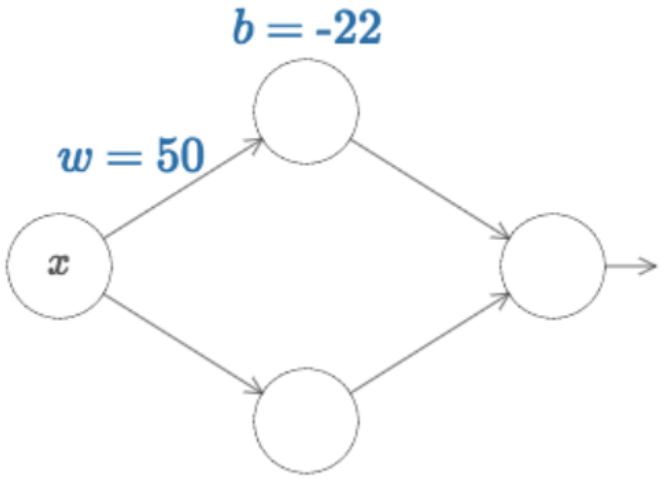
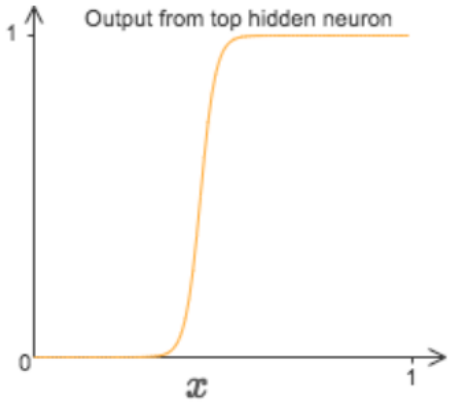
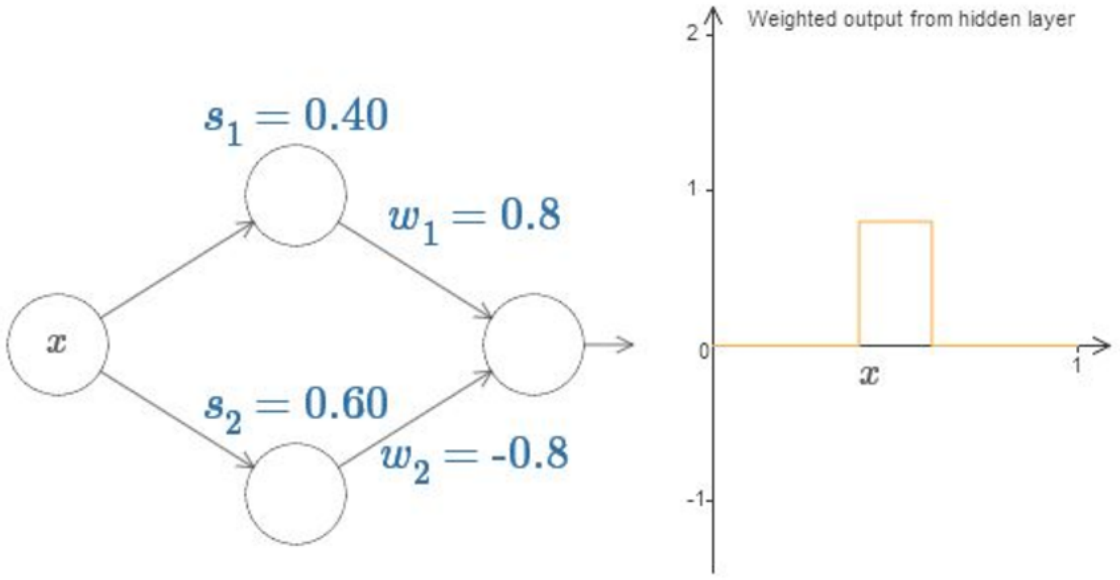
chapter 4: neural networks can compute any function

