Activation Layers

Consider a sequence of layers

- Each layer performing a dot product
- With **no* activation function (or equivalently one that is the identity function)

A dot product is equivalent to a linear transformation (e.g, matrix multiplication).

- A sequence of dot products can be re-written as a matrix multiplication
- Involving the product of the individual matrices

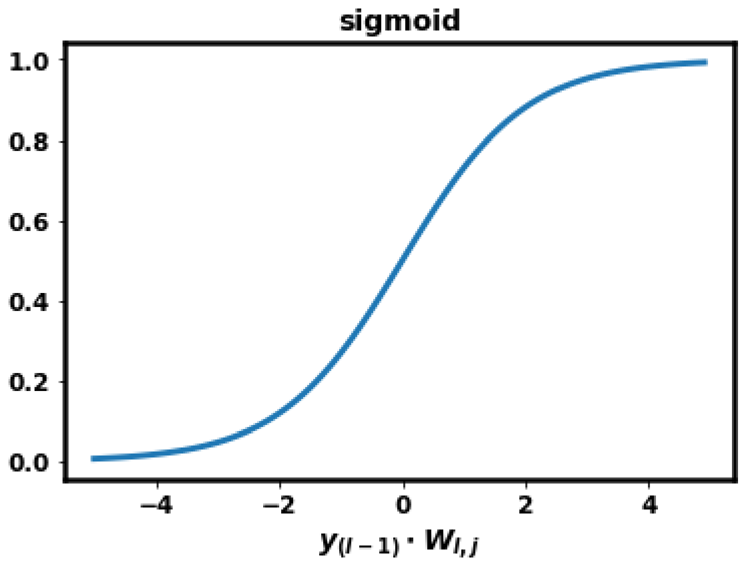
That is: the composition of linear functions is just a linear function.

Thus, the layer architecture would have no real purpose *without* the non-linear activation functionss.

Let's examine the sigmoid activation function

In [5]:

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



A binary switch would

- Output a 1 if the dot product exceeded a *threshold* (0 in the above plot)
- Output a 0 otherwise

The sigmoid is an approximation of the binary switch.

- Maps most values of the dot product to 0 or 1
- This is why it was useful for binary classification

It almost acts as a True/False gate for the question: "Does $\mathbf{y}_{(l-1)}$ have some particular feature ?"

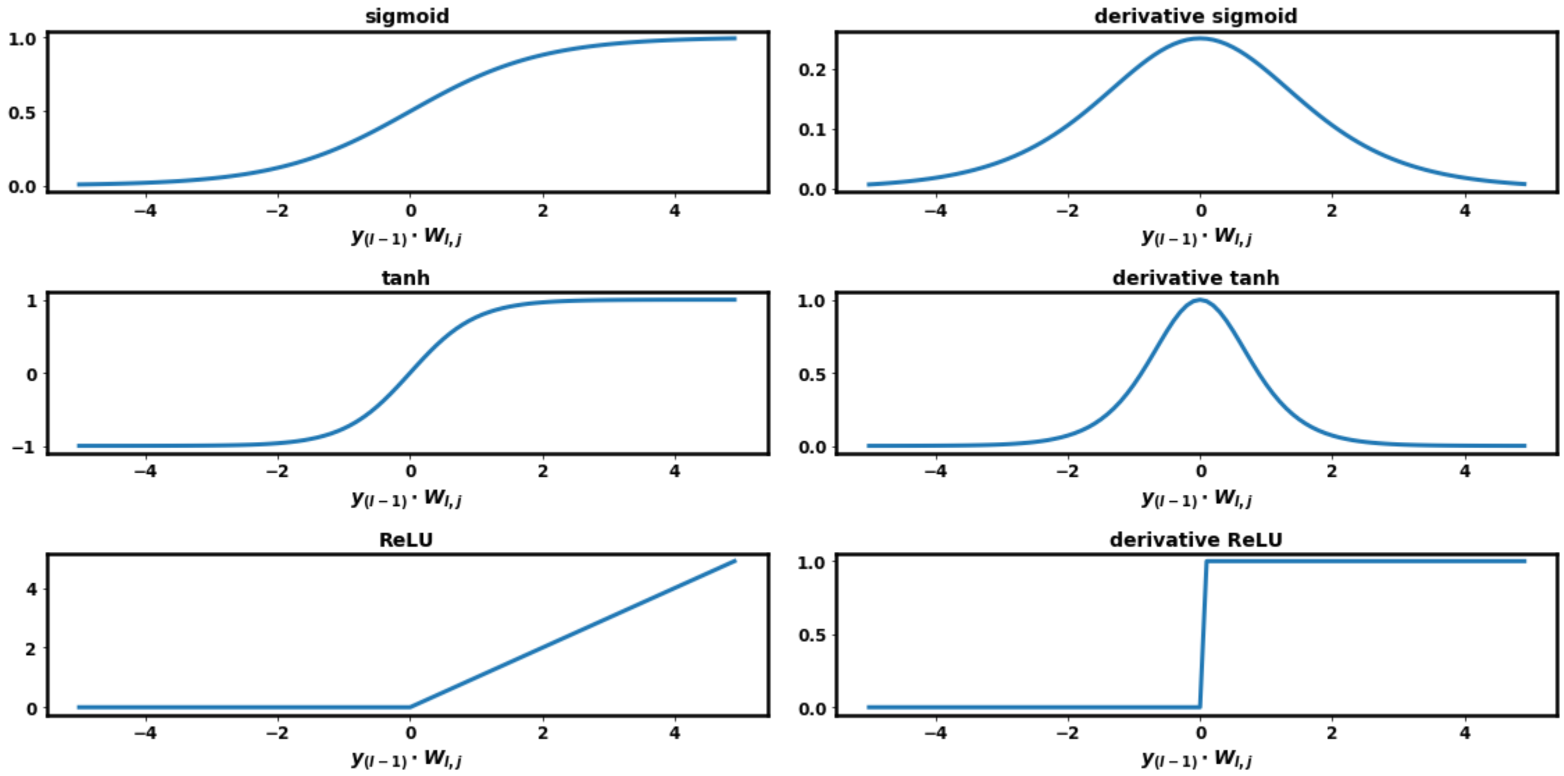
The ability to turn a continuous value into a (near) discrete binary choice is the power of the non-linearity.

