(i)}$ for some i

Step function: binary switch with threshold $\mathbf{x}^{(i)}$



The Binary Switch can be constructed with a simple Neural network

- one network that
 - multiplies the input \mathbf{x} by a very large number
 - whose output is fed into a ReLU
 - with bias = $-\mathbf{x}^{(i)}$

This network has

- 0 output until $\mathbf{x} \geq \mathbf{x}^{(i)}$
- a nearly infinite slope

We can combine this network with a second network with similar construction

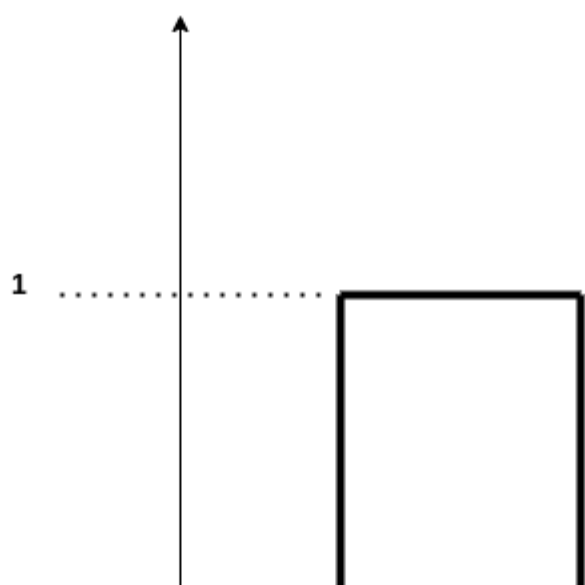
- but multiplies the input \mathbf{x} by the negative of the first large number
- and a ReLU with bias $-(\mathbf{x}^{(i)} + \epsilon)$
 - for very small ϵ

The combined network computes the Binary Switch i with threshold $\mathbf{x}^{(i)}$

Although we will not use it in our construction

- we can create a pure Impulse function
- by adding another Neural Network with
 - negative multiple
 - $\text{bias} = -\mathbf{x}^{(i+1)}$

Impulse function active between $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(i+1)}$



We will create m Binary Switches, one for each $1 \leq i \leq m$ example $\mathbf{x}^{(i)}$.

We will pair the Binary Switch with a neuron (Fully connected network with one input and one output)

- that scales the output to $f(\mathbf{x}^{(i)})$.

By careful arrangement of the Binary Switches, we will create a NN computes a function that

- exactly replicates the empirical $f(\mathbf{x})$
- has a continuous domain
 - outputs a value for all \mathbf{x} , not just $\mathbf{x} \in \mathbf{X}$

That's the idea at a very intuitive level.

The rest of the notebook demonstrates exactly how to achieve this.

Universal function approximator

A Neural Network is a Universal Function Approximator.

This means that an NN that is sufficiently

- wide (large number of neurons per layer)
- and deep (many layers; deeper means the network can be narrower)

can approximate (to arbitrary degree) the function represented by the training set.

Recall that the training data $\langle \mathbf{X}, \mathbf{y} \rangle = [(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) | 1 \leq i \leq m]$ is a sequence of input/target pairs.

The training data defines a function empirically (defined only at values $\mathbf{x} \in \mathbf{X}$).

This may look like a strange way to define a function

- but it is indeed a mapping from the domain of \mathbf{x} (i.e., \mathcal{R}^n) to the domain of \mathbf{y} (i.e., \mathcal{R})
- subject to $\mathbf{y}^i = \mathbf{y}^{i'}$ if $\mathbf{x}^i = \mathbf{x}^{i'}$ (i.e., mapping is unique).

We will demonstrate how to construct a Neural Network that exactly computes the empirically-defined function.

For simplicity of presentation

- we demonstrate this for a one-dimensional function
 - all vectors \mathbf{x} , \mathbf{y} , \mathbf{W} , \mathbf{b} are length 1.
- we assume that the training set is presented in order of increasing value of \mathbf{x} , i.e.
$$\mathbf{x}^{(0)} < \mathbf{x}^{(1)} < \dots \mathbf{x}^{(m)}$$

We will build a NN to compute this empirically defined function.