- like... any function! Almost any process you can imagine can be thought of as function computation. Consider the problem of naming a piece of music based on a short sample of the piece. language translation is a function (granted, not unique), 3-d reconstruction is a function, a movie review is a function (granted, not unique). these can be $\mathbb{R}^m \rightarrow \mathbb{R}^n$ or really any set to any set.
- this universality theorem holds even if we restrict our networks to have just a single layer intermediate between the input and the output neurons!
  - then why do we need deeper nets?
- Of course, just because we know a neural network exists that can (say) translate Chinese text into English, that doesn't mean we have good techniques for constructing or even recognizing such a network.
- A caveat - more precisely, a neural net can approximate a function to any given arbitrary precision.
- The second caveat, other than it being an approximation, is that the class of functions which can be approximated in the way described are the *continuous* functions. However, even if the function we'd really like to compute is discontinuous, it's often the case that a continuous approximation is good enough.
- in this chapter we'll prove a slightly weaker result → that a neural net with two hidden layers (instead of one) can approximate any continuous function to any arbitrary precision.
- note that for the proof that Michael constructs, it is just a simple lookup table. No fancy learning. more like if x is in a certain interval, output this, for a certain other interval, output this etc.
- A visual proof:
  - if we have a really high weight, the sigmoid becomes more like a step function. the point of crossover is when $s = \frac{-b}{w}$. Below is a plot of $y = wx + b$ (x is just a scalar here)

