From where do Neural Networks derive their power ?

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

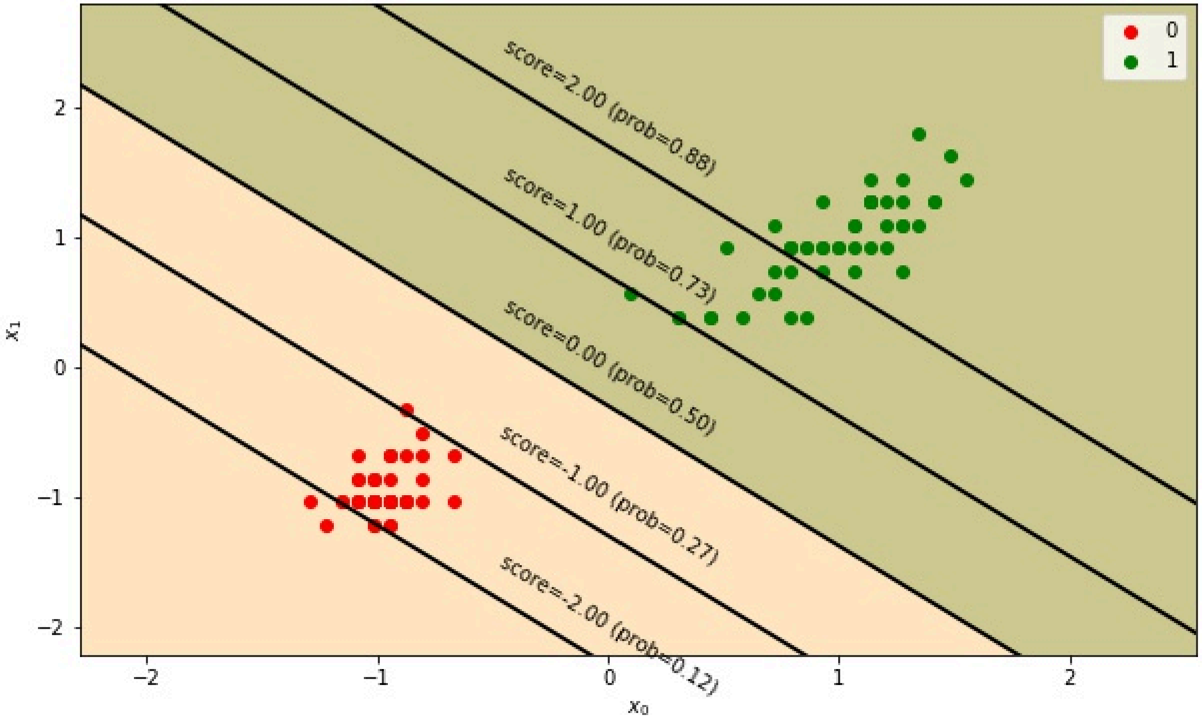
Why might that be ?

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.

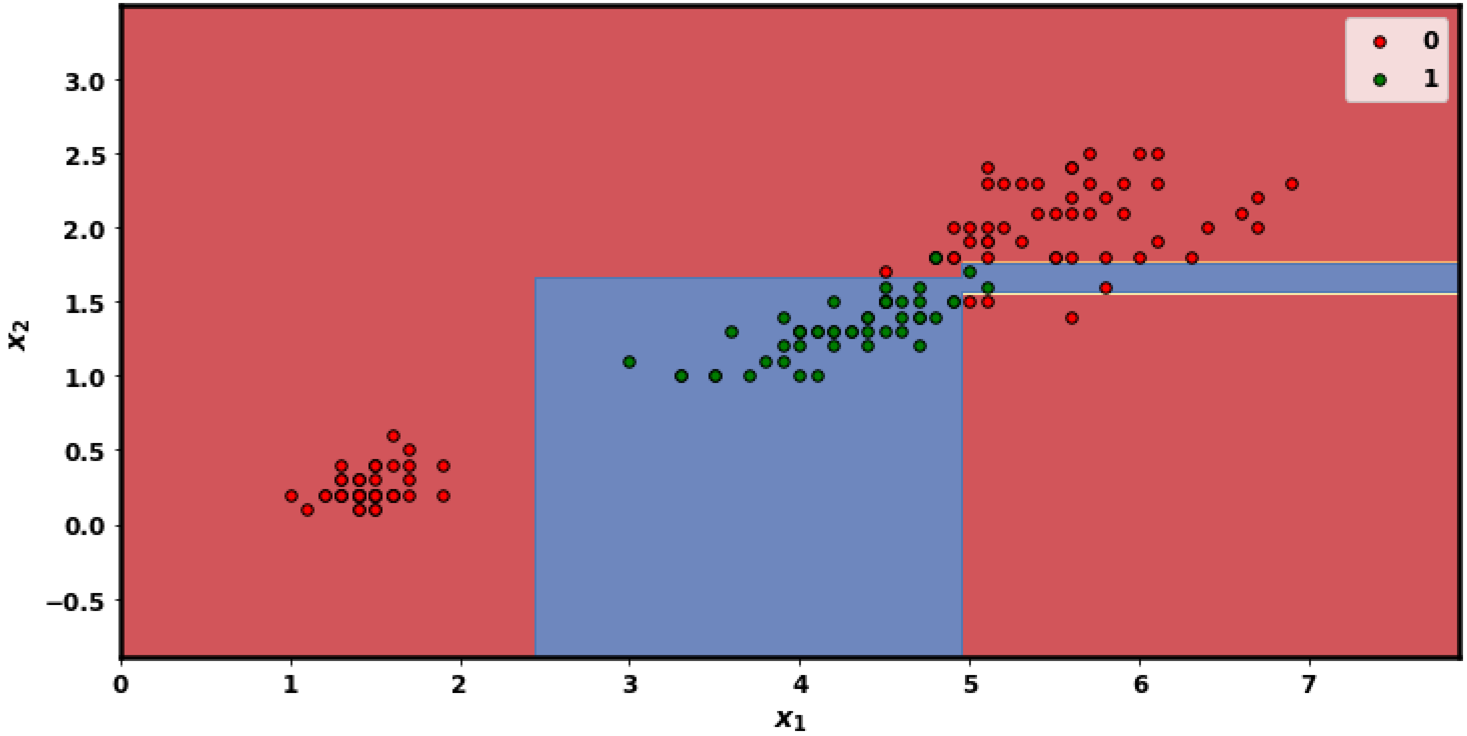
For example, a linear classifier like Logistic Regression creates linear boundaries