The NN will consist of m Binary Steps (one per training example)

- Binary Step i is associated with example $\langle \mathbf{x}^{(i)}, f(\mathbf{x}^{(i)}) \rangle$

Binary Step $i > 0$ computes

$$\text{Step}^{(i)}(\mathbf{x}) = \begin{cases} f(\mathbf{x}^{(i)}) - f(\mathbf{x}^{(i-1)}) & \text{if } \mathbf{x} \geq \mathbf{x}^{(i)} \\ 0 & \text{otherwise} \end{cases}$$

That is

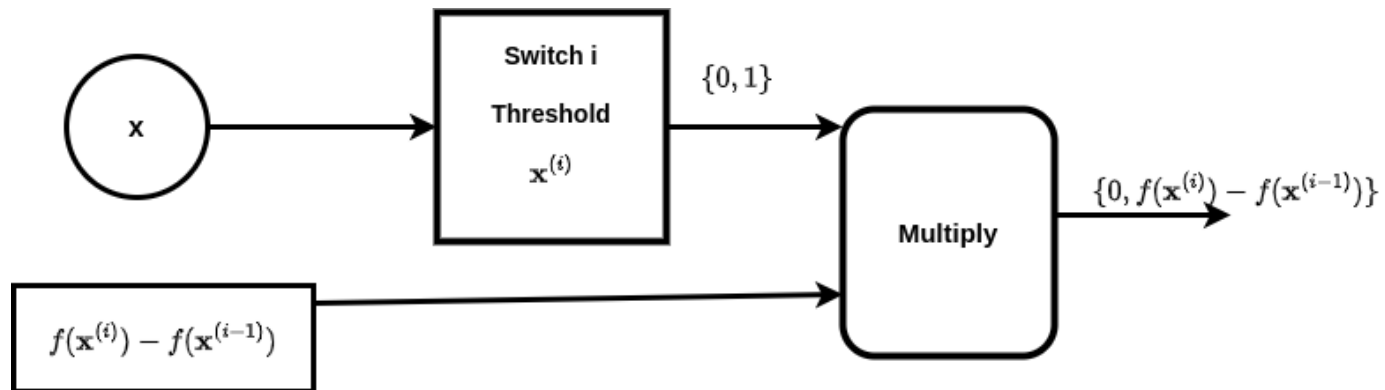
- the increment of the value of the function from input $\mathbf{x}^{(i-1)}$ to $\mathbf{x}^{(i)}$
- condition on input \mathbf{x} being at least equal to $\mathbf{x}^{(i)}$

The initial Binary Step 0 simply outputs $f(\mathbf{x}^{(0)})$

$$\text{Step}^{(0)}(\mathbf{x}) = f(\mathbf{x}^{(0)})$$

We can construct Binary Step i with

- a Binary switch that activates at $\mathbf{x}^{(i)}$
- and a multiply by a constant: $f(\mathbf{x}^{(i)}) - f(\mathbf{x}^{(i-1)})$

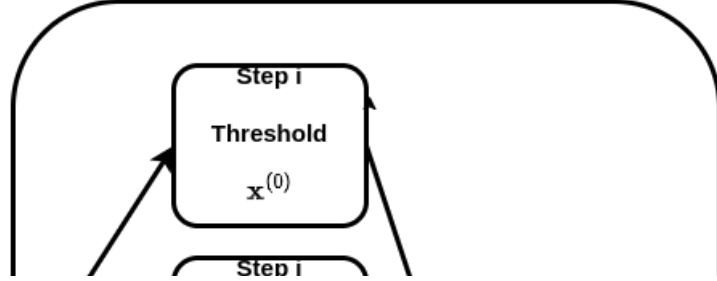


Binary Step i

- is "active"
 - output non-zero
- when input $\mathbf{x} \geq \mathbf{x}^{(i)}$

We can combine the Steps for all i

- to create a Neural Network
- that computes $f(x)$



To see that this computes an approximation of $f(x)$

- consider input \mathbf{x}
- the set I of Binary Steps i that are active are

$$I = \{i \mid \mathbf{x} \geq \mathbf{x}^{(i)}\}$$

The sum of the outputs of the active Binary steps is

$$\sum_{i \in I} f(\mathbf{x}^{(i)}) - f(\mathbf{x}^{(i-1)})$$

which equals

$$f(\mathbf{x}^{i_{\max}})$$

where

$$i_{\max} = \max_{i \in I}$$

So, for input \mathbf{x}

$$f(x) = f(\mathbf{x}^{i_{\max}})$$

Financial analogy: if we have call options with completely flexible strikes and same expiry, we can mimic an arbitrary payoff in a similar manner.

