

# 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

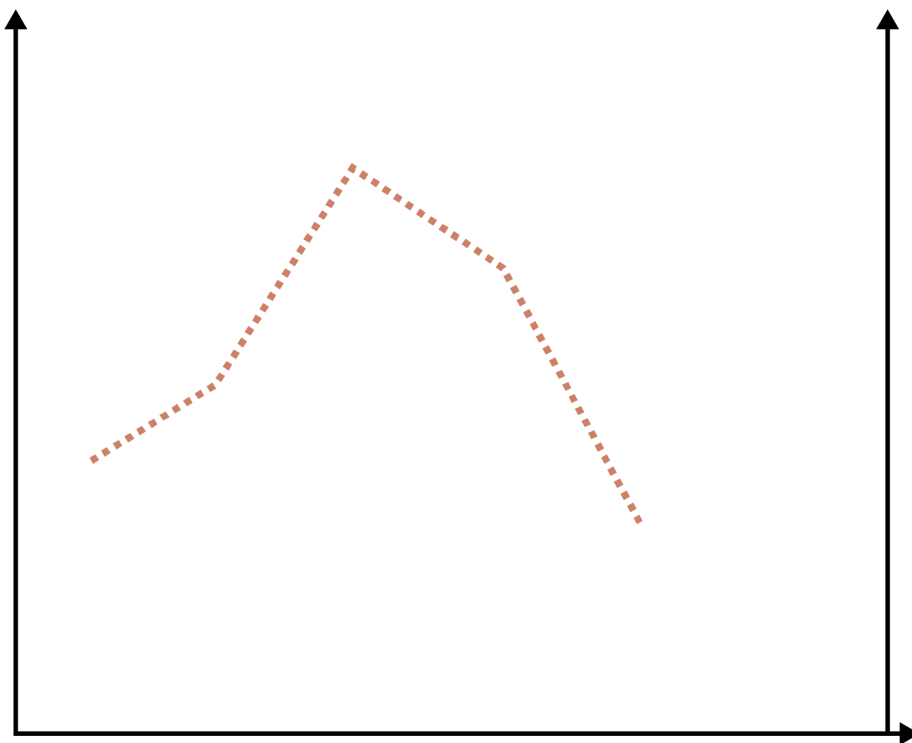
- The switch, in the region in which it is active, defines one piece
- Changing the bias/threshold allows us to relocate the piece

Consider the following function  $f$ :