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

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" ?

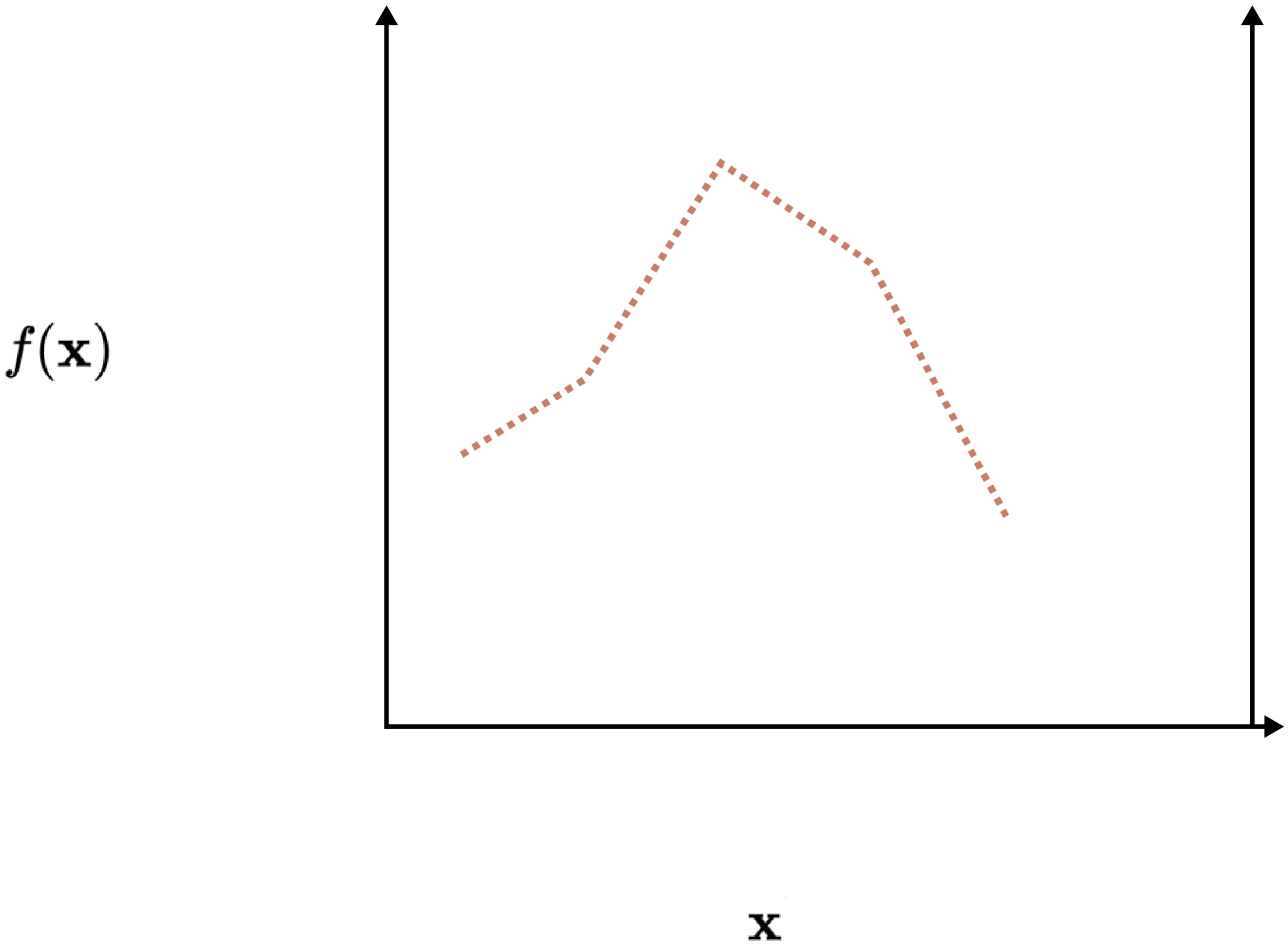
By changing the "bias" from 0, we can move the threshold of the switch to an arbitrary value.