Conclusion

This proof demonstrates that **in theory** a sufficiently large Neural Network can compute any empirically-defined function.

Thus, Neural Networks are very powerful.

Observe that the key to the power is the ability to create "switches"

- which are possible only using non-linear functions (e.g., activations)

This is not to say that **in practice** this is how Neural Networks are constructed

- The network constructed is specific to a particular training set (through the definition of weights and biases)
- Not feasible to construct one network per training set
- m can be very large, and variable

In practice: we construct multi-layer ("deep") networks with fewer units and hope that Gradient Descent can "learn" weights

- to enable the network to approximate the empirical function

We don't know exactly how or why this works in practice.

We will subsequently present a module on Interpretation that offers some theories.

Alternative construction of Binary Switch with height 1

Our Binary Switch ignored the input, except to define the bias, computing

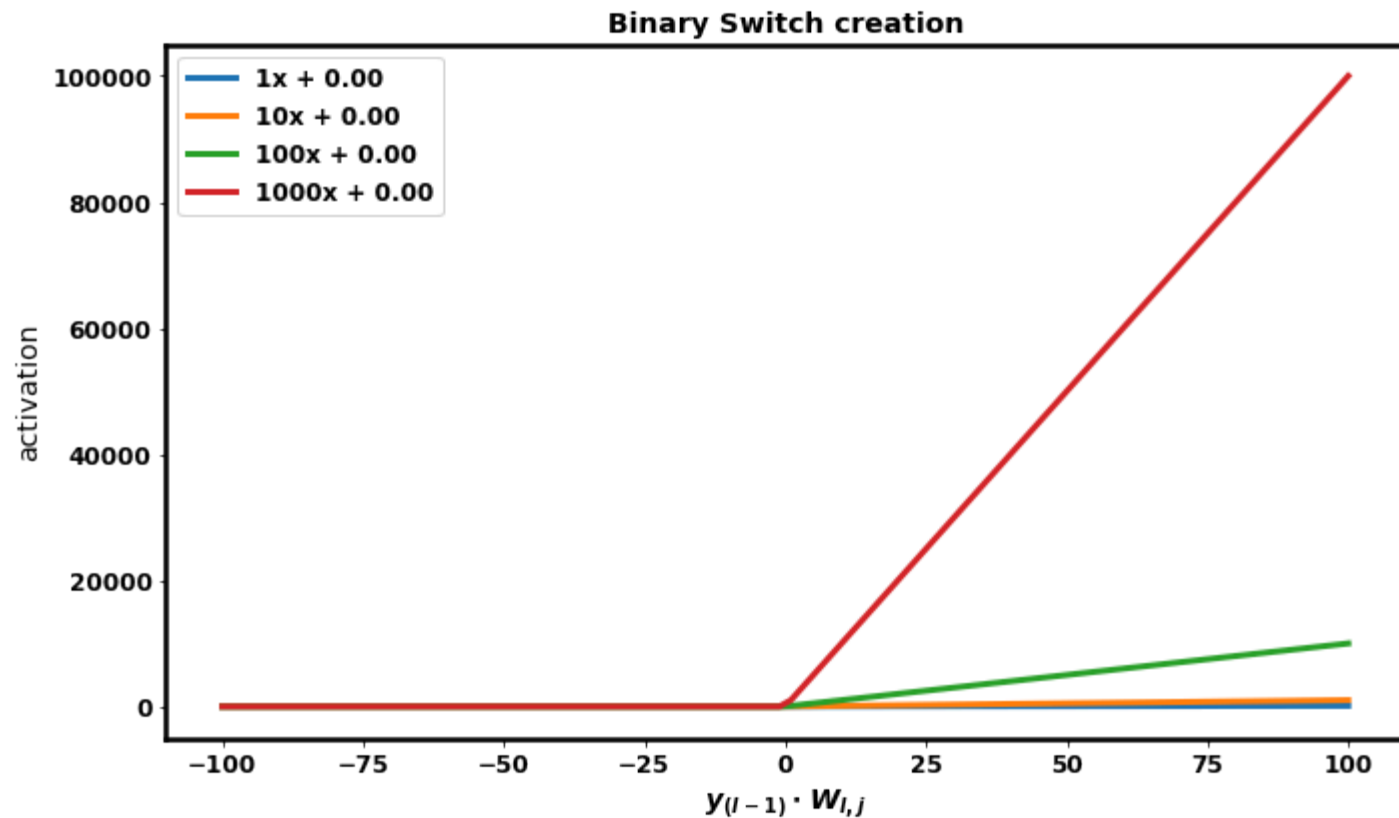
$$\max \left(0, \mathbf{W}^{(i)} * 1 + (-\mathbf{x}^{(i)}) \right)$$