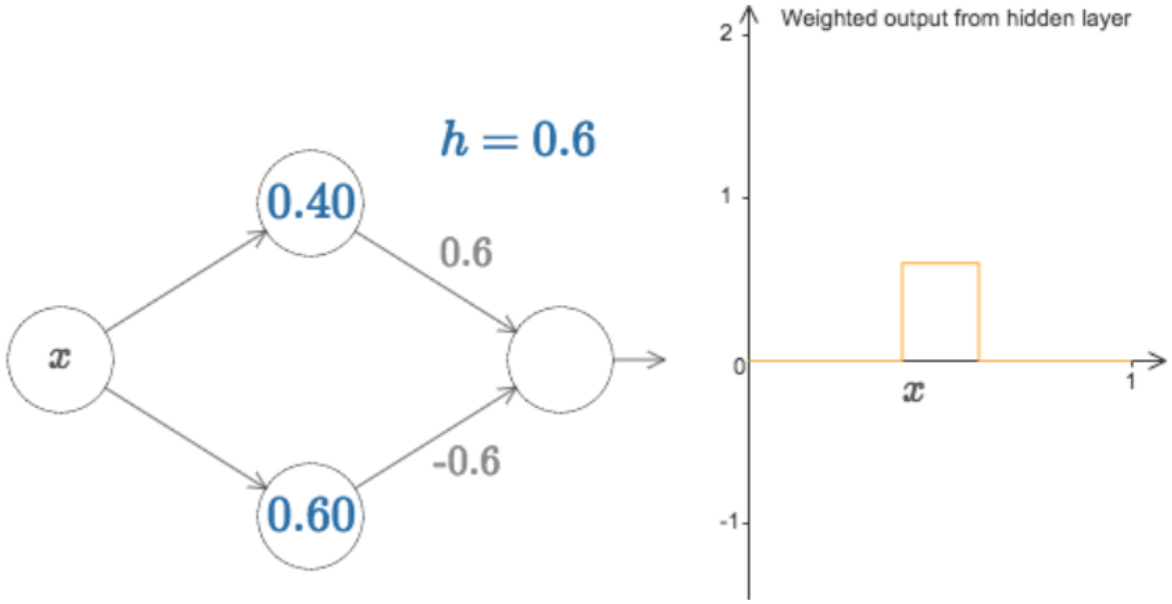




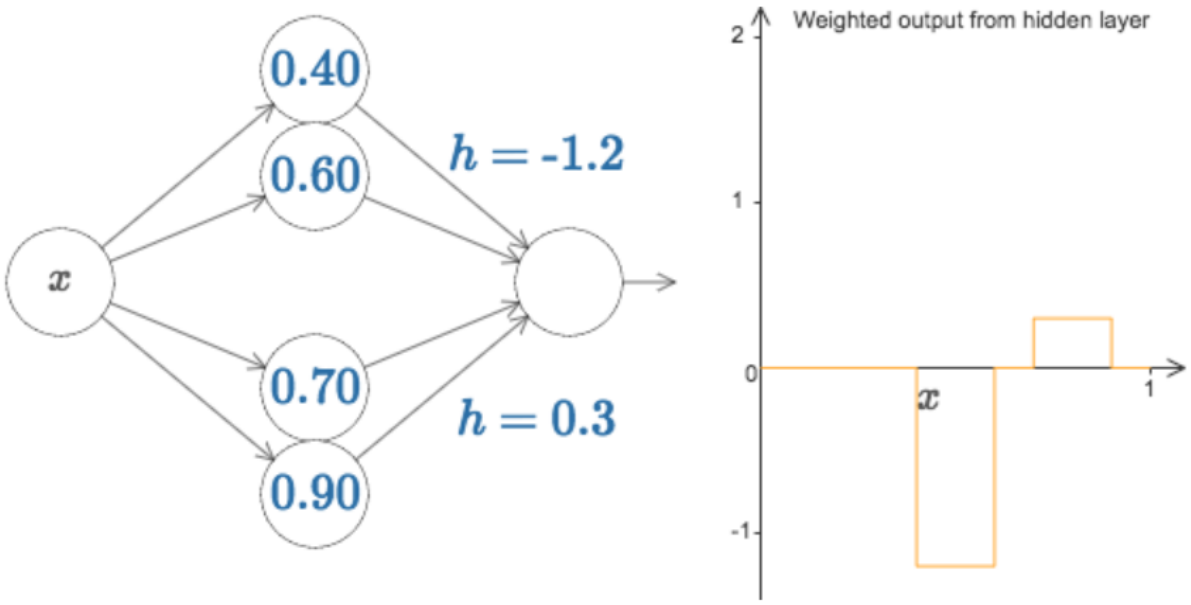
- if we want to achieve one bump of $h = 0.8$, then we setup it up as below with weights h and $-h$.



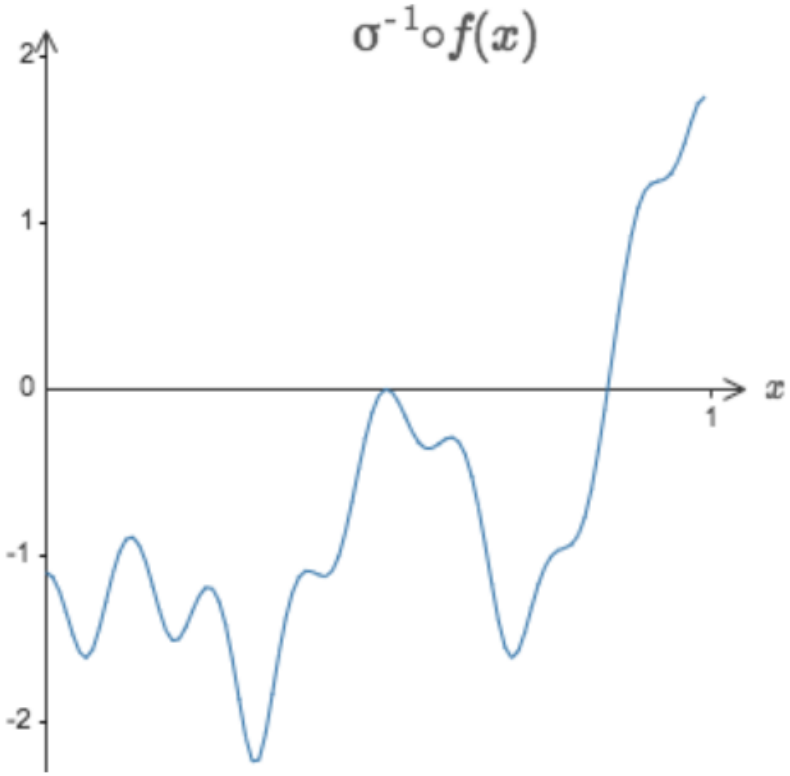
- indicating h more explicitly:



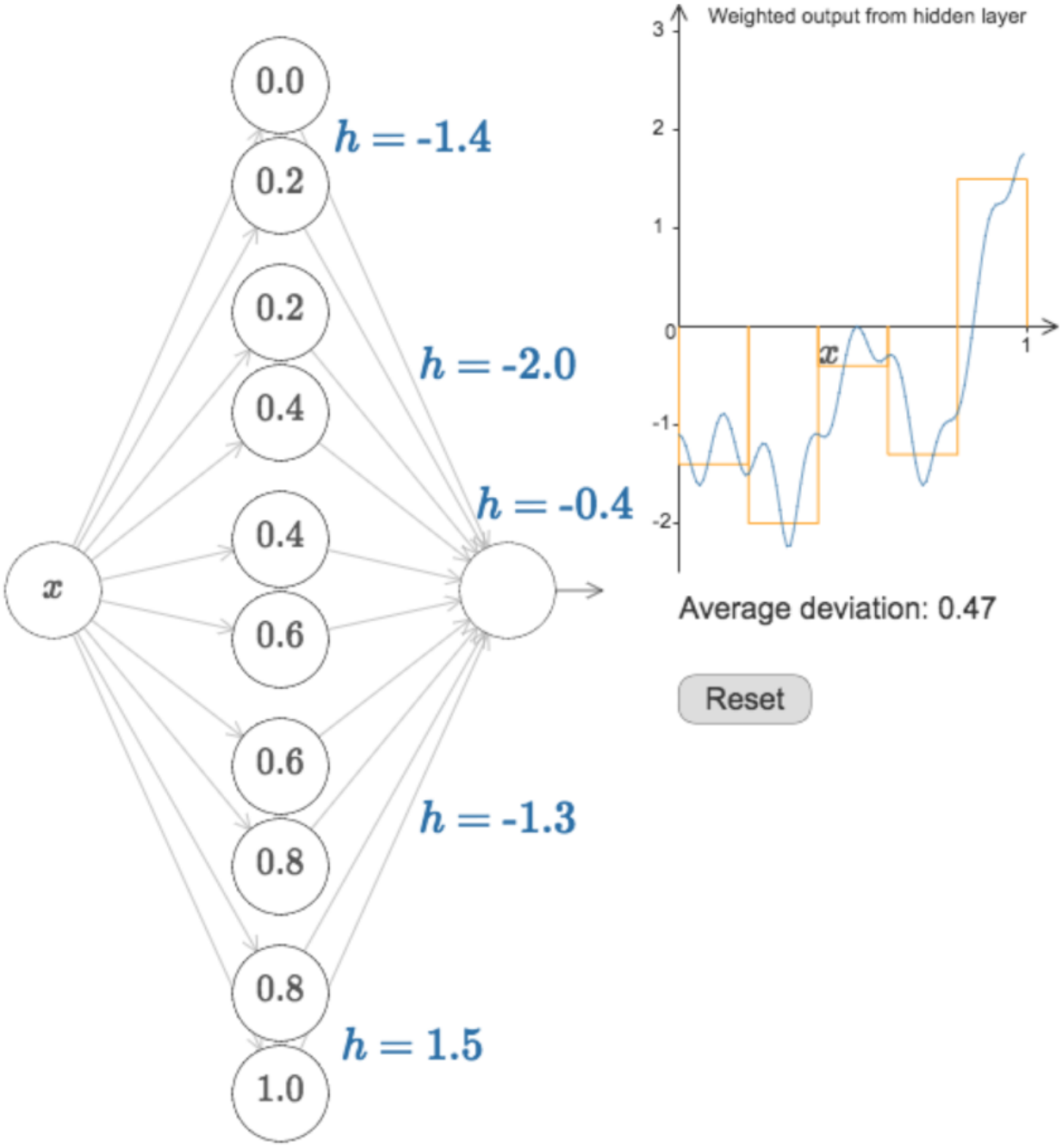
- if we want two bumps, just add a pair of hidden layer neurons.



- if we want the output to be $f(x)$ then the input to the output layer should be $\sigma^{-1}(f(x))$.