The signmod is not the only choice for a non-linear activation function.

Let's examine some others, along with their derivatives

In [10]:

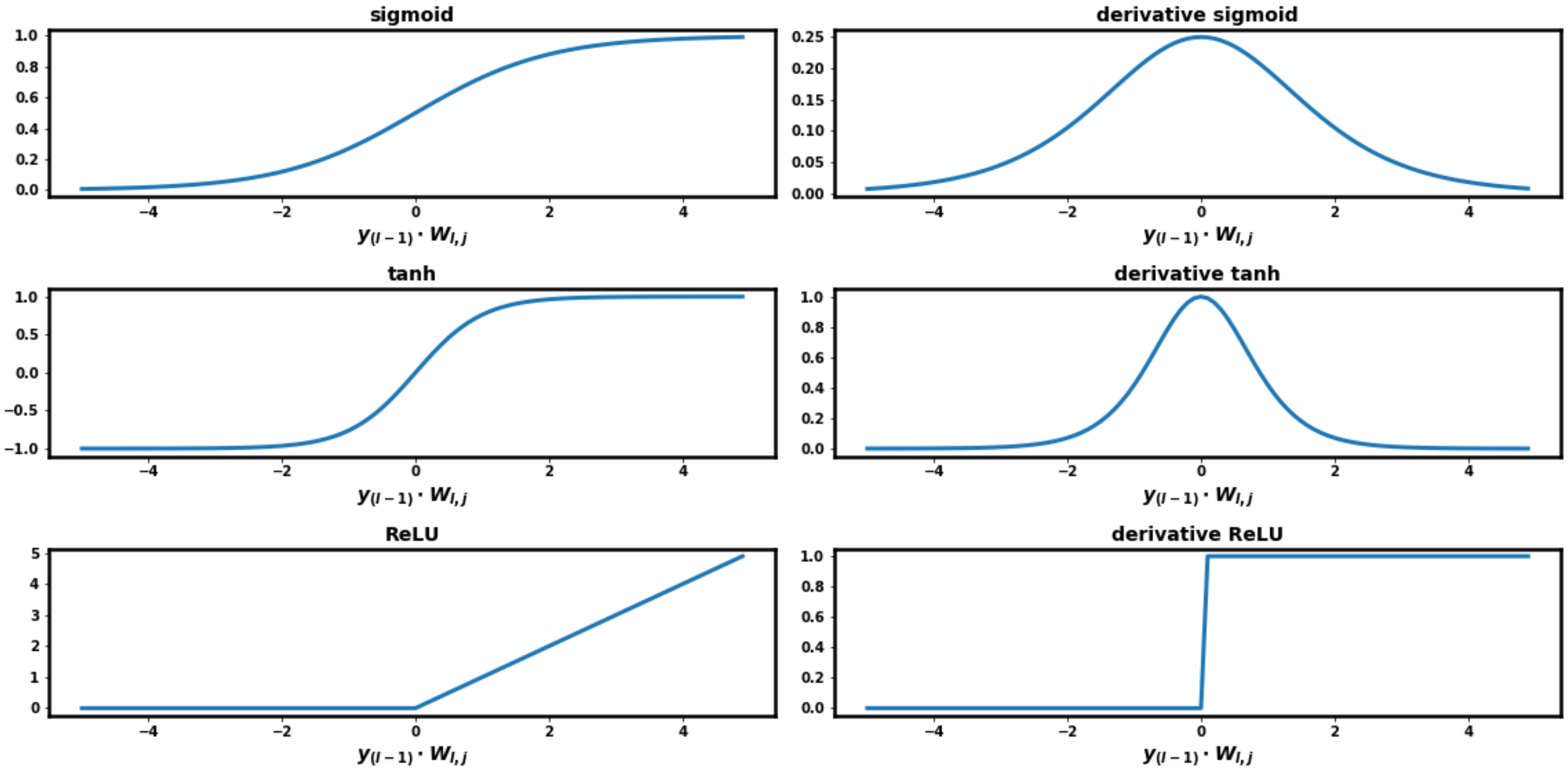
```
fig, _ = nnh.plot_activations( np.arange(-5,5, 0.1) )
```