We can achieve similar effect using the more standard construction where the dot product of the Neuron references \mathbf{x} , computing

$$\max \left(0, \mathbf{W}^{(i)} * \mathbf{x} + b \right)$$

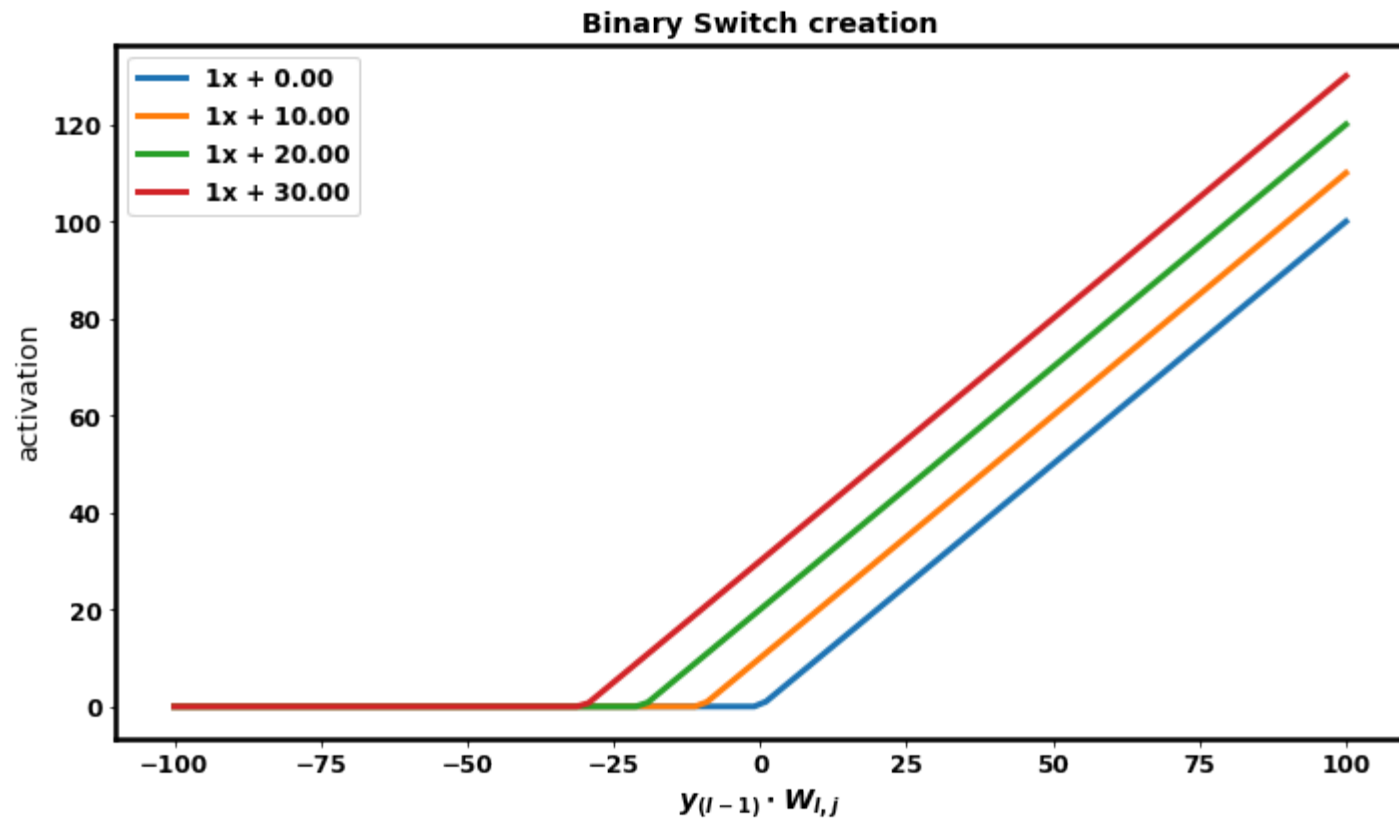
By making slope \mathbf{W} extremely large, we can approach a vertical line.


```
In [6]: _ = nnh.plot_steps( [ nnh.NN(1,0), nnh.NN(10,0), nnh.NN(100,0), nnh.NN(1000,0),  
 ])
```



And by varying the intercept (bias) we can shift this vertical line to any point on the feature axis.

