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

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

Consider a single neuron with a ReLU activation, computing

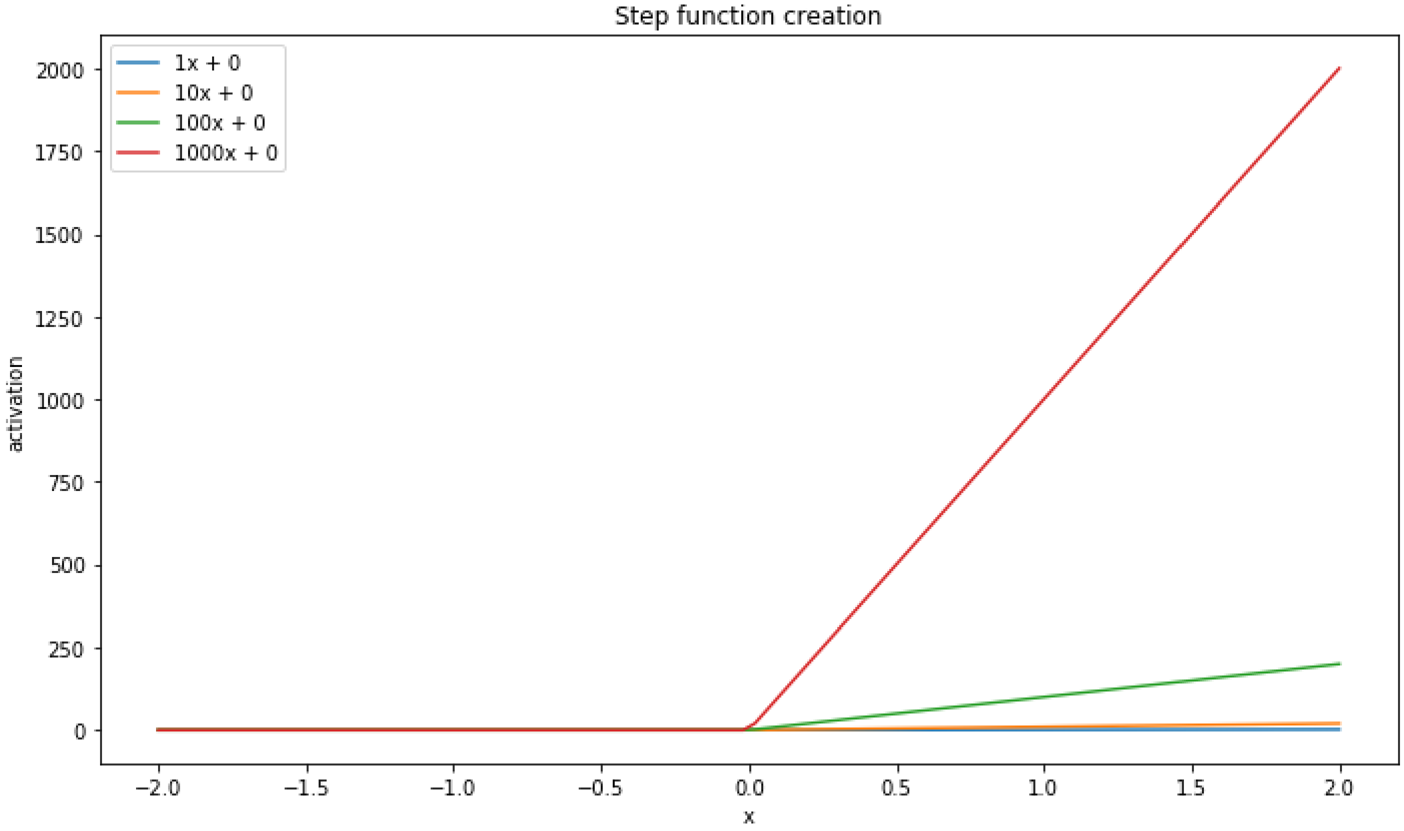
$$\max(0, \mathbf{W}\mathbf{x} + \mathbf{b})$$

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 $\mathbf{b}$.