Which activation function should we choose ?

The choice depends on what the layer needs to accomplish:

- The Head layer in a Binary Classification task has output range matching the Sigmoid: $[0, 1]$
- The Head layer in a Regression task may have **no** activation: output range unbounded
- If a zero-centered output is required the $\tanh$ with range $[-1, +1]$ is a good match
- If we want to create a gate/switch that turns "on" at a given threshold: the ReLU is a good match



Although it is hard to appreciate at the moment

- Managing derivatives is **one of the key** insights that enabled the explosive growth of DL !

We will explore this more in a subsequent lecture; for now:

- A zero derivative can hamper learning that uses Gradient Descent (tanh, sigmoid)
- The magnitude of the derivative modulates the "error signal" during back propagation
    - so smaller maximum values diminish the signal more than larger ones (sigmoid)

The other thing to notice are the derivatives:

- both the tanh and sigmoid have large regions, at either tail, of near zero derivatives
- the derivative of the sigmoid has a maximum value of about 0.25

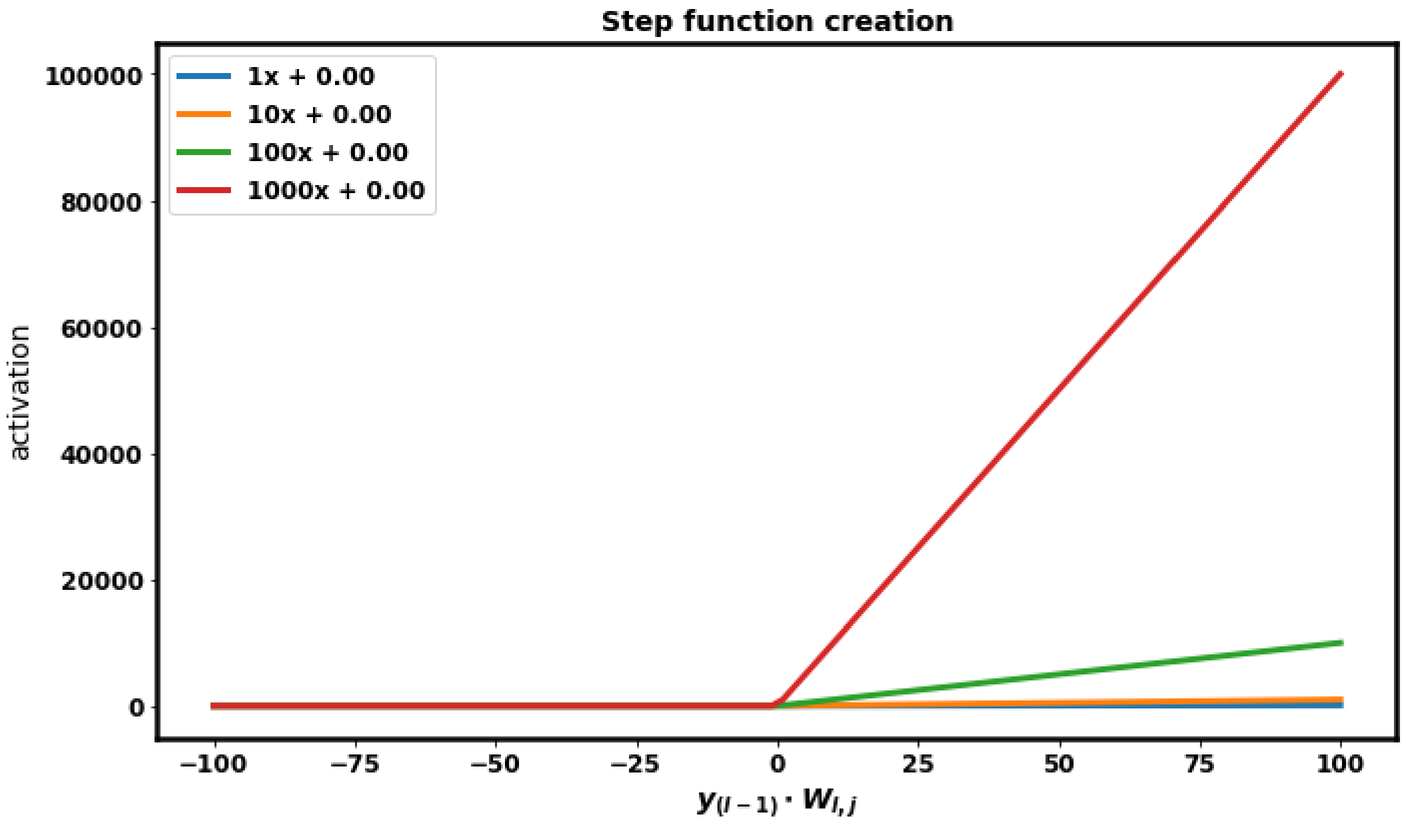
In most cases, we will use the ReLU activation function.

- Half of the domain results in a non-zero derivative, which facilitates learning

As a refresher: here is what the ReLU function looks like for various values of $\mathbf{W}$.

First, let's vary W_1, the "slope"

```
In [7]: _ = nnh.plot_steps( [ nnh.NN(1,0), nnh.NN(10,0), nnh.NN(100,0), nnh.NN(1000,0),
]) 