Function to approximate

$f(\mathbf{x})$

$\mathbf{x}$

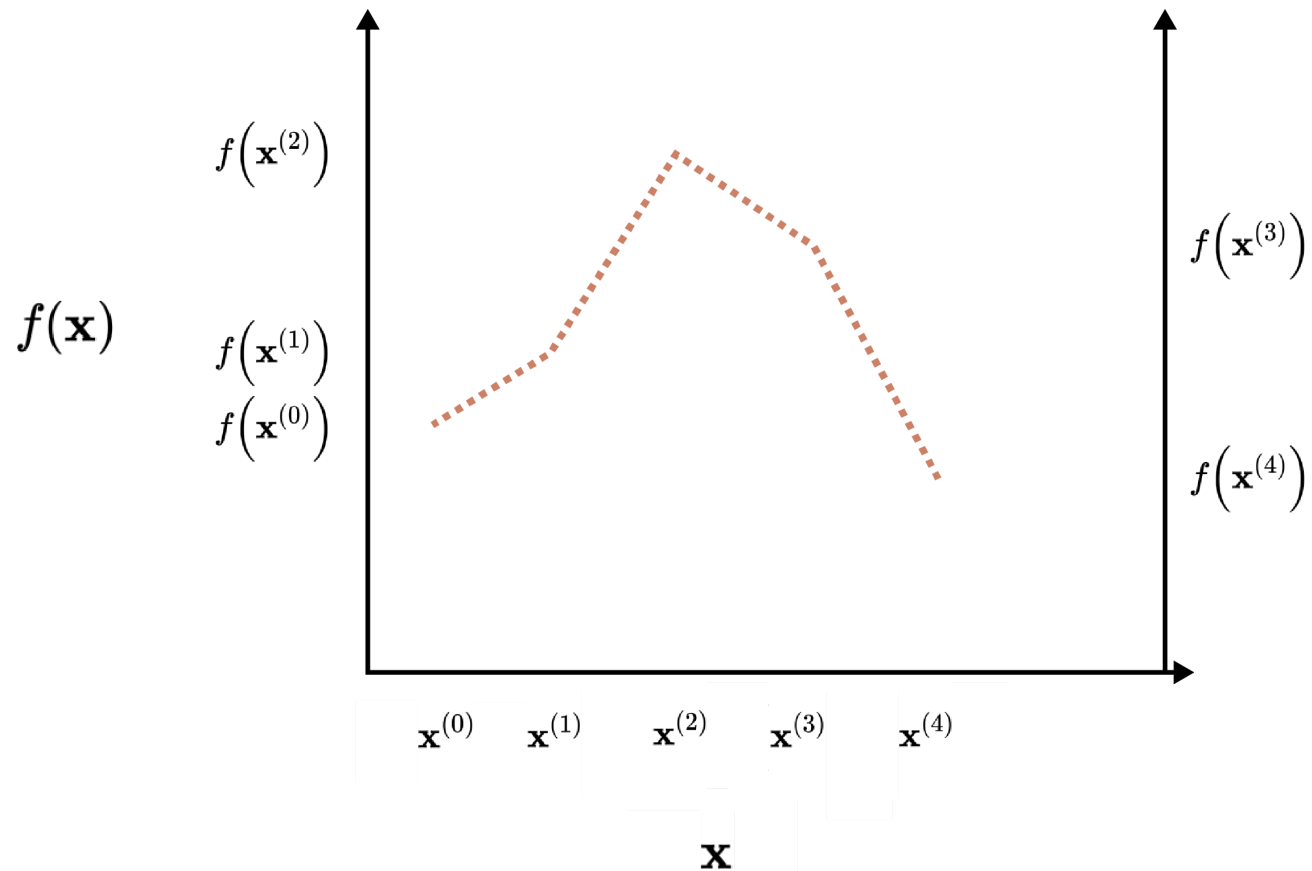


This function is

- Not continuous
- Define over set of discrete examples

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

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