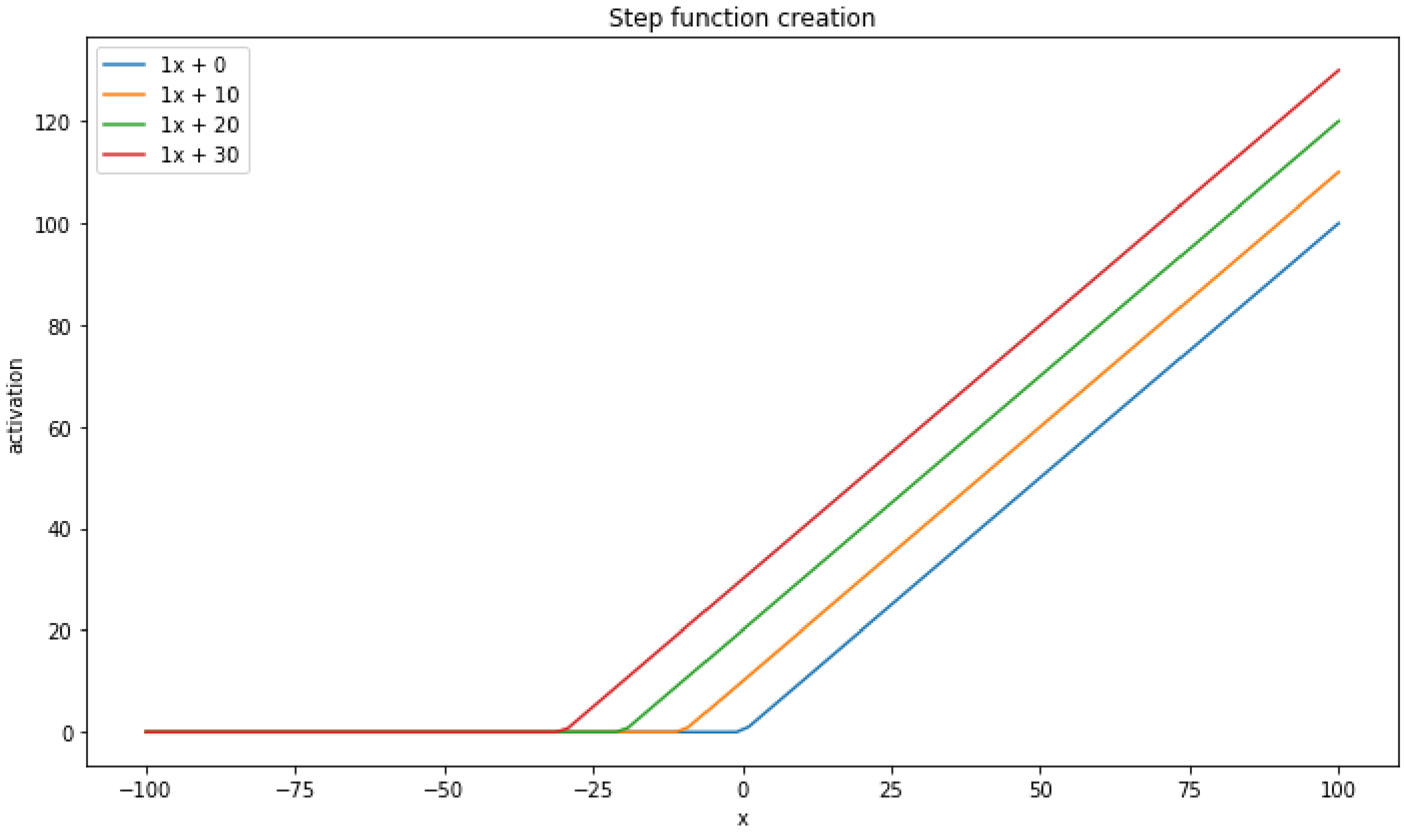
By making slope $\mathbf{W}$ extremely large, we can approach a vertical line.

```
In [4]: _ = 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 [14]: _ = nnh.plot_steps( [ nnh.NN(1,0), nnh.NN(1,10), nnh.NN(1,20), nnh.NN(1,30), ])
```