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



This function is

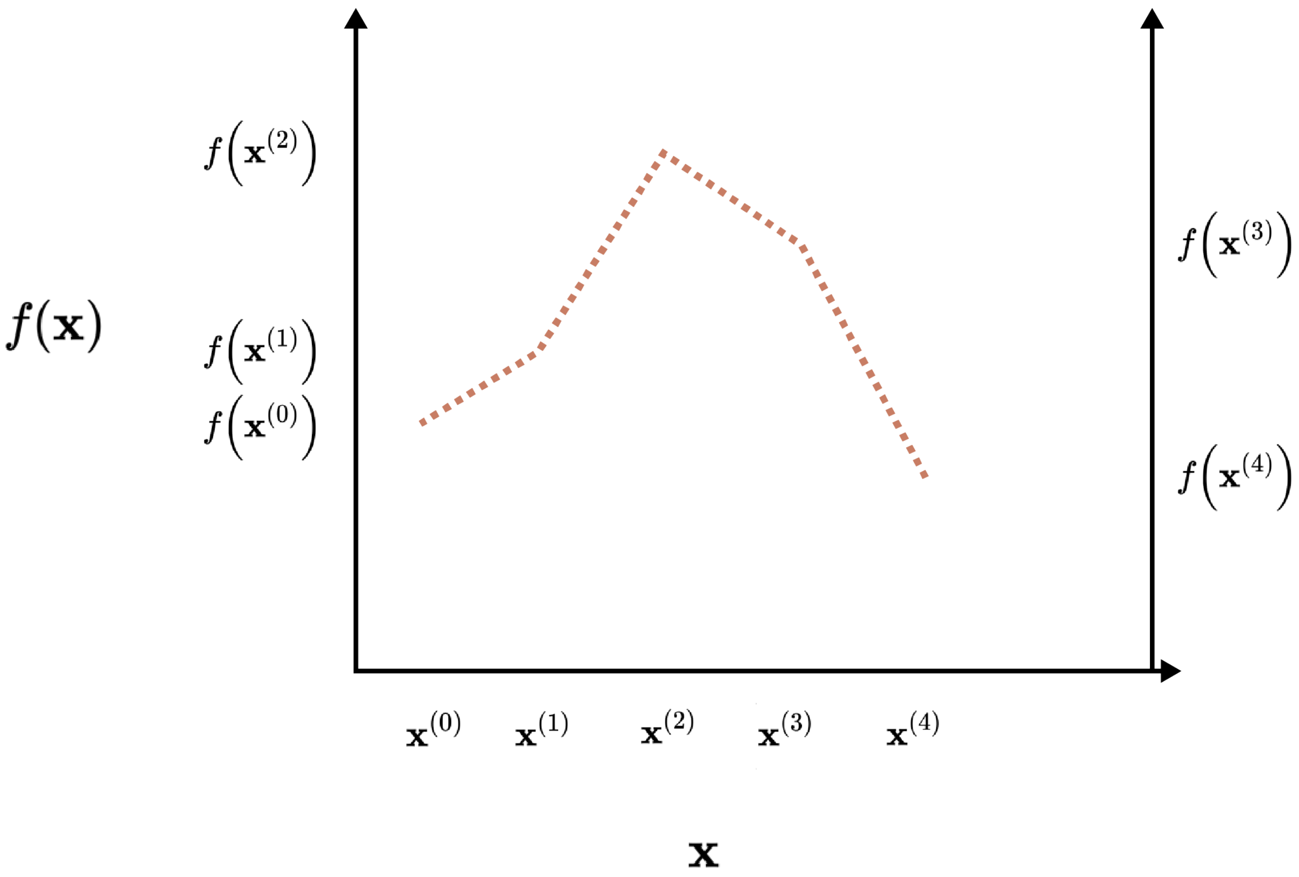
- Not continuous
- Define over set of discrete examples