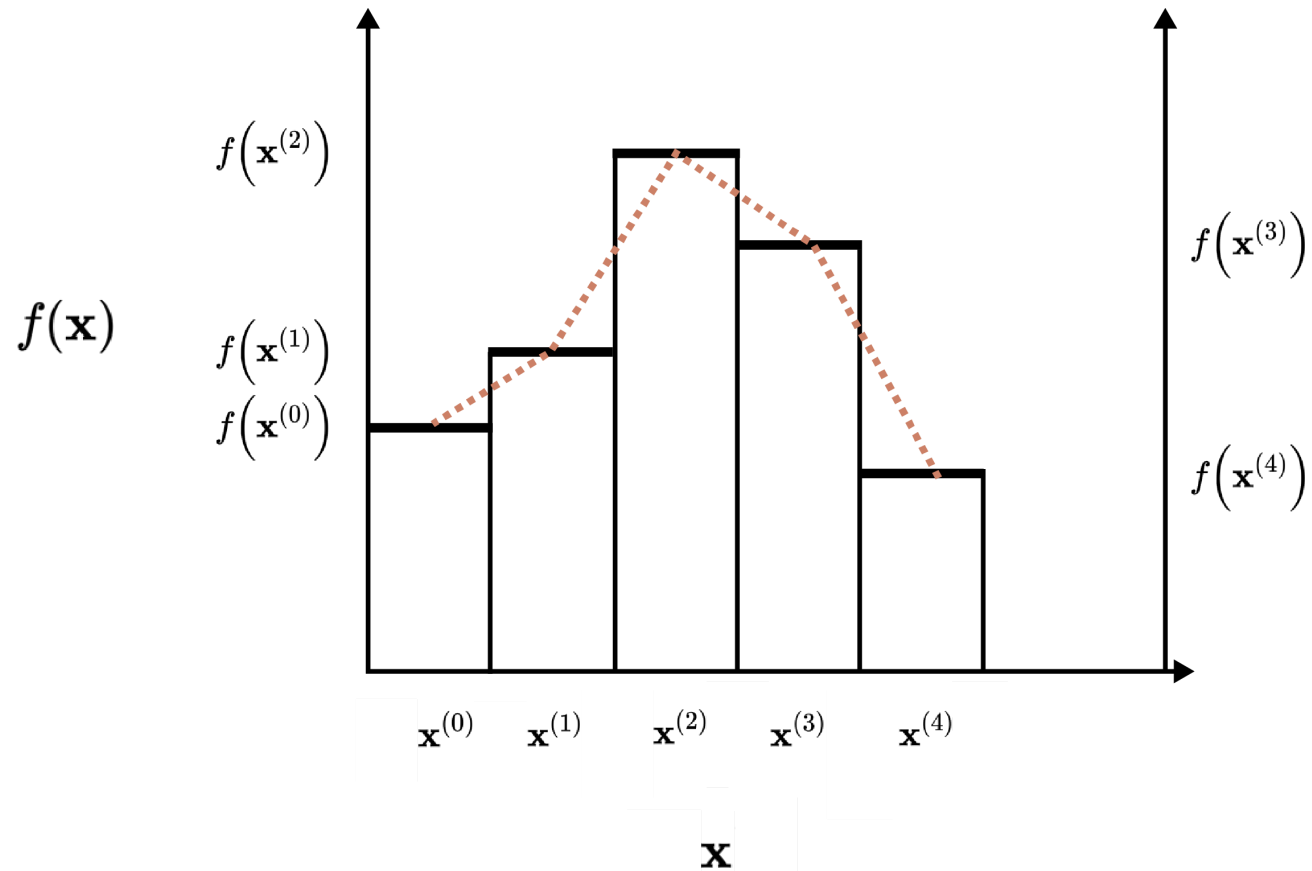
>

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

- Dot product
- ReLU activation with 0 threshold

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

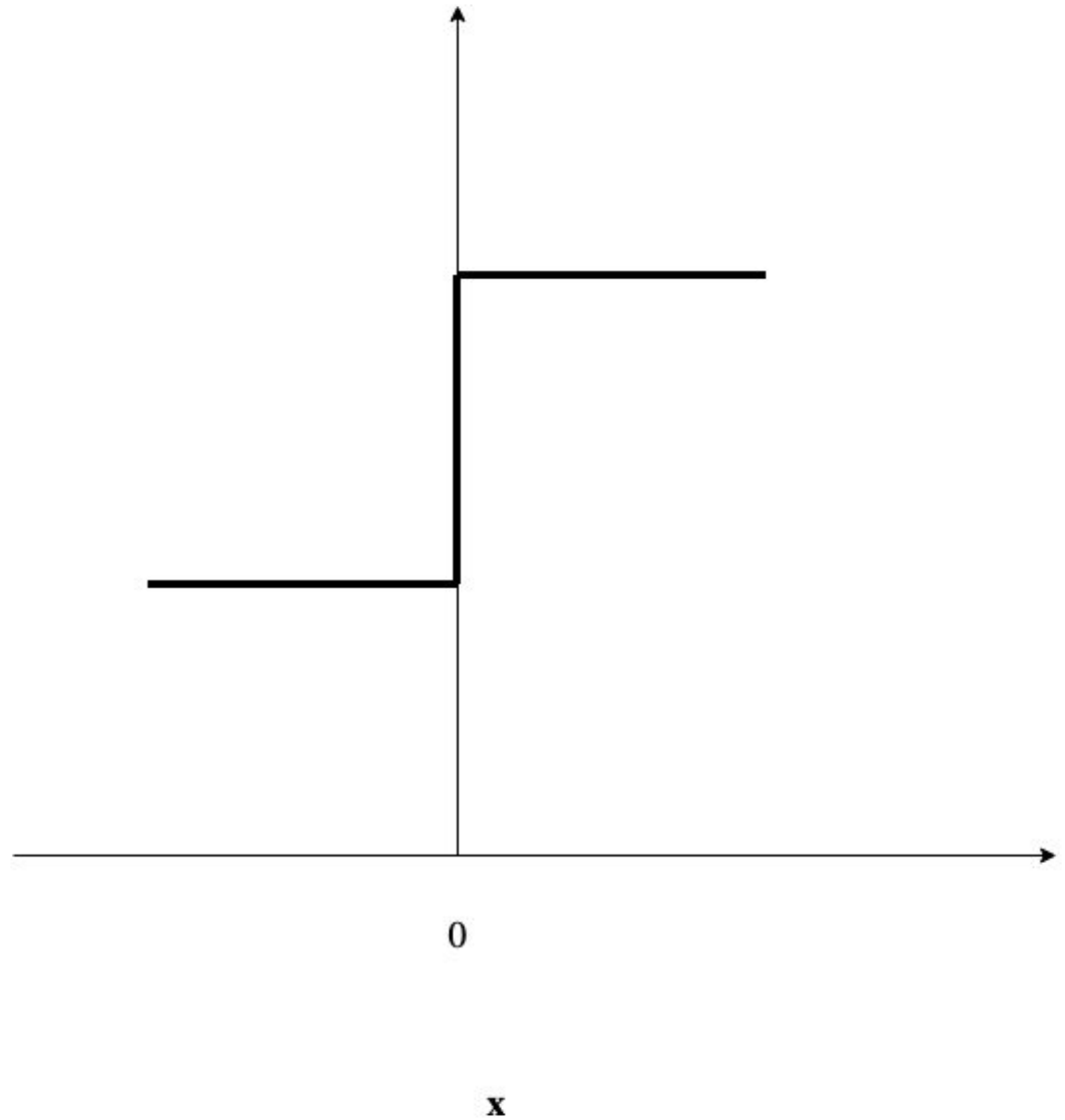
- By adjusting the threshold of the ReLU

The plan is:

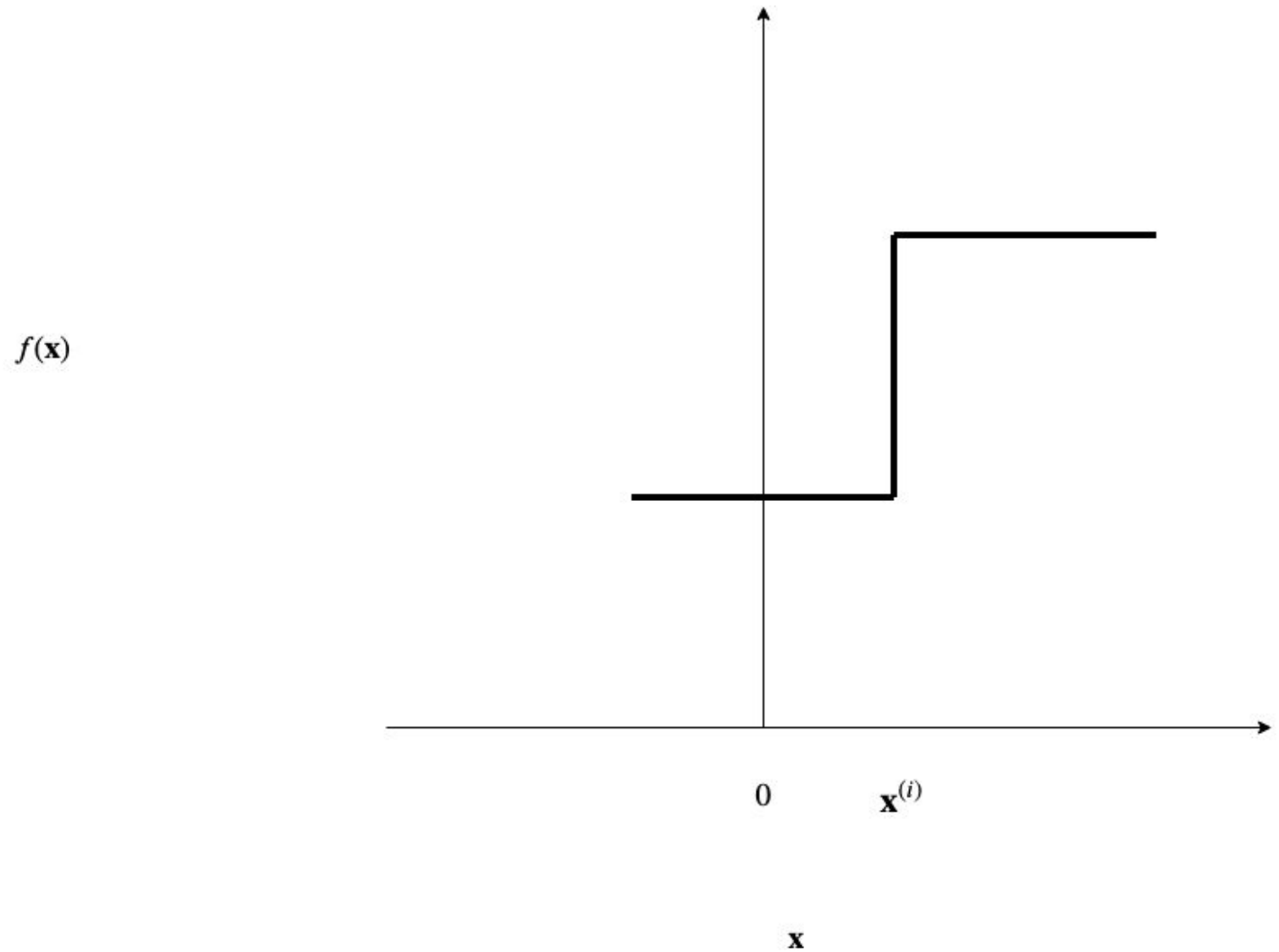
- Construct a step function for the  $i^{th}$  example
- Step  $i$  becomes "active" when its input is at least  $x^{(i)}$ , using the bias of the ReLU
- Height of  $i^{th}$  step is  $f(\mathbf{x}^{(i)})$
- The amount by which  $f(\mathbf{x})$  increases between steps is  $(f(\mathbf{x}^{(i+1)}) - f(\mathbf{x}^{(i)}))$

