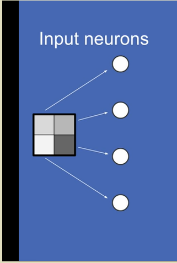
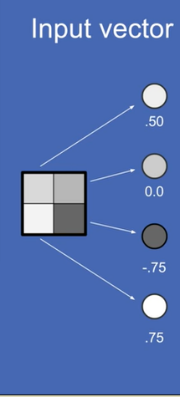
**How Neural Networks work**

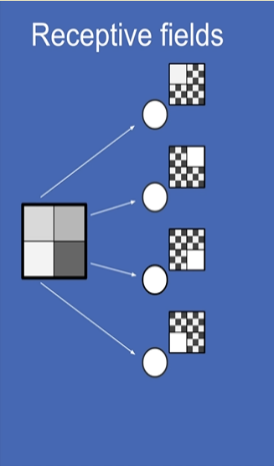
**2020-07-10**

Notes and pictures taken from: <https://e2eml.school/how_neural_networks_work.html>

Let’s assume you have a 4-pixel image. Each pixel is broken out into *input neurons*.

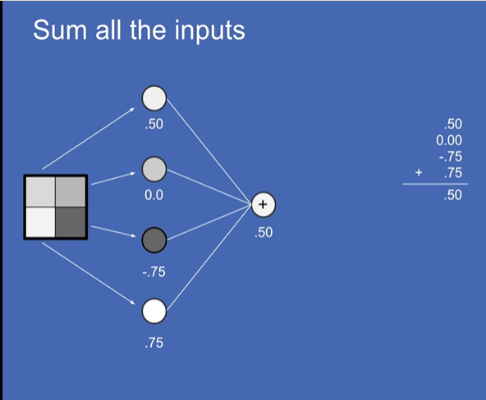


A number is assigned to each pixel as a function of the brightness (1) or darkness (-1). This creates an input vector or array – this is a list of numbers that represents your inputs.

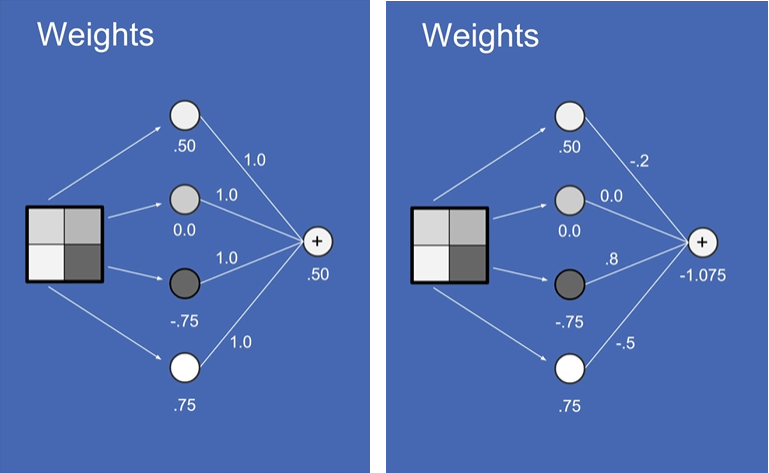


It’s useful to think about the receptive field of a neuron. Essentially this translates to: ‘what set of inputs make the value of this neuron as high as it could possibly be?’. For input neurons, this is easy because each one is associated with 1 input pixel *only*. When a given pixel is completely white, then the value of the corresponding input pixel is as high as it can go.

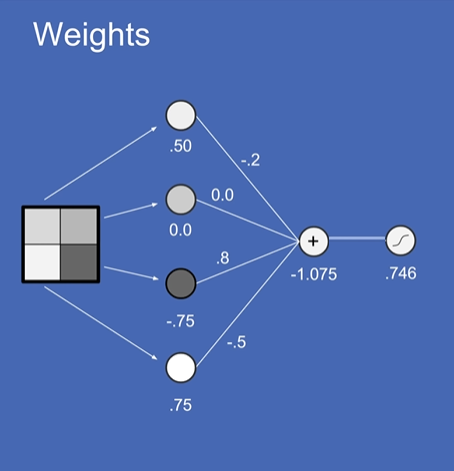
The black and white checkered area are regions that the input neuron *does not* care about. It only cares about it’s sole input pixel and not neighboring pixels.