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

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

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

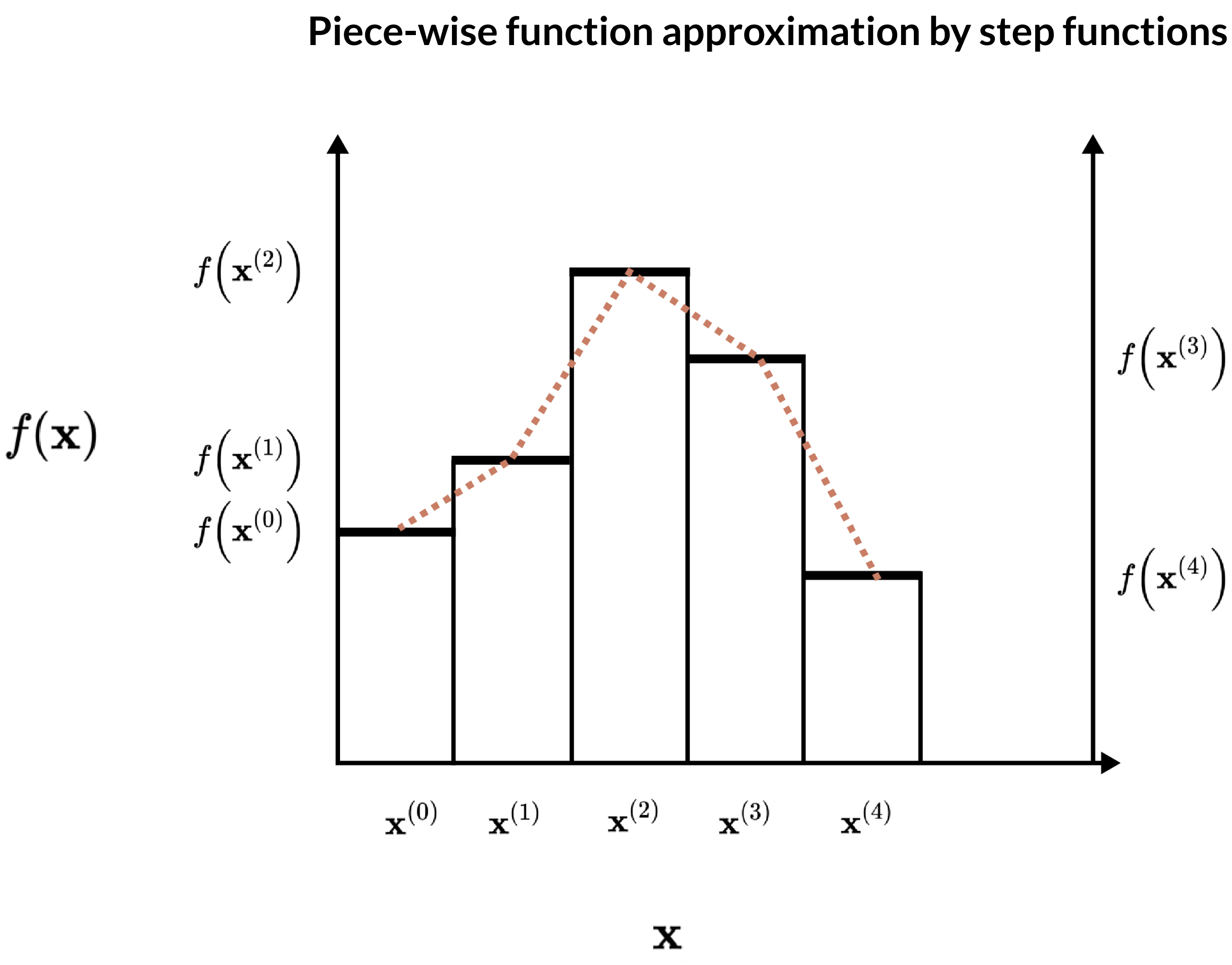
Function to approximate, defined by examples x



>

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
$f(\mathbf{x})$
$f\left(\mathbf{x}^{(2)}\right)$
$f\left(\mathbf{x}^{(1)}\right)$
$f\left(\mathbf{x}^{(0)}\right)$
$f\left(\mathbf{x}^{(3)}\right)$
$f\left(\mathbf{x}^{(4)}\right)$
$\mathbf{x}^{(0)}$
$\mathbf{x}^{(1)}$
$\mathbf{x}^{(2)}$
$\mathbf{x}^{(3)}$
$\mathbf{x}^{(4)}$
$\mathbf{x}$