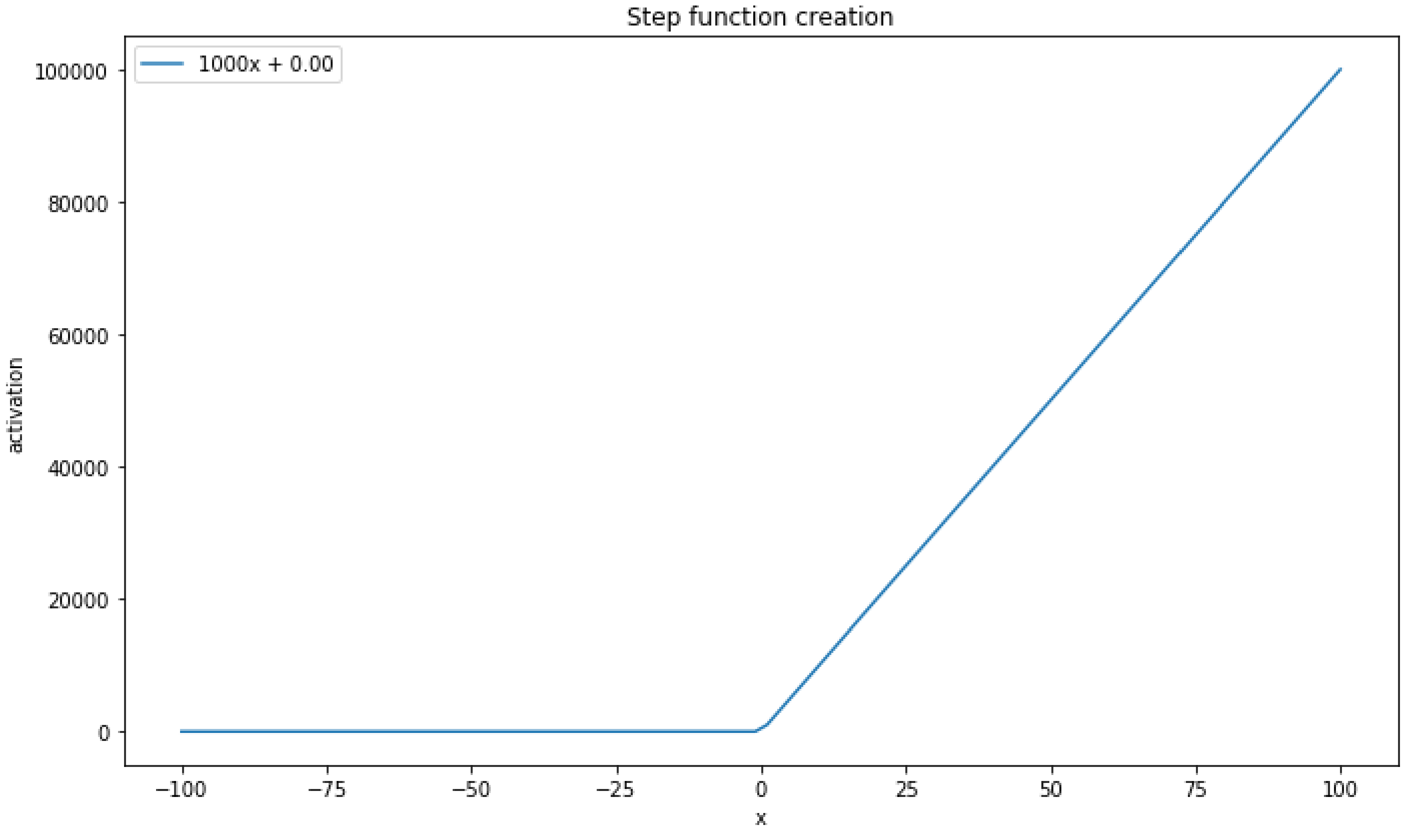


With a little effort, we can construct a neuron

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

In [38]:

```
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 [39]:

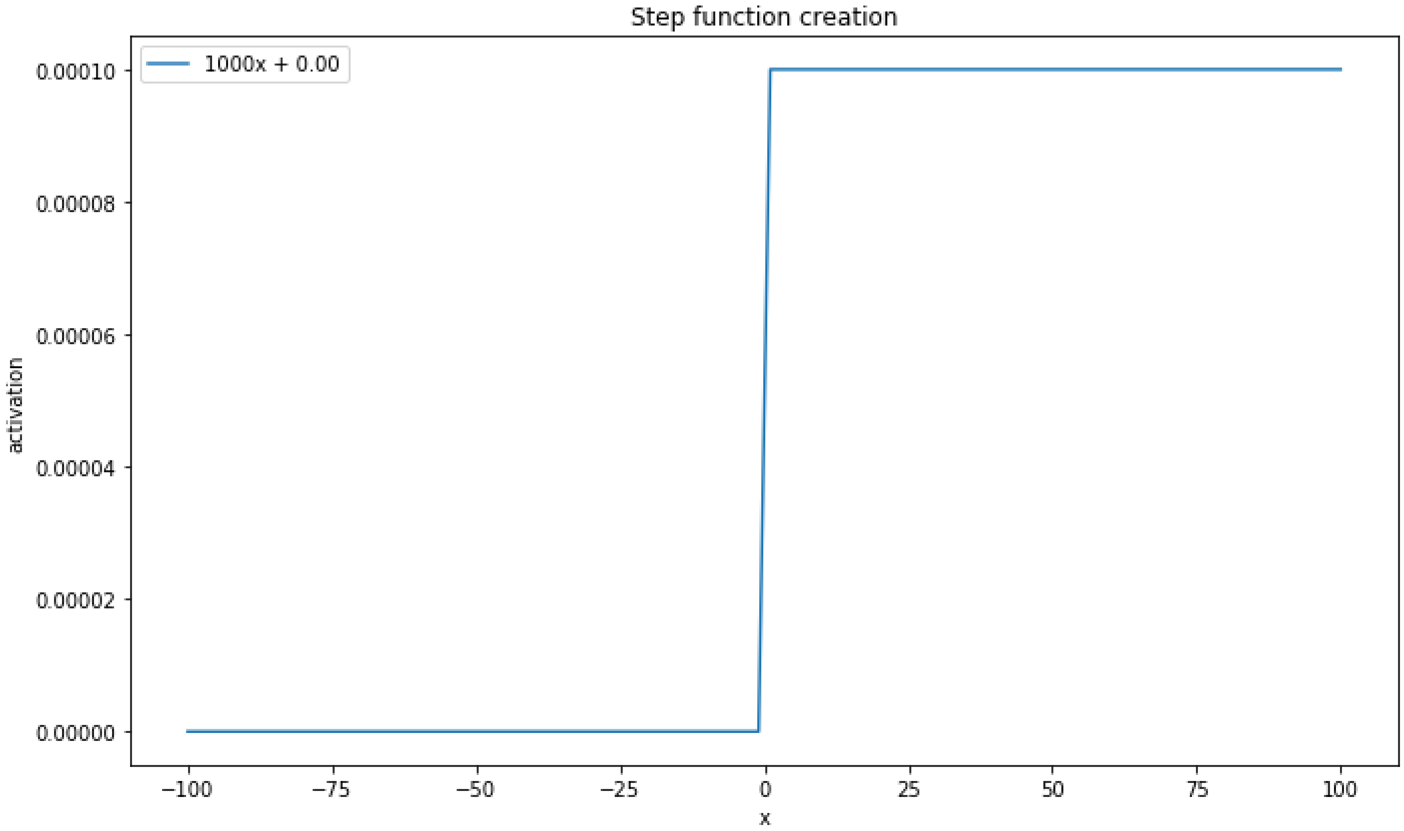
```
end_offset = start_offset + .0001