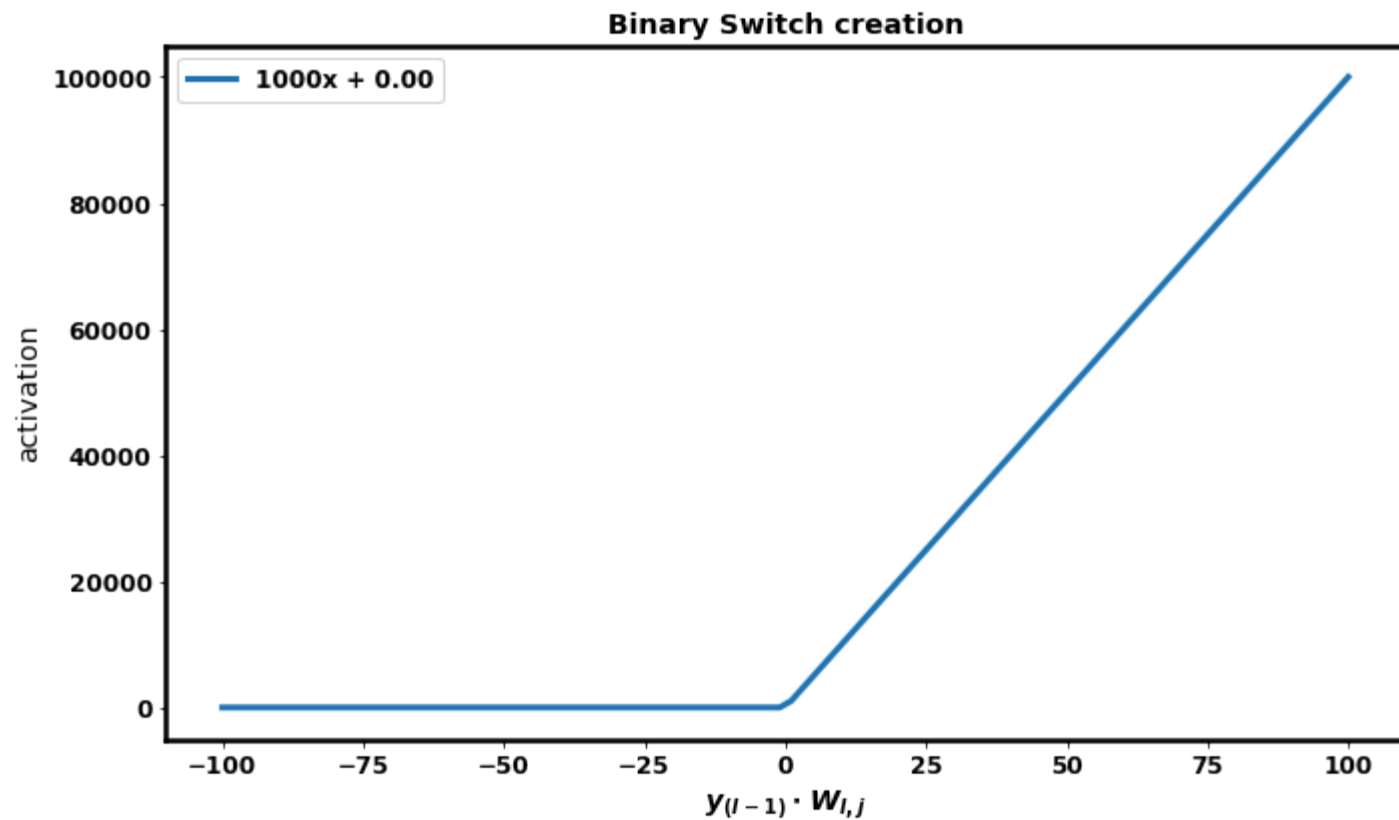
```
In [7]: _ = nnh.plot_steps( [ nnh.NN(1,0), nnh.NN(1,10), nnh.NN(1,20), nnh.NN(1,30), ])
```



With a little effort, we can construct a neuron

- With near infinite slope
- Rising from the x-axis at any offset.

```
In [8]: slope = 1000  
start_offset = 0  
  
start_step = nnh.NN(slope, -start_offset)  
  
_= nnh.plot_steps( [ start_step ] )
```