We will show how to construct a Step Function using

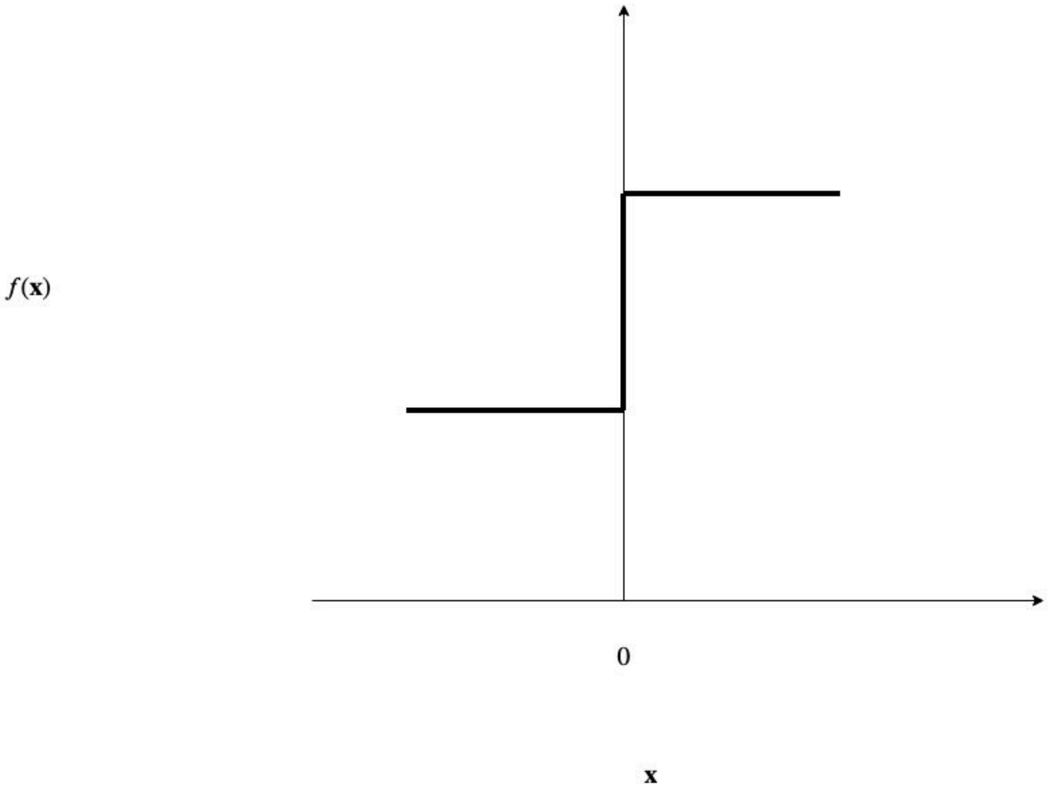
- ReLU activation with 0 threshold

Once we have a step, we can place the center of the step anywhere along the $\mathbf{x}$ axis

- By adjusting the threshold of the ReLU

We start off by constructing a binary switch (output of a ReLU with constant input equal to 1) whose output is either 0 or 1

Step function: binary switch with threshold 0



We can re-center the binary switch from activating at $\mathbf{x} = 0$ to activating at $\mathbf{x} = \mathbf{x}^{(\mathbf{i})}$

- by adjusting the bias of the ReLU to $-\mathbf{x^{(i)}}$

By adding an inverted step function (step function with negative weight) that becomes active at $\mathbf{x} = \mathbf{x}^{(i+1)}$

- we can create an impulse function that is non-zero in the range $\mathbf{x}^{(\mathbf{i})} \le \mathbf{x} \le \mathbf{x}^{(i+1)}$

Impulse function: Center x^(i); width (x^(i+1) - x^(i))


$f(\mathbf{x})$

Note that both the Binary Switch and the Impulse (step) function are created using noting more than a ReLU.

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 $\langle \mathbf{X}, \mathbf{y} \rangle = [(\mathbf{x}^{(\mathbf{i})}, \mathbf{y}^{(\mathbf{i})}) | 1 \le i \le m]$ 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 $\mathbf{y}^i = \mathbf{y}^{i'} \text{ if } \mathbf{x}^i = \mathbf{x}^{i'}$ (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.
$$\mathbf{x}^{(0)} < \mathbf{x}^{(1)} < \ldots \mathbf{x}^{(m)}$$

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

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

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

Here is the Binary Switch i that we will associate with example i having $\mathbf{x} = \mathbf{x^{(i)}}$

- A Fully Connected network with one unit ("neuron")
- Constant input equal to the value 1
- Bias equal to $-\mathbf{x^{(i)}}$
- Weight $\mathbf{W^{(i)}}$ = $(f(\mathbf{x^{(i)}}) - f(\mathbf{x}^{i-1}))$
    - The amount by which $f(\mathbf{x})$ increases between steps is

Binary Switch i becomes "active" (non-zero output) for $\mathbf{x} \geq \mathbf{x^{(i)}}$

Binary Switch i

- computes
$$\max\left(0, \mathbf{W^{(i)}} * 1 + (-\mathbf{x^{(i)}})\right)$$
- is "active" (non-zero output) only if $\mathbf{x} \geq \mathbf{x^{(i)}}$

Let us construct m Binary Switches, one per training example

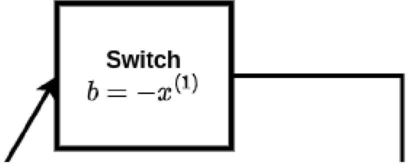
- one per example
- bias for Binary Switch i is $-\mathbf{x^{(i)}}$
- weights are

$$
\begin{aligned}
\mathbf{W}^{(1)} &= \mathbf{y}^{(1)} \\
\mathbf{W^{(i)}} &= \mathbf{y}^{(i)} - \mathbf{y}^{(i-1)}
\end{aligned}
$$

We connect all m Binary Switches to a "final" neuron that simply adds the outputs of all m Binary Switches

- m inputs
- all weights equal to 1
- Bias equal to 0

Function Approximation by Binary Switches


Switch
$b = -x^{(1)}$

Consider what happens when we input $\mathbf{x} = \mathbf{x^{(i)}}$ to this network.

- The only active Binary Switches are those with index at most i
- The Final Neuron computes

$$\begin{aligned} \sum_{i'=1}^{i} \mathbf{W}^{(i')} &= \mathbf{y}^{(1)} + \sum_{i'=2}^{i} \mathbf{y}^{(i')} - \mathbf{y}^{(i'-1)} & \text{definition of } \mathbf{W}^{(i')} \\ &= \mathbf{y^{(i)}} \end{aligned}$$

Thus, our two layer network outputs $\mathbf{y^{(i)}}$ given input $\mathbf{x^{(i)}}$.

It also computes a value for **any** $\mathbf{x}$, not just $\mathbf{x} \in \mathbf{X}$.

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.