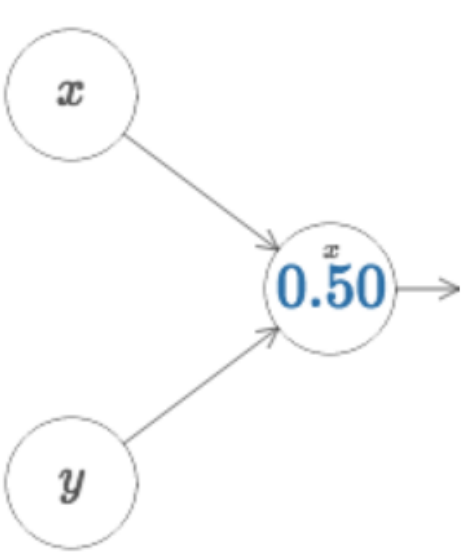


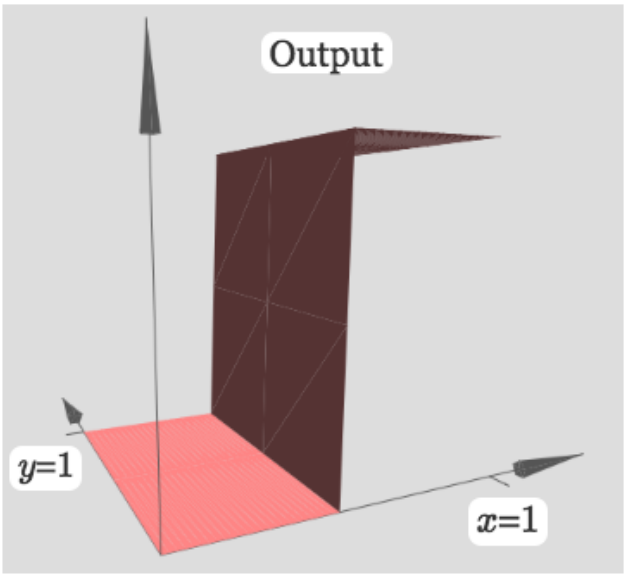
- finally,



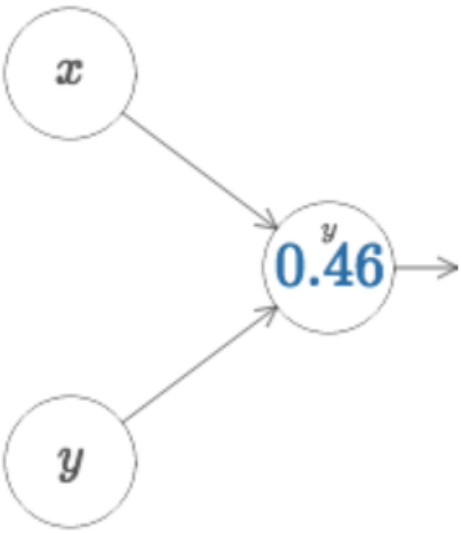
note that the range of the target function should be $[0, 1]$, obviously since that's the range of the output neuron.

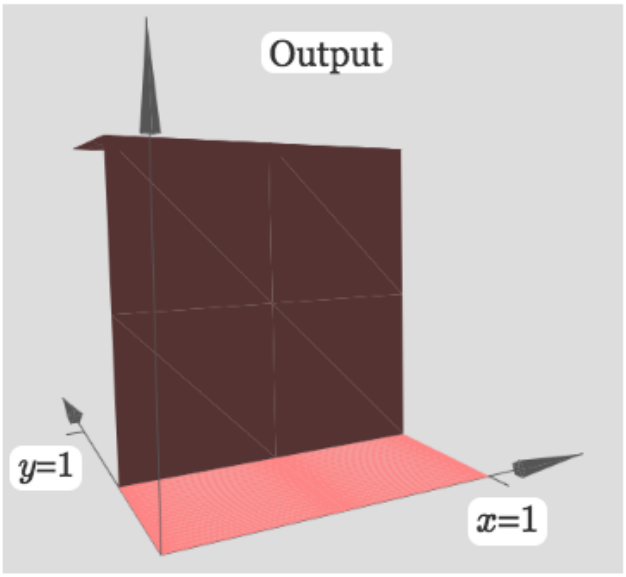
- proof for the multivariate case:
  - again, it is a simple lookup table, with k^d buckets where k is number of intervals in 1 dimension and d is the number of dimensions.
  - we can construct a step function in the x direction by setting the output weight from y to 0.