# Step function: binary switch with threshold 0

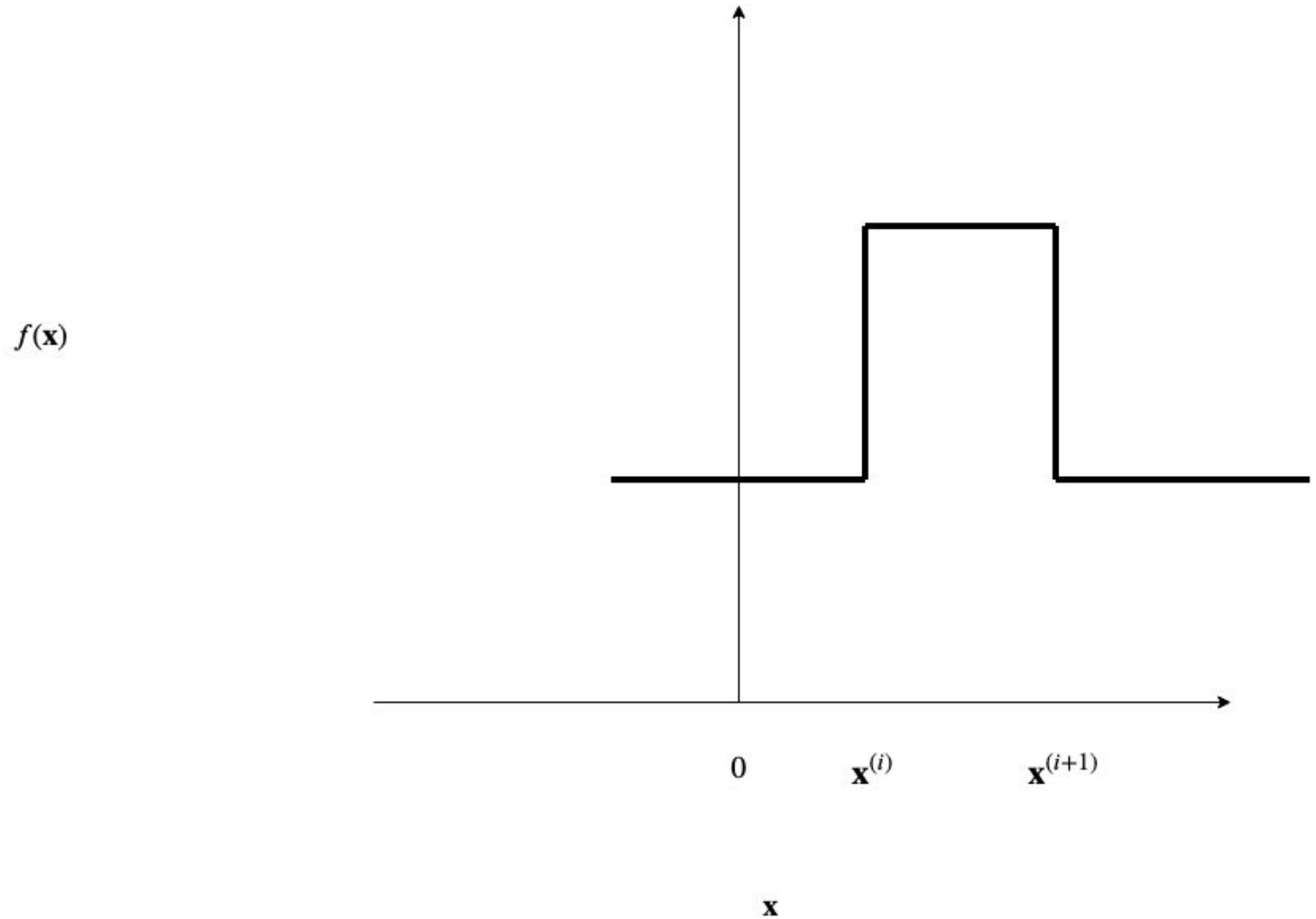
$f(\mathbf{x})$



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



Impulse function: Center  $\hat{x}^{(i)}$ ; width  $(\hat{x}^{(i+1)} - \hat{x}^{(i)})$



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.



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 give an intuitive proof for a one-dimensional function