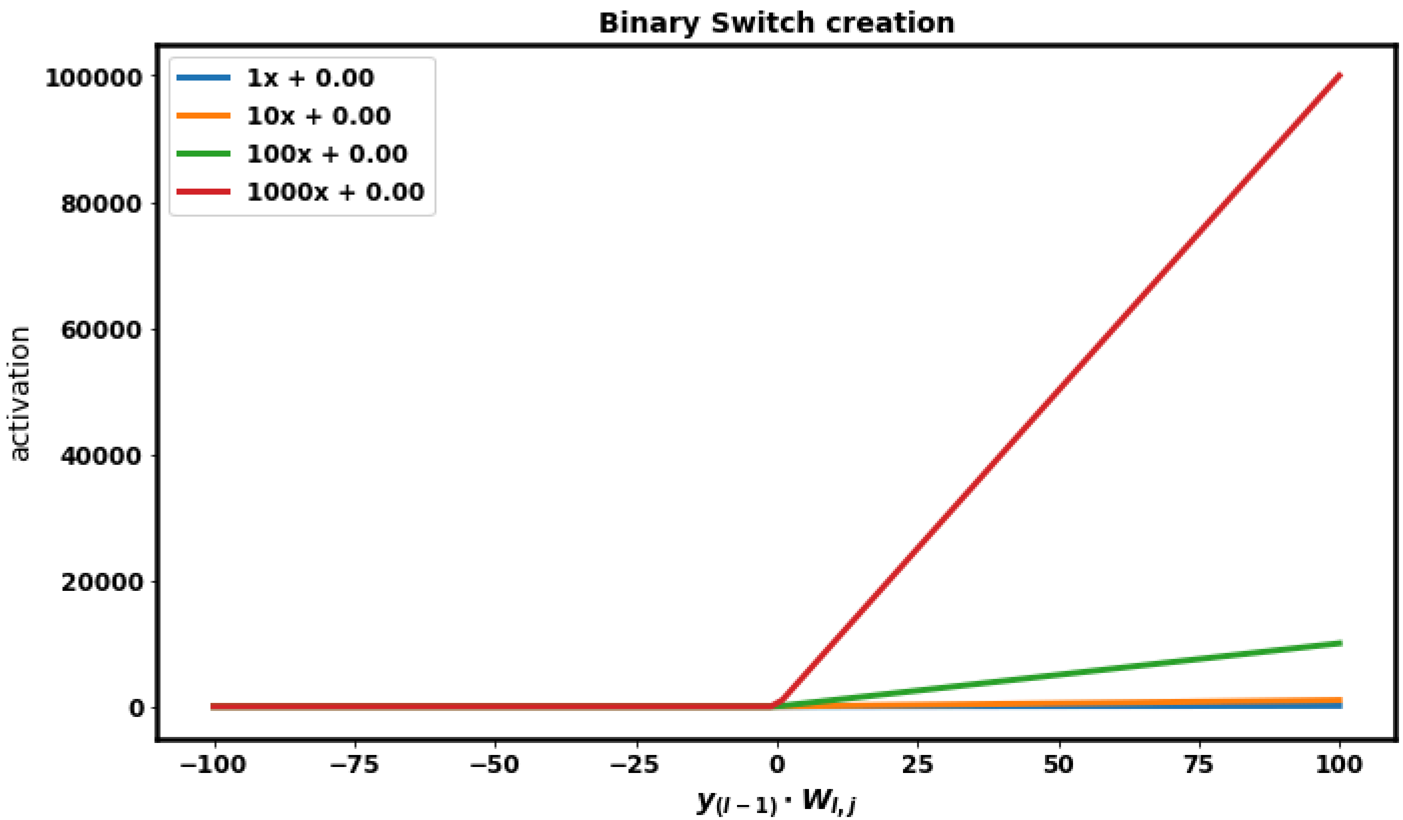
Alternative construction of Binary Switch with height 1