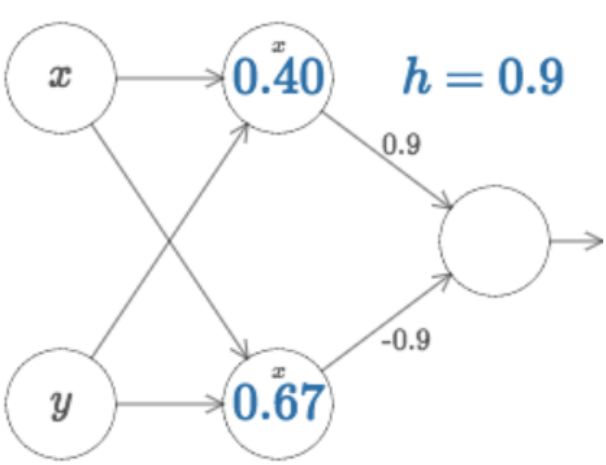


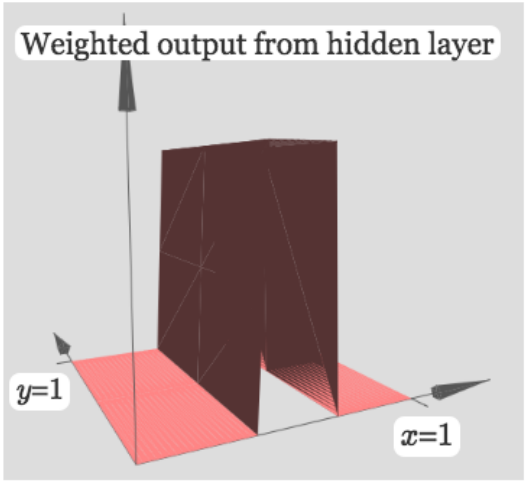
- similarly, in the other direction.



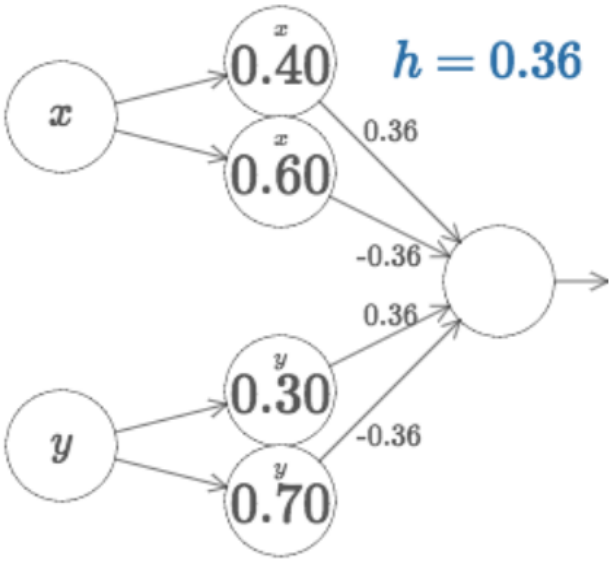


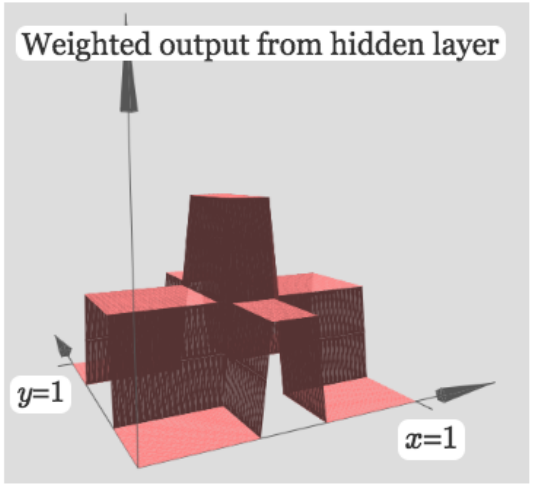
- a way to construct a bump:



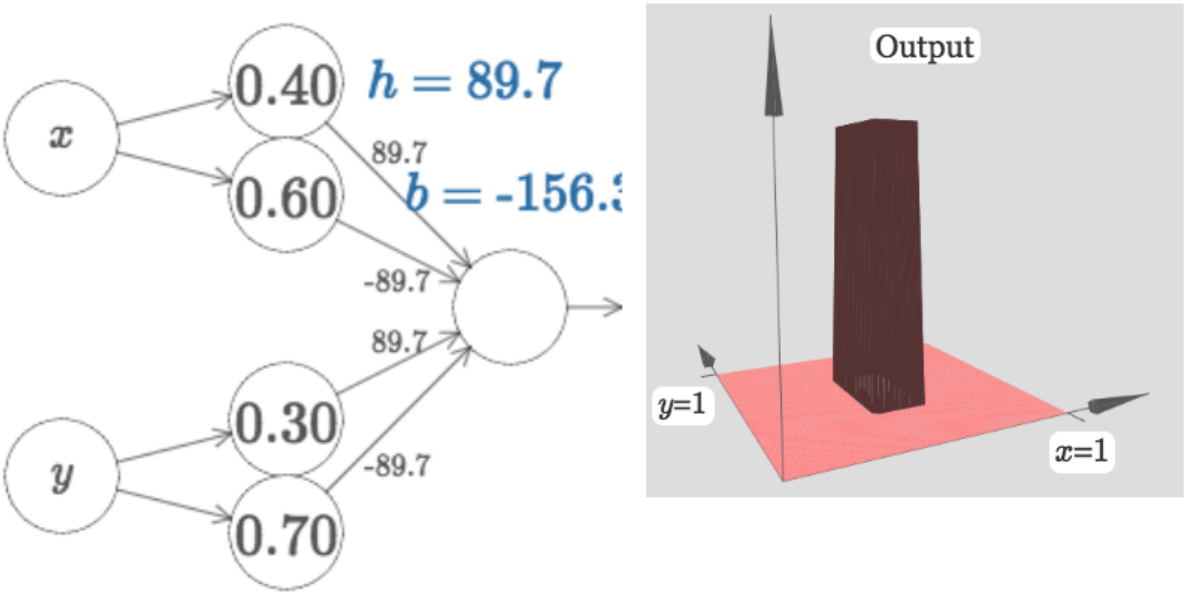


- a way to construct a central tower with plateaus surrounding it:
  (note that here we have taken away the 0 weight connections. also this visualizes the output of the hidden layer, not the output of the final layer)