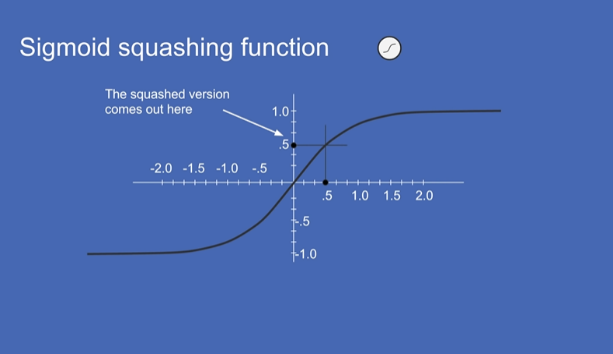


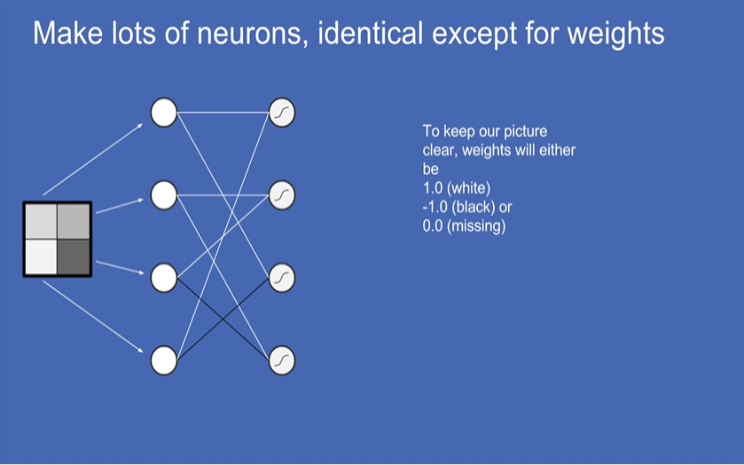
To build a neural network, we build a neuron. First, all the values of the input neurons are summed. In our example, the sum of all input neurons equals 0.5. This is the value of the neuron.

Each of the connections are weighted, meaning they’re multiplied by a number. The weights can be anywhere between -1 and 1. If something has a weight of 0, the value is effectively ignored.

Notice that after weighting (image on the right), the value of the neuron is completely different. The value of the neuron at this point may be outside of the -1 to 1 range that the input neurons were bounded to.

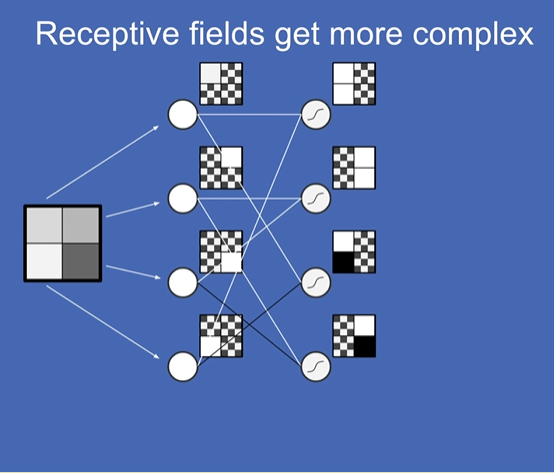
Next, the value of the neuron is normalized using a sigmoid signature, i.e., ‘S-shaped’ function, that bounds the value between -1 and 1. In this case, we get a squashed value of 0.746. More on sigmoid squashing function below.

As you can see from the S-shaped function, the nueron values will always fall between -1 and 1 after being normalized or ‘squashed’. This ensures that the neurons value is always bounded. This is helpful for keeping the computations in the neural network stable.



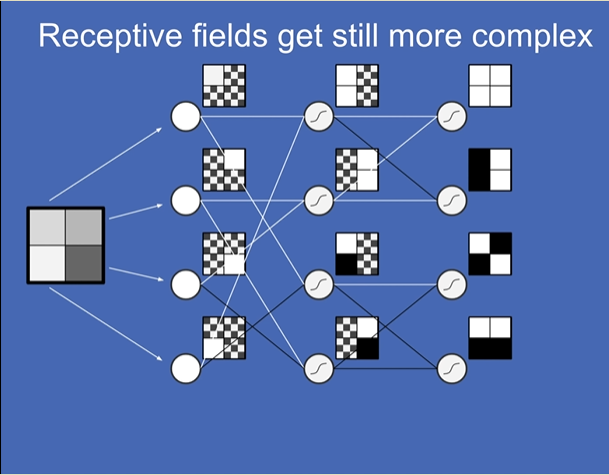
Instead of just having 1 neuron, let’s assume that we have many neurons. There may be 4, 400, 400000+ etc. For here, we’ll assume we have 4 neurons and that the weights are either -1, 0, or +1

In actuality, all neurons are attached to all of the input neurons and they all have some weight between -1 and +1.

