From Math to Program

Before introducing more advanced layer types (like the LSTM)

- we want to provide some simple intuition
- for what will appear to be complicated equations that govern these new layer types.

Neural Networks have the flavor of a Functional Program

- A Sequential Model computes the composition of per-layer functions
- Layer l is computing a function $\mathbf{y}_{(l)} = F_{(l)}$

$$F_{(l)}(\mathbf{y}_{(l-1)}; \mathbf{W}_{(l)}) = \mathbf{y}_{(l)}$$

$$F_{(l)} : \mathcal{R}^{||\mathbf{y}_{(l-1)}||} \mapsto \mathcal{R}^{||\mathbf{y}_{(l)}||}$$

If we expand $F_{(l)}$, we see that it is the l-fold composition of functions $F_{(1)}, \ldots, F_{(l)}$

$$\begin{aligned}
\mathbf{y}_{(l)} &= F_{(l)}(\mathbf{y}_{(l-1)}; \mathbf{W}_{(l)}) \\
&= F_{(l)}(\; F_{(l-1)}(\mathbf{y}_{(l-2)};\; \mathbf{W}_{(l-1)});\; \mathbf{W}_{(l)}\;) \\
&= F_{(l)}(\; F_{(l-1)}(\; F_{(l-2)}(\mathbf{y}_{(l-3)};\; \mathbf{W}_{(l-2)});\; \mathbf{W}_{(l-1)}\;); \mathbf{W}_{(l)}\;) \\
&= \vdots
\end{aligned}$$

It turns out that it is not too difficult to endow a Neural Network with familiar *imperative* programming constructs

- `if` statement
- `switch/case` statement

This is sometimes called *Neural Programming*.

So one way of understanding some complicated equations (e.g., for the LSTM)

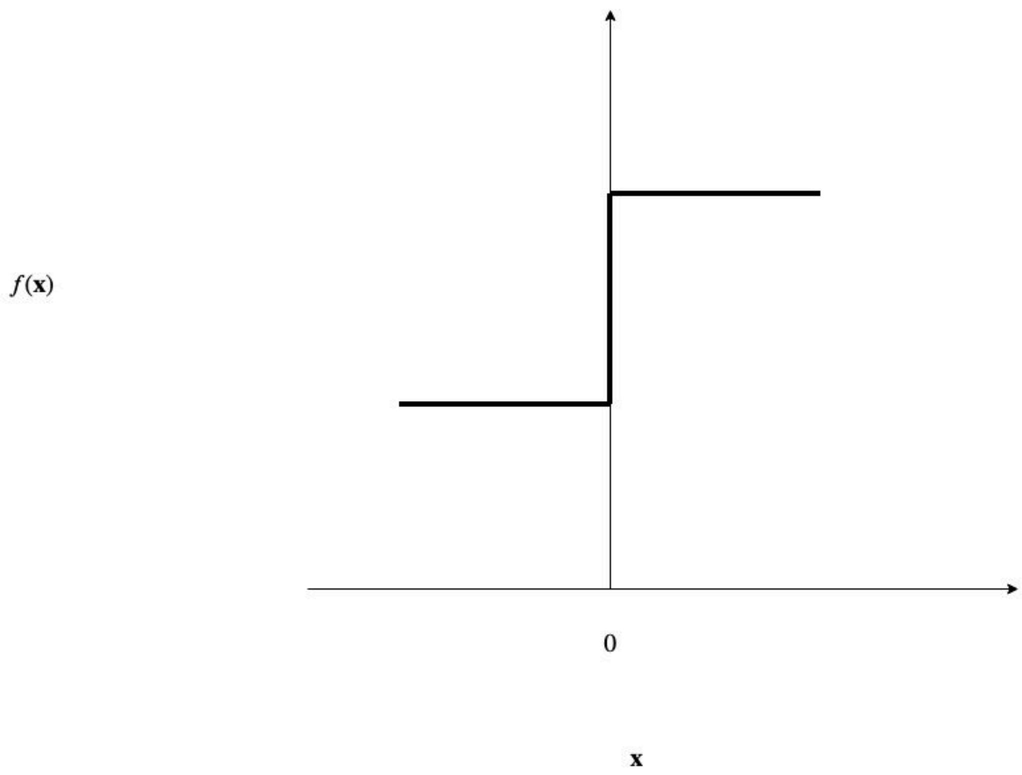
- is to realize that they are encoding "soft" analogs of familiar programming concepts

Binary switches

When we introduced Neural Networks, we argued that their power derived from the ability of Activation Functions