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

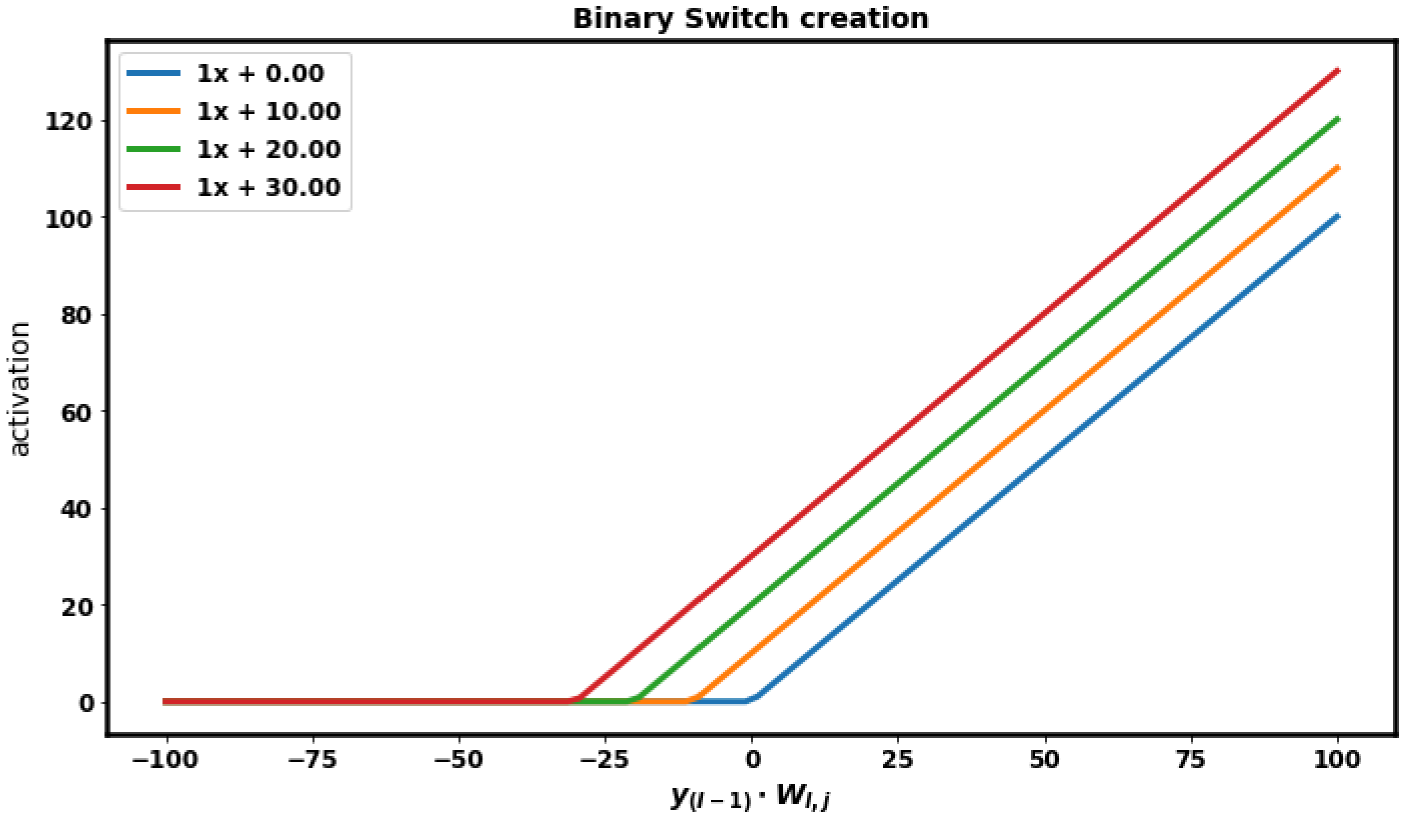
By making slope $\mathbf{W}$ extremely large, we can approach a vertical line.