We can create a second "inverted" (negative slope) neuron with intercept "epsilon" from the first neuron

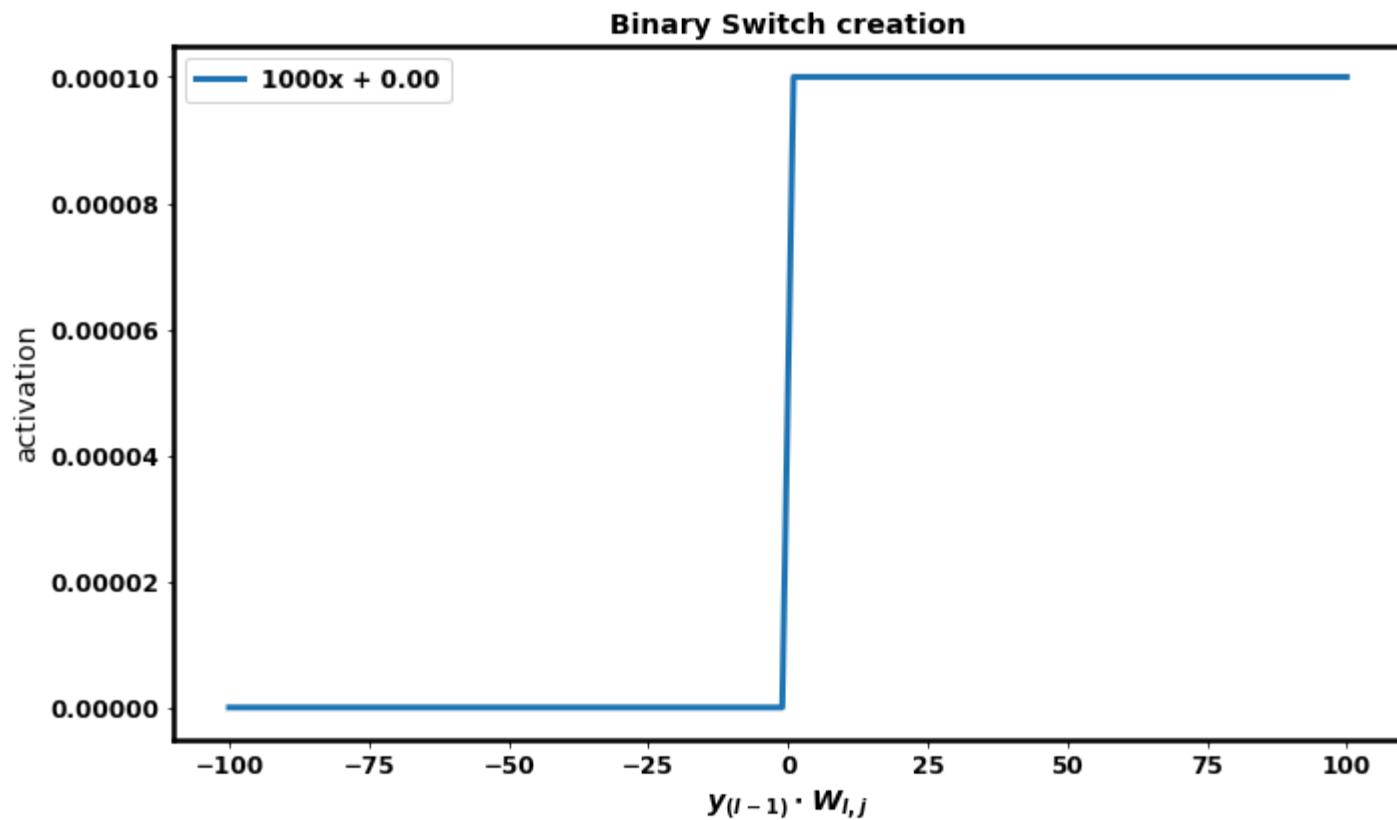
```
In [9]: end_offset = start_offset + .0001  
end_step = nnh.NN(slope,- end_offset)
```

Adding the two neurons together creates a Binary Switch

- unit height
- 0 output at inputs less than the x-intercept
- unit output for all inputs greater than the intercept).

(The sigmoid function is even more easily transformed into a step function).


```
In [10]: step= {"x": start_step["x"],
                "y": start_step["y"] - end_step["y"],
                "w": slope,
                "b": 0
            }
_ = nnh.plot_steps( [ step ] )
```



```
In [11]: print("Done")
```

Done