- all vectors  $\mathbf{x}$ ,  $\mathbf{y}$ ,  $\mathbf{W}$ ,  $\mathbf{b}$  are length 1.

For simplicity, let's assume that the training set is presented in order of increasing value of  $\mathbf{x}$ , i.e.

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

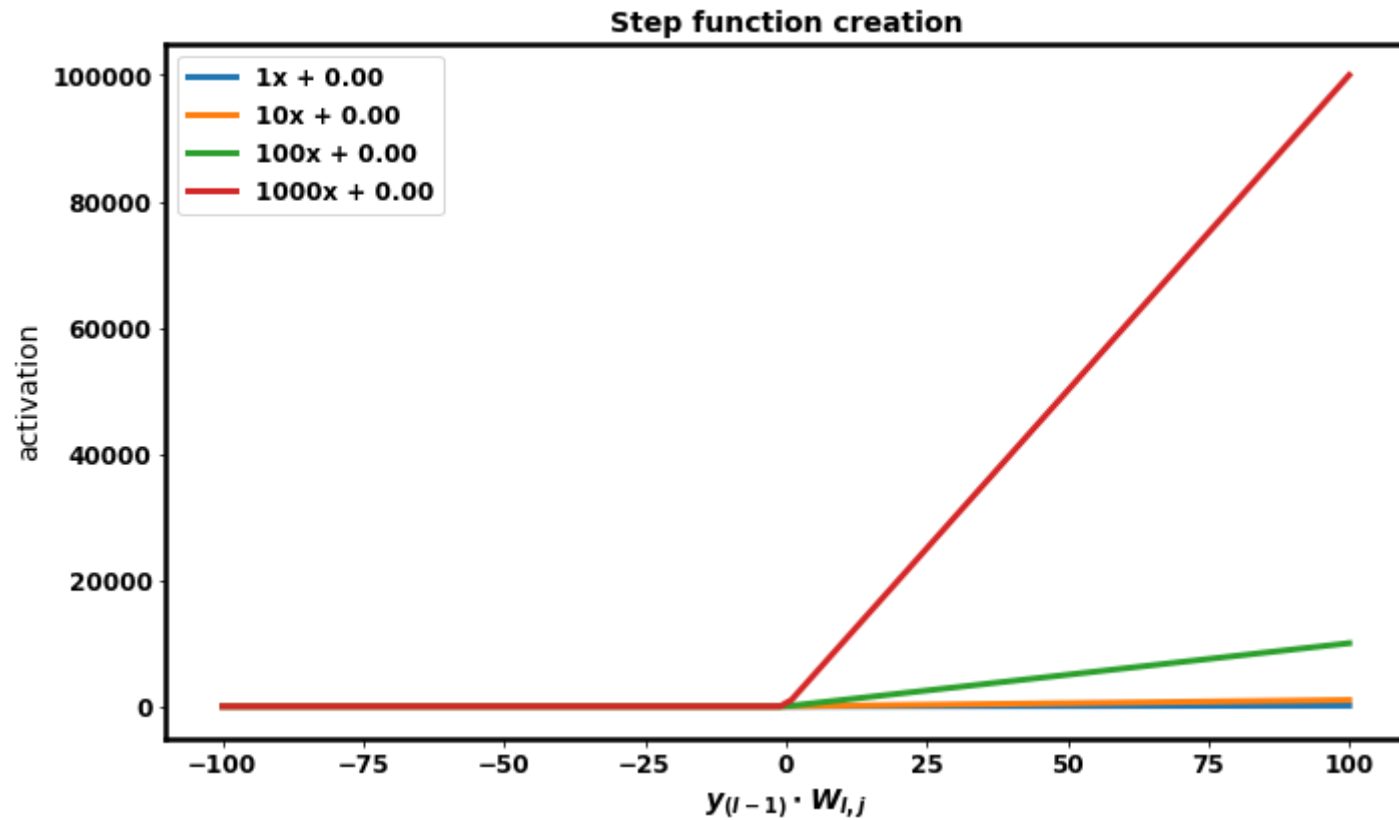
Consider a single neuron with a ReLU activation, computing  
 $\max(0, \mathbf{W}\mathbf{x} + \mathbf{b})$

Let's plot the output of this neuron, for varying  $\mathbf{W}$ ,  $\mathbf{b}$ .