end_step = nnh.NN(slope,- end_offset)
```

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

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



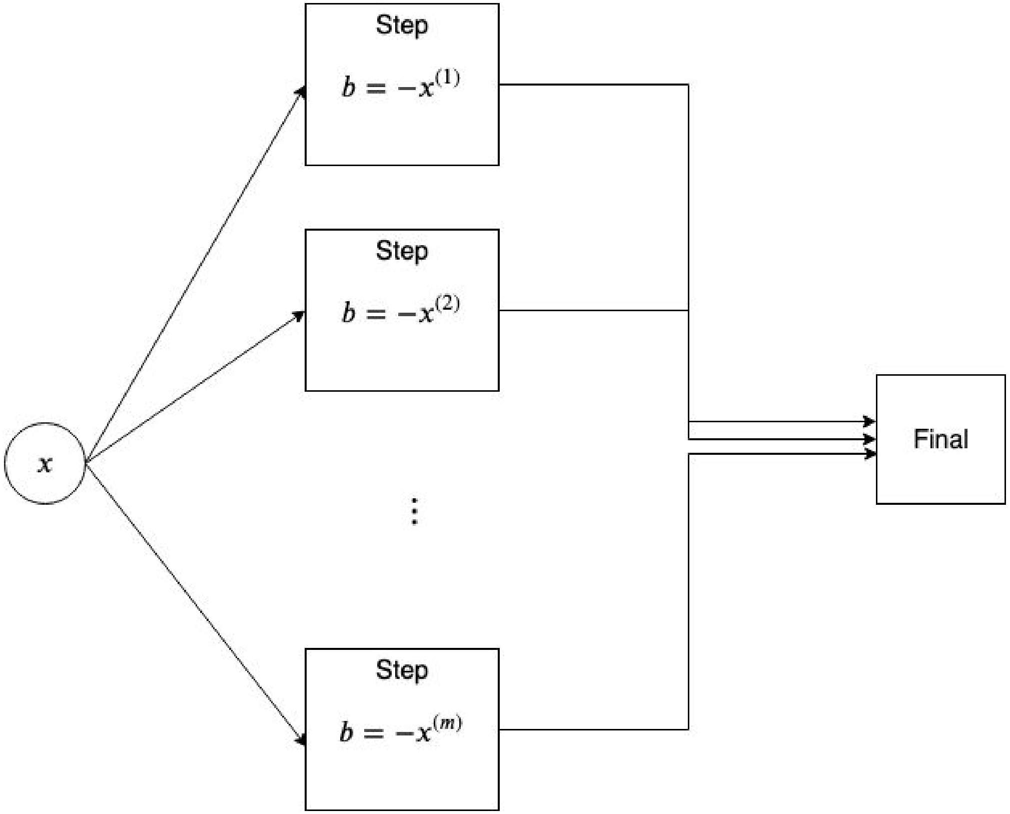
Let us construct m step neurons

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

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

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

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 [7]:

```
print("Done")
```

```
Done
```