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), ])
```

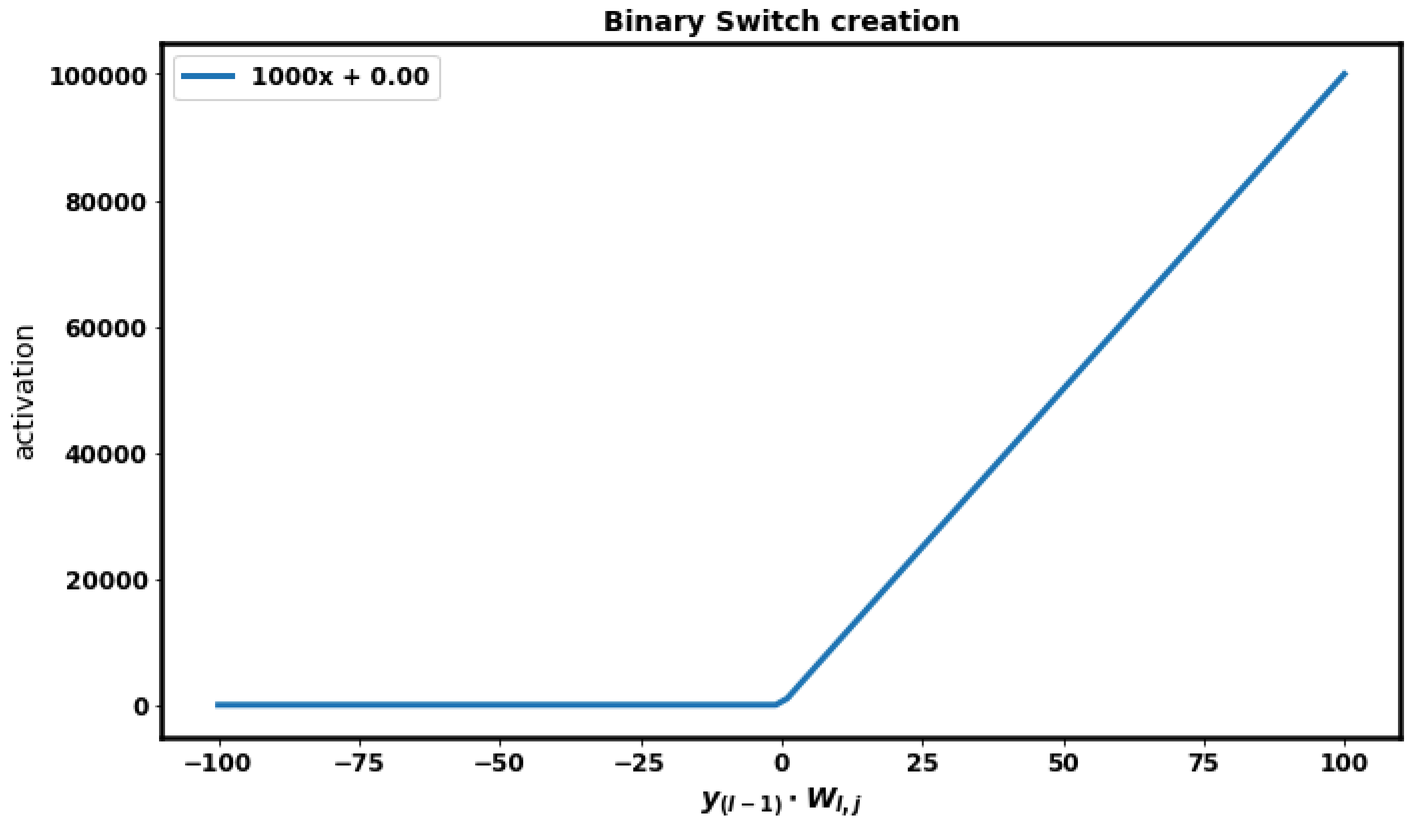


With a little effort, we can construct a neuron

- 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

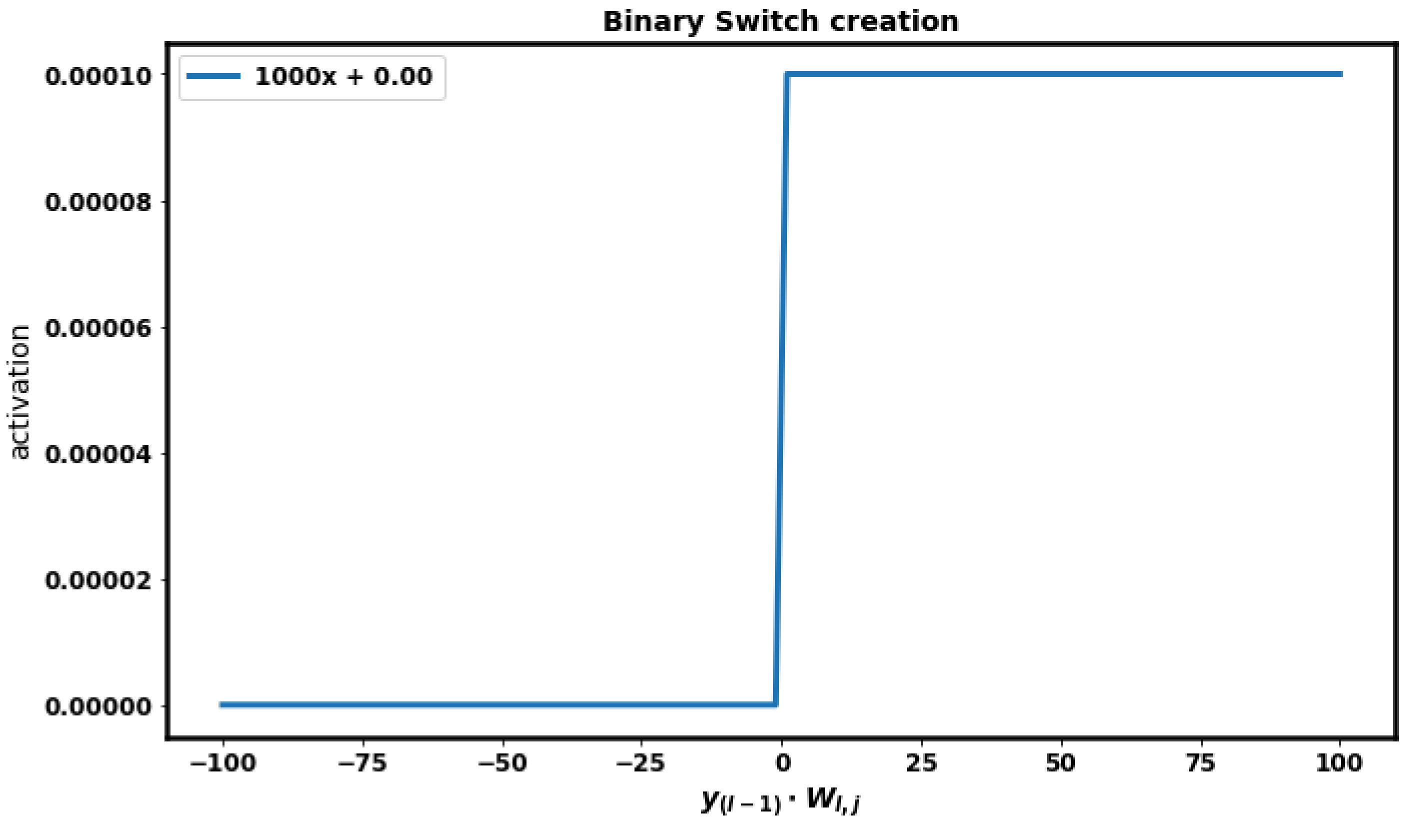
In [9]:
```
end_offset = start_offset + .0001

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
```