The slope of the neuron's activation is  $\mathbf{W}$  and the intercept is  $\mathbf{b}$ .

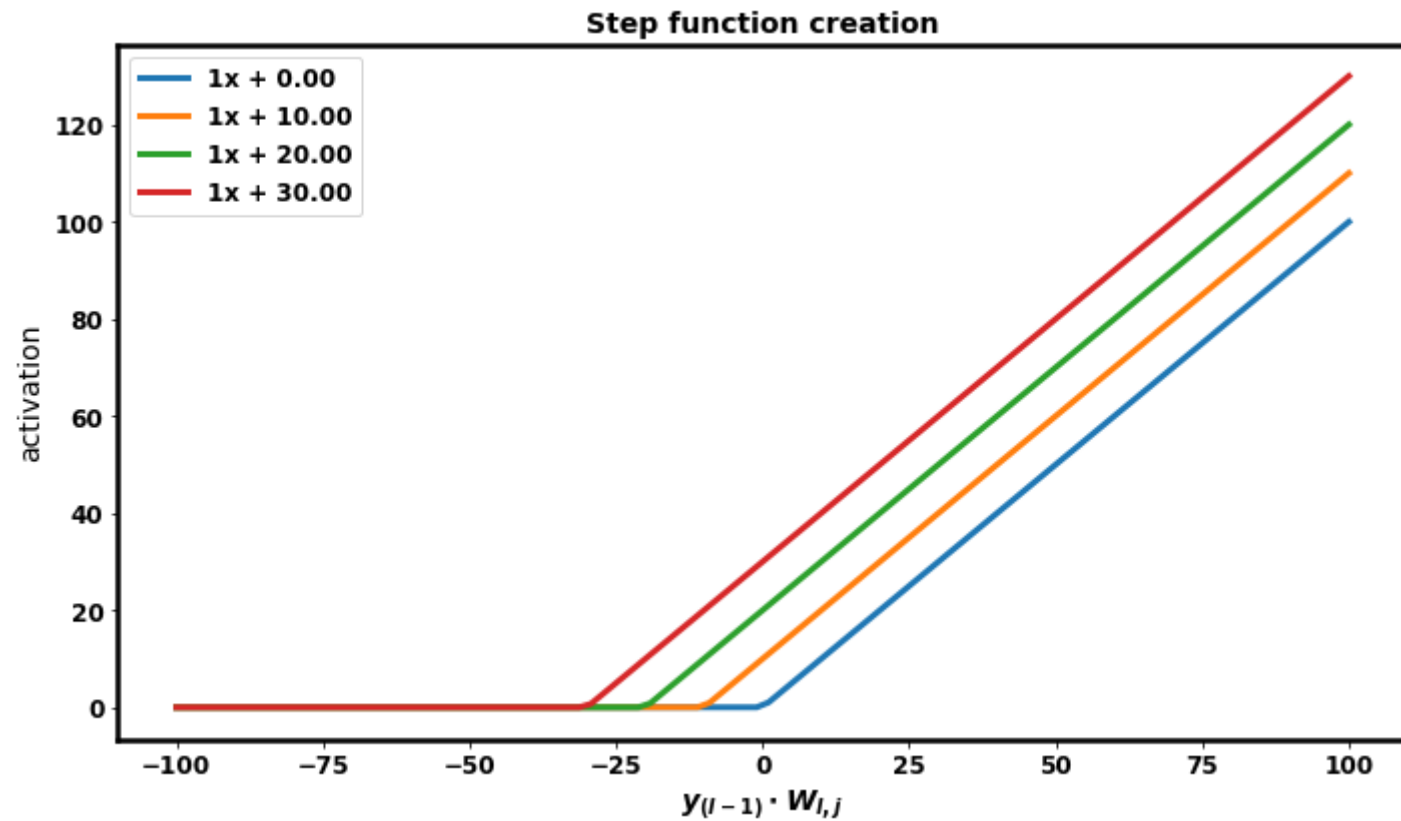
By making slope **W** extremely large, we can approach a vertical line.

```
In [5]:  $\bar{y}$  = 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.

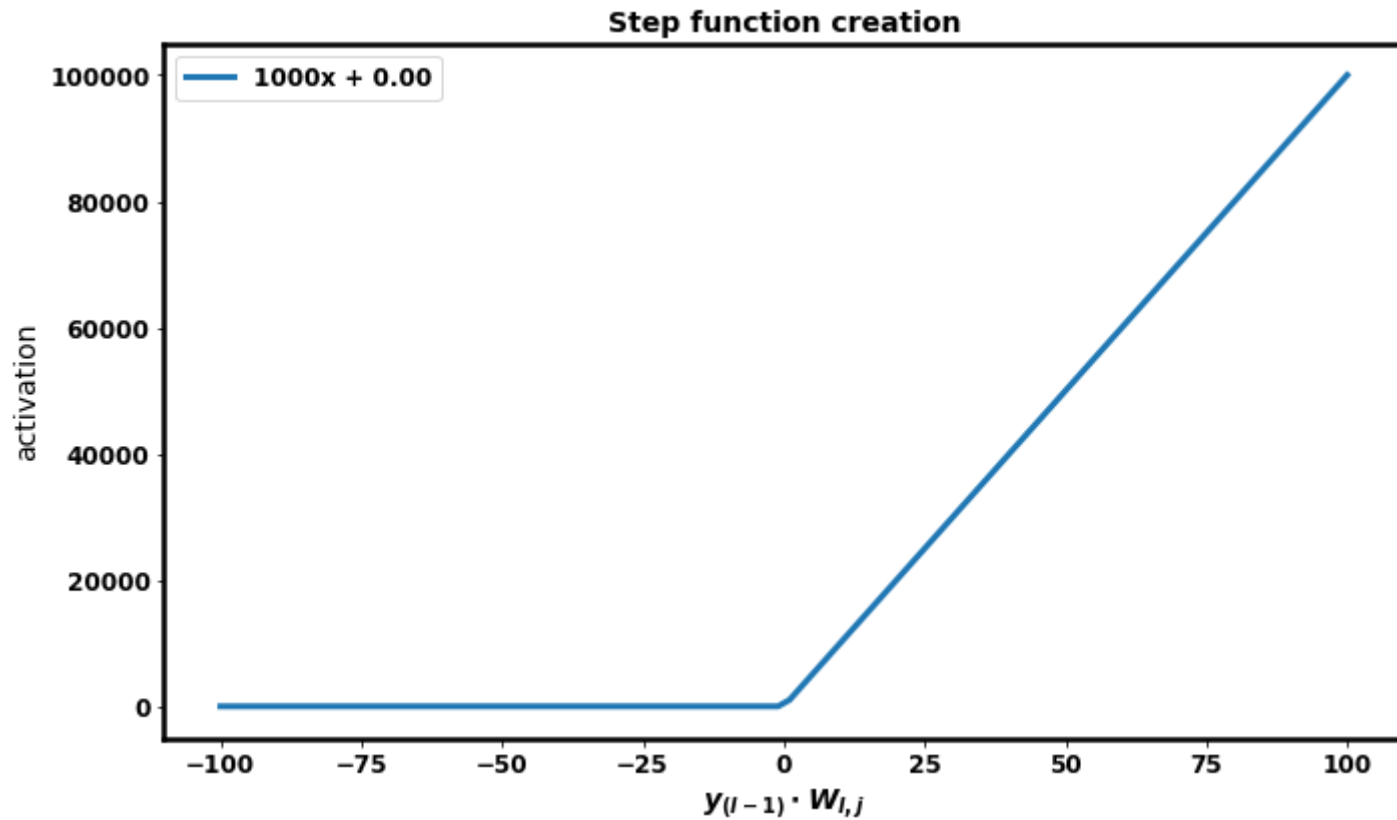
```
In [6]: _ = nnh.plot_steps( [ nnh.NN(1,0), nnh.NN(1,10), nnh.NN(1,20), nnh.NN(1,30), ])
```



With a little effort, we can construct a neuron

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