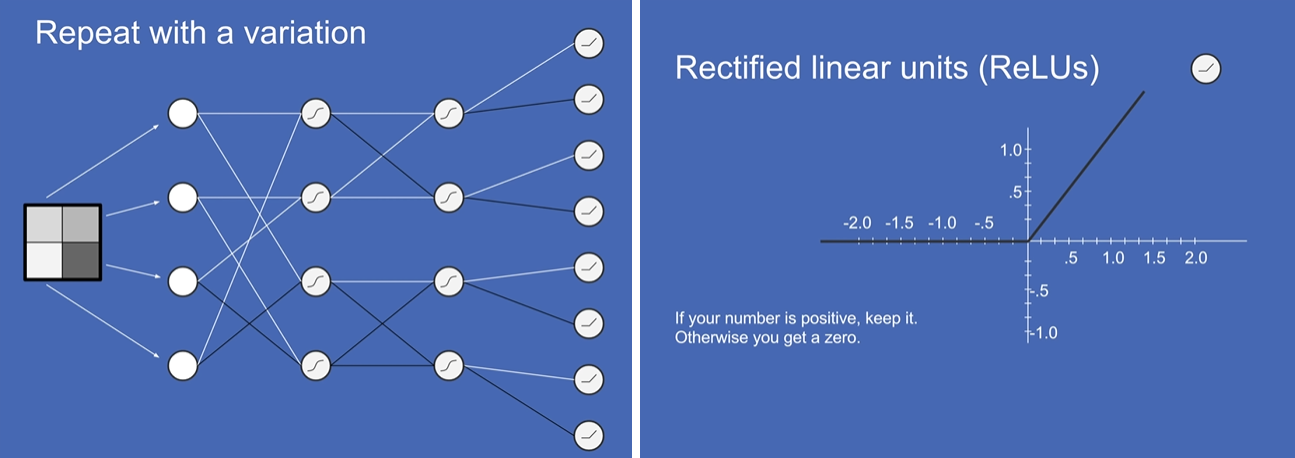
When we create the first *layer* of the neural network, the receptive fields get more complex. In this example, each neuron is a combination of 2 input neurons. So the pixel values that create the first layer of neurons now look like hairs of pixels, either all white or all black, depending on the weights.

Looking at the receptive field of the neuron in the top-right, we can see that it’s a combination of the top-left neuron and the bottom-left neuron. Individually, both neurons are white, and combined they create a receptive field showing two white pixels that are vertically-stacked.

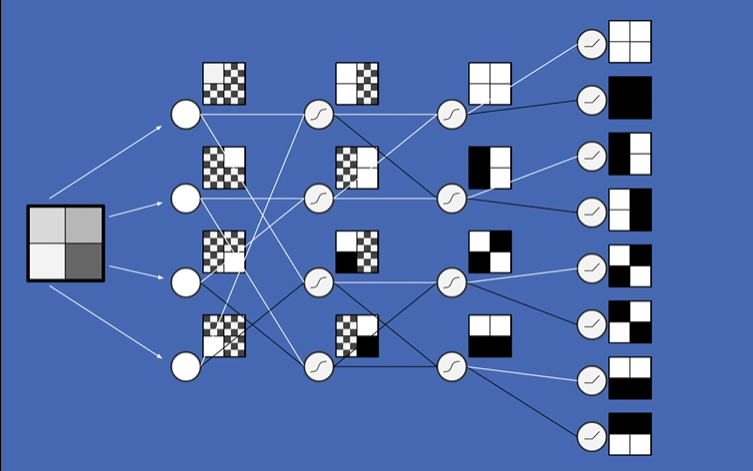
We can then use the output from one set of neurons to initialize the next layer of neurons. The output of one layer is the input of the next layer. We can repeat this N number of times to create as many layers as we want. Each time the receptive fields get more complex.



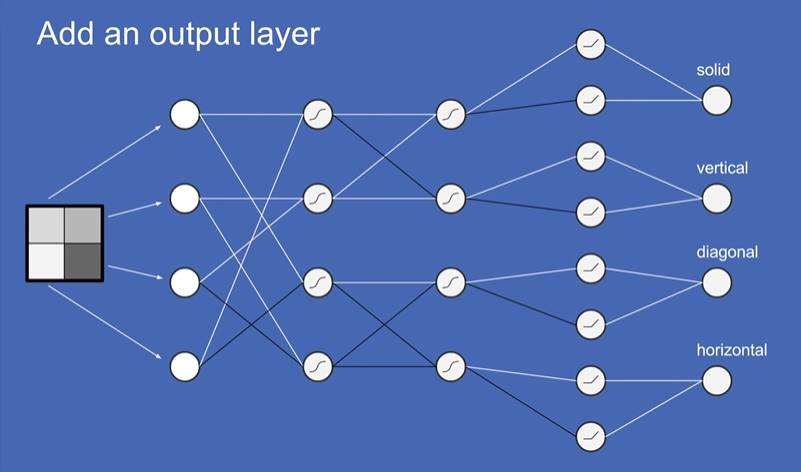
We can create another layer, with a variation. This time, notice that the sigmoid squashing is not present in these new neurons. We are now using a function called a Rectified Linear unit (ReLUs). You do the weighted sum of all your inputs but instead of squashing, you use ReLUs, which essentially translates to elimination of negative values.



After we do this we get a collection of receptive fields AND their opposites.



Finally, when we’ve created as many layers with as many neurons as we want, we create an output layer. In this example we have 4 outputs that we’re interest in. Is the image solid, vertical, diagonal, or horizontal?



Let’s walk thru an example of how this works. Let’s say we start with the input image shown at the start, dark pixels on-top, white pixels on the bottom.