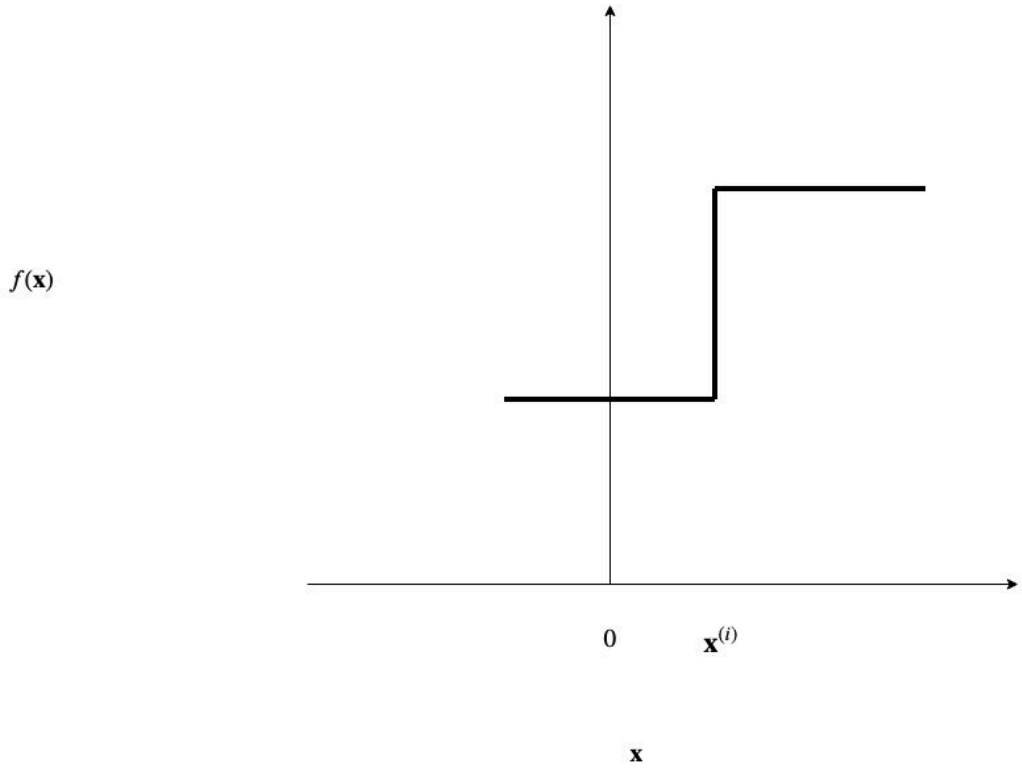
- To act like binary "switches"
- Converting the scalar value computed by the dot product
- Into a True/False answer
- To the question: "Is a particular feature present" ?

Step function: binary switch with threshold 0



By varying the threshold/bias, we can control the region in which the switch is "active"

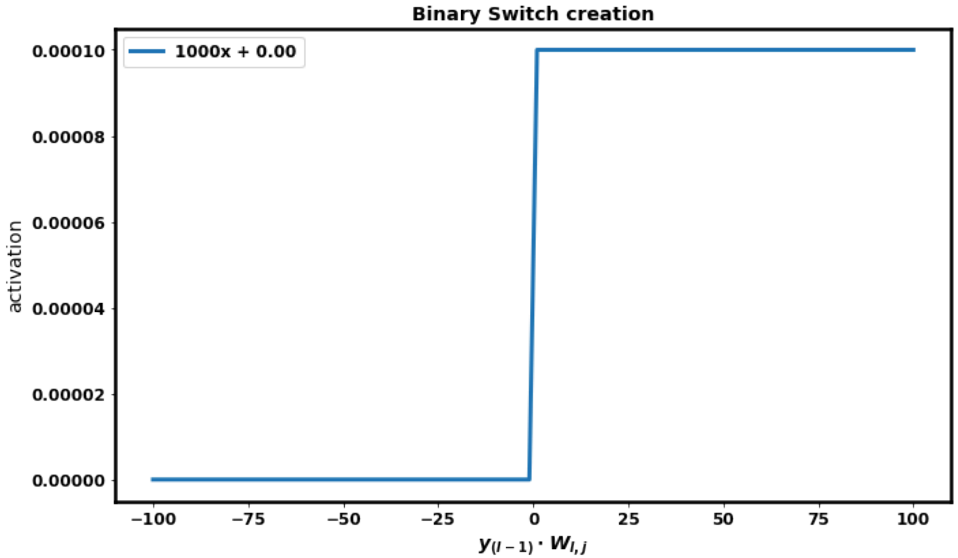
Step function: binary switch with threshold - x^(i)



And, in fact, we showed (Universal_Function_Approximator.ipynb) how to construct a very precise approximation of a binary switch:

In [5]:

```
fig, ax = nnh.step_fn_plot()
```



Neurons as statements

With the ability to implement a binary switch

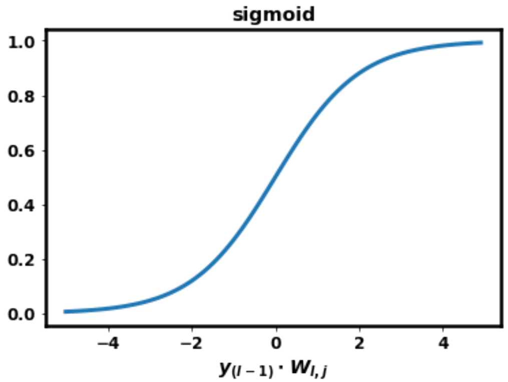
- We can construct Neural Networks
- With elements that look like primitive statements of a programming language

Rather than building a true step function

- We will settle for the approximation offered by the Sigmoid function σ

In [6]:

```
_= nnh.sigmoid_fn_plot()
```



This is more than laziness or convenience

- The step function is **not** differentiable
- The sigmoid function **is** differentiable

Recall that Gradient Descent is the tool we use to train Neural Networks

- Hence it is important that our functions be differentiable !