```
In [7]: slope = 1000  
start_offset = 0  
  
start_step = nnh.NN(slope, -start_offset)  
_= nnh.plot_steps( [ start_step ] )
```





If we create a neuron with intercept "epsilon" from the first neuron

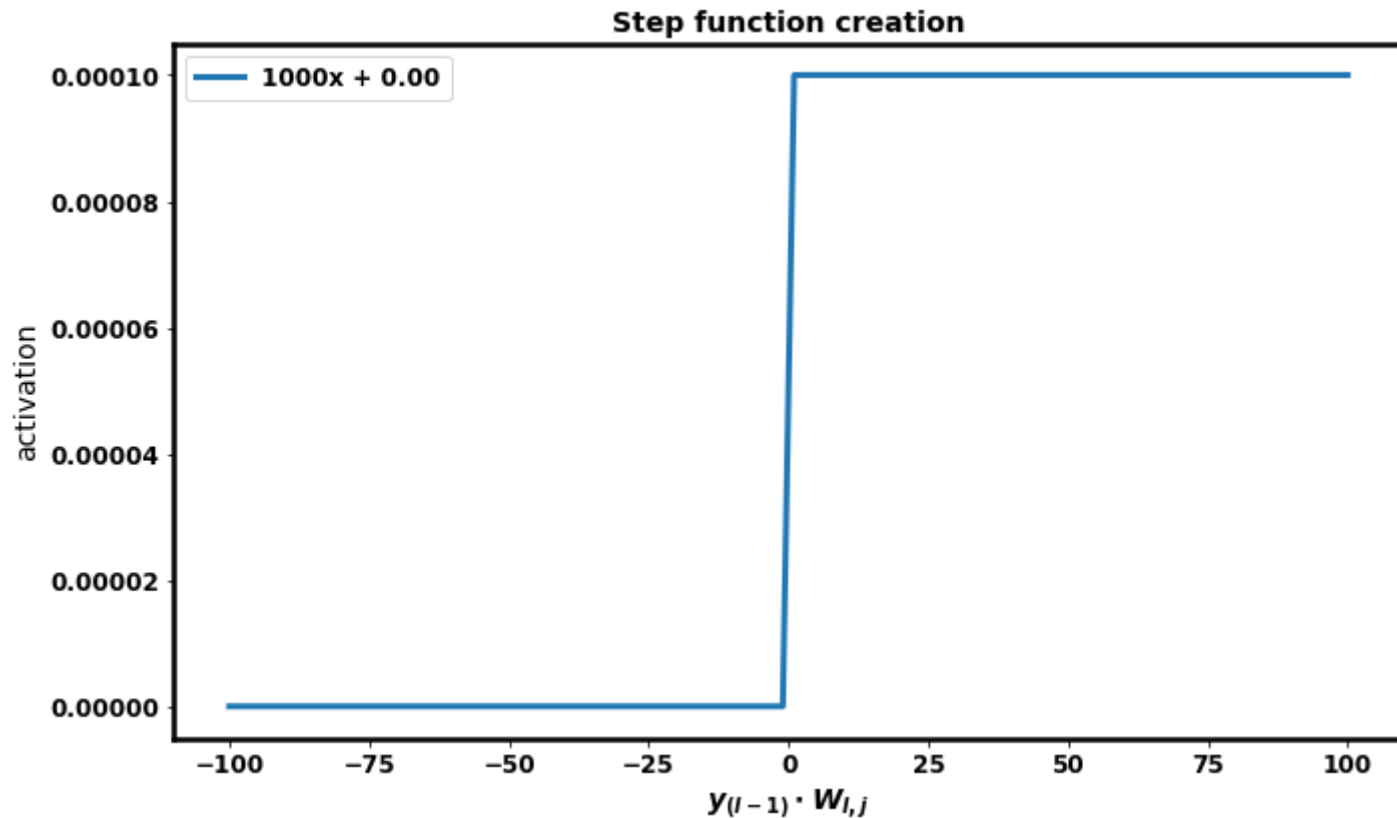
```
In [8]: end_offset = start_offset + .0001  
        end_step = nnh.NN(slope,- end_offset)
```

and add the two neurons together, we can approximate a step function

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



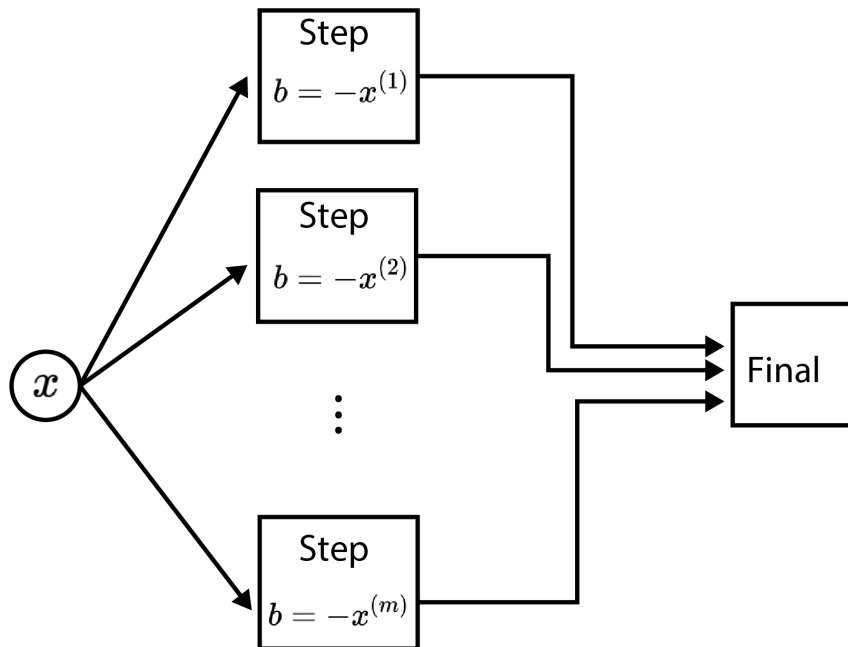
Let us construct  $m$  step neurons

- step neuron  $i$  with intercept  $\mathbf{x}^{(i)}$ , for  $1 \leq i \leq m$

If we connect the  $m$  step neurons to a "final" neuron with 0 bias, linear activation, and weights

$$\begin{aligned}\mathbf{W}_1 &= \mathbf{y}^{(1)} \\ \mathbf{W}_i &= \mathbf{y}^{(i)} - \sum_{i'=1}^{i-1} \mathbf{W}_{i'}\end{aligned}$$

## Function Approximation by Step functions



$$W_1 = y^{(1)}$$

$$W_2 = y^{(2)} - y^{(1)}$$

$$W_i = y^{(i)} - \sum_{i'=1}^{i-1} y^{(i')}$$

We claim that the output of this neuron approximates the training set.

To see this:

- Consider what happens when we input  $\mathbf{x}^{(i)}$  to this network.
- The only step neurons that are active (non-zero) are those corresponding to inputs  $1 \leq i' \leq i$ .
- The output of the final neuron is the sum of the outputs of the first  $i$  step neurons.
- By construction, this sum is equal to  $\mathbf{y}^{(i)}$ .

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

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



```
In [10]: print("Done")
```

Done