```



Varying the threshold: the bias

All of the activation functions approximate a binary switch

- They divide the range of the dot product into two regions: 0 and non-0
- The division is centered around a threshold value of the dot product (0)

We will show how to vary the threshold.

By doing so, we can show (see Deep Dive on *Universal Approximation Theorem*)

- How to approximate a function of the dot product
- With aribtrarily complex shape
- Via a piece-wise approximation
    - Each threshold contributes one linear segment

You will often see the equation for unit j of layer l written with an extra term $\mathbf{b}_j$

$$\mathbf{y}_{(l),j} = \mathbf{y}_{(l-1)} \cdot \mathbf{W}_{(l),j} + \mathbf{b}_{(l),j}$$

$\mathbf{b}_{(l),j}$ is called the *bias* of unit j of layer l.

The bias term $\mathbf{b}_{(l),j}$ seems like an isolated annoyance.

Far from it !

It controls the region at which the activation functions "switches" from 0 to 1

$$(\mathbf{y}_{(l-1)} \cdot \mathbf{W}_{(l),j} + \mathbf{b}_{(l),j} > 0) \text{ when } (\mathbf{y}_{(l-1)} \cdot \mathbf{W}_{(l),j} > -b)$$

Rather than keeping $\mathbf{b}_{(l),j}$ apart from the dot product

- We apply a "trick" familiar from the Classical Machine Learning part of the course
- We imagine augmenting input $\mathbf{y}_{(l-1)} = [\mathbf{y}_{(l-1),1}, \ldots, \mathbf{y}_{(l-1),n_{(l-1)}}]$
- With element $\mathbf{y}_{(l-1),0} = 1$
- Such that $\mathbf{y}_{(l-1)} = [1, \mathbf{y}_{(l-1),1}, \ldots, \mathbf{y}_{(l-1),n_{(l-1)}}]$
- Setting $\mathbf{W}_{(l),j,0} = \mathbf{b}_{(l),j}$