Thus the switches (analogous to conditions in an `if` statement)

- Will not output one of True/False
- But rather a "soft" approximations

"If" statements - Gates

Suppose we want a Neural Network to

- Compute a (vector) output $\mathbf{y}$
- That takes on vector value T if some condition g is `True`
- And F otherwise.

This would be trivial in any programming language having an `if` statement:

```
if (g):
    y = T
else:
    y = F
```

Let's show how to construct the `if` statement with just a little arithmetic.

Suppose scalar $g \in \{0, 1\}$ was the value output by a switch.

Then

$$\mathbf{y} = (g * \mathbf{T}) + (1 - g) * \mathbf{F}$$

does the trick.

In general, we tend to compute vectors rather than scalars.

Let

- $\mathbf{g}, \mathbf{y}$ be vectors of equal length
- $\mathbf{T}, \mathbf{F}$ be vectors of equal length (not necessarily the same as $\mathbf{g}, \mathbf{y}$)
  - So elements of $\mathbf{y}$ have length $||\mathbf{T}|| = ||\mathbf{F}||$

We will construct a "vector" `if` statement

- Making a conditional choice for *each element* of $\mathbf{y}$, independently.
$$\mathbf{y}_j = (\mathbf{g}_j * \mathbf{T}) + (1 - \mathbf{g}_j) * \mathbf{F}$$

Letting

- $\otimes$ denote element-wise vector multiplication (*Hadamard product*)
- $\sigma(\ldots)$ be a sigmoid approximation of a binary switch

The following product (almost) does the trick

$$\begin{aligned}\mathbf{g} &= \sigma(\ldots)\\ \mathbf{y} &= \mathbf{g} \otimes \mathbf{T} + (1 - \mathbf{g}) \otimes \mathbf{F}\end{aligned}$$

It is only "almost"

- Because the sigmoid only takes a value in the range $[0, 1]$
- Rather than exactly either 0 or 1

So $\mathbf{g}$ is a "soft" condition rather than a hard (either True or False) condition.

- This means that $\mathbf{y}$ will be a blend of $\mathbf{T}$ and $\mathbf{F}$

What we have is

- A continuous (soft) decision $\mathbf{g}$.
- That creates a vector `if`
- Whose elements are *mixtures* of $\mathbf{T}$ and $\mathbf{F}$

This is the price we pay for having $\mathbf{g}$ be differentiable !

Note that the individual elements of vector $\mathbf{y}$ are independent

- $\mathbf{y}_j$ is influenced only by $\mathbf{g}_j$
- The synthetic features represented by $\mathbf{y}$ are not dependent on one another.
- Most importantly: the derivatives of each feature are independent

"Switch/Case" statements

We can easily generalize from a two-case `if` to a `switch/case` statement with $||\mathbf{C}||$ cases.

Suppose we need to set $\mathbf{y}$ to one value from among multiple choices in $\mathbf{C}$

$$\mathbf{g} = \mathrm{softmax}(\ldots)$$
$$\mathbf{y} = \mathbf{g} \otimes \mathbf{C}$$

The *softmax* function

- Was introduced in Multinomial Classification
- Computes a vector (of length $||C||$) values
- With each element being in the range $[0, 1]$
- And summing to 1

We refer to $\mathbf{g}$ as a *mask* for $\mathbf{C}$.

The `if` statement is a special case of the `switch/case` statement where

$$\mathbf{C} = \begin{bmatrix} \mathbf{T} \\ \mathbf{F} \end{bmatrix}$$

Soft Lookup

The "approximate case" statement we created has an interesting application:

- A lookup table (`dict` in Python) that does soft matching

Whereas an ordinary `dict` in Python returns an undefined value when the query key q does not match any key in the dictionary

- A soft lookup table M (*context sensitive memory*) returns a *weighted sum* of the values associated with all keys

Details can be found [here (Context_Sensitive_Memory.ipynb)](Context_Sensitive_Memory.ipynb)

Conclusion

We wanted to show that, in concept

- We could create the logic of a simple imperative program
- Using the machinery of Neural Networks

The only catch was

- We cannot use true binary logic (hard decisions)
- All choices are *soft*
- In order to preserve differentiability
- Which is necessary for training with Gradient Descent

This background will facilitate our explanation of more advanced Layer types.

In [7]:

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

Done