Thus the dot product

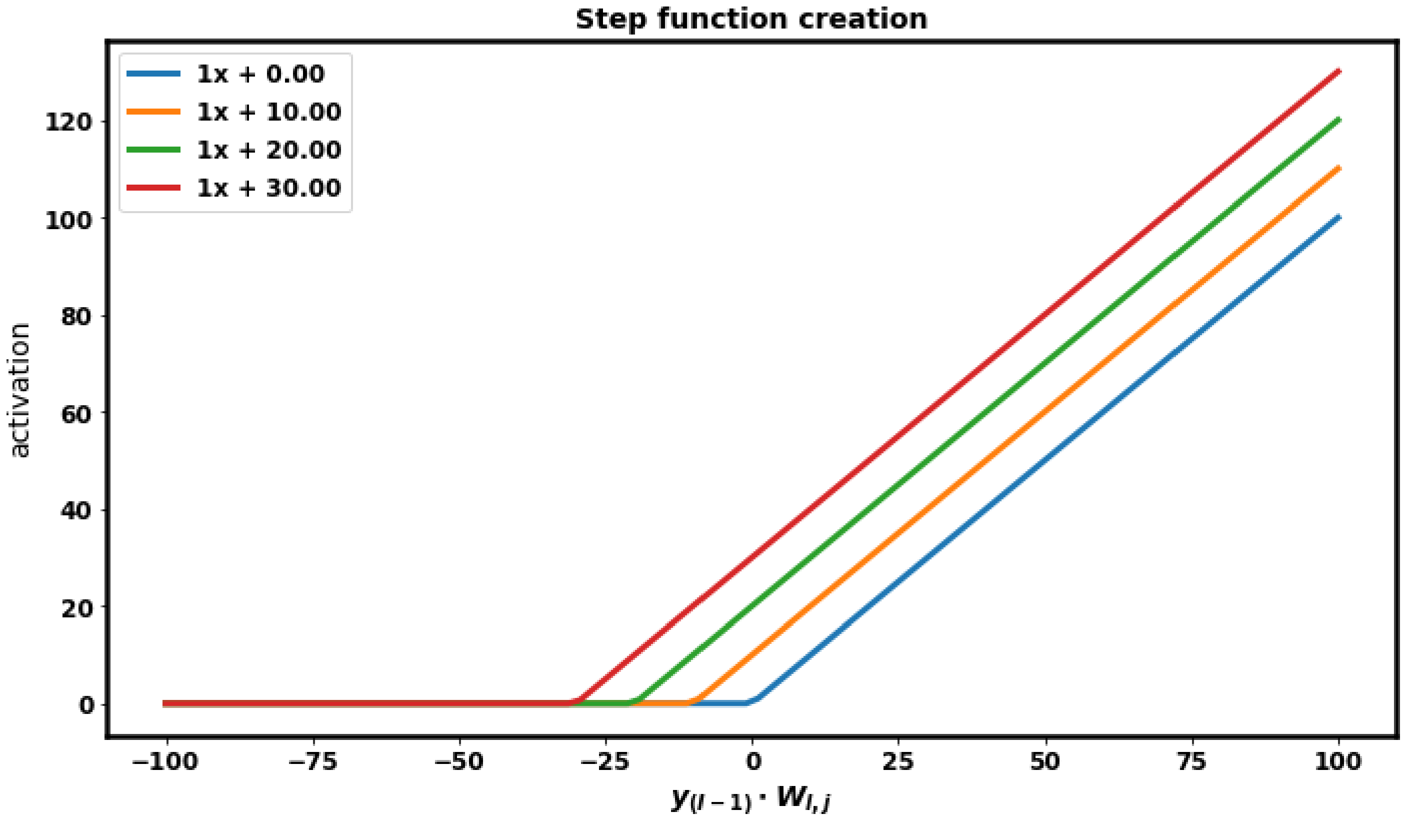
$$\mathbf{y}_{(l-1)} \cdot \mathbf{W}_{(l),j}$$

becomes equal to

$$\mathbf{y}_{(l-1)} \cdot \mathbf{W}_{(l),j} + \mathbf{b}_{(l),j}$$

You can see how varying $\mathbf{W}_0$ (the "intercept") shifts the function left/right

In [8]:
```
_ = nnh.plot_steps( [ nnh.NN(1,0), nnh.NN(1,10), nnh.NN(1,20), nnh.NN(1,30), ])
```



Other activation functionss

Linear/Identity

Softmax Layer

Leaky ReLU Layer

In [9]:

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

```
Done
```