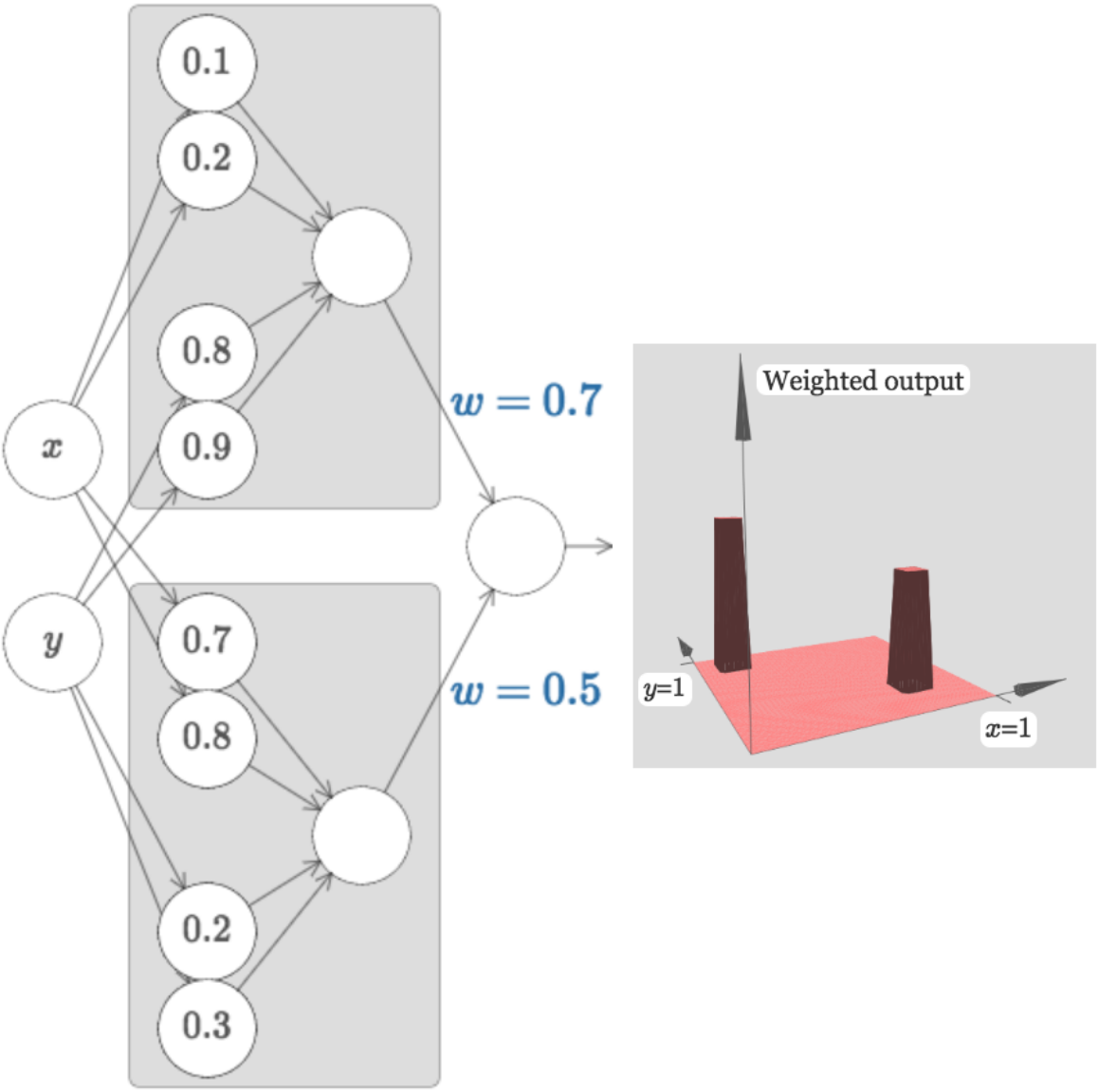




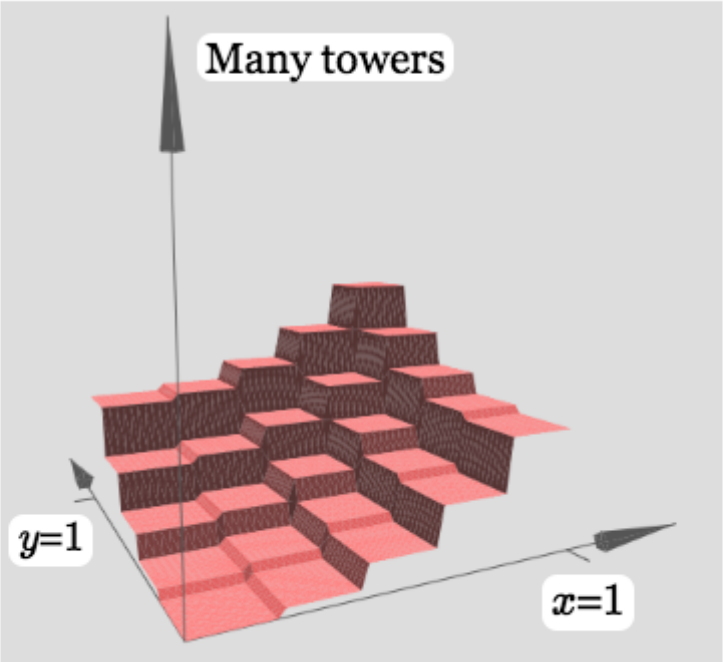
- the final layer weight and bias can be chosen so that the plateaus are suppressed and we can get any desired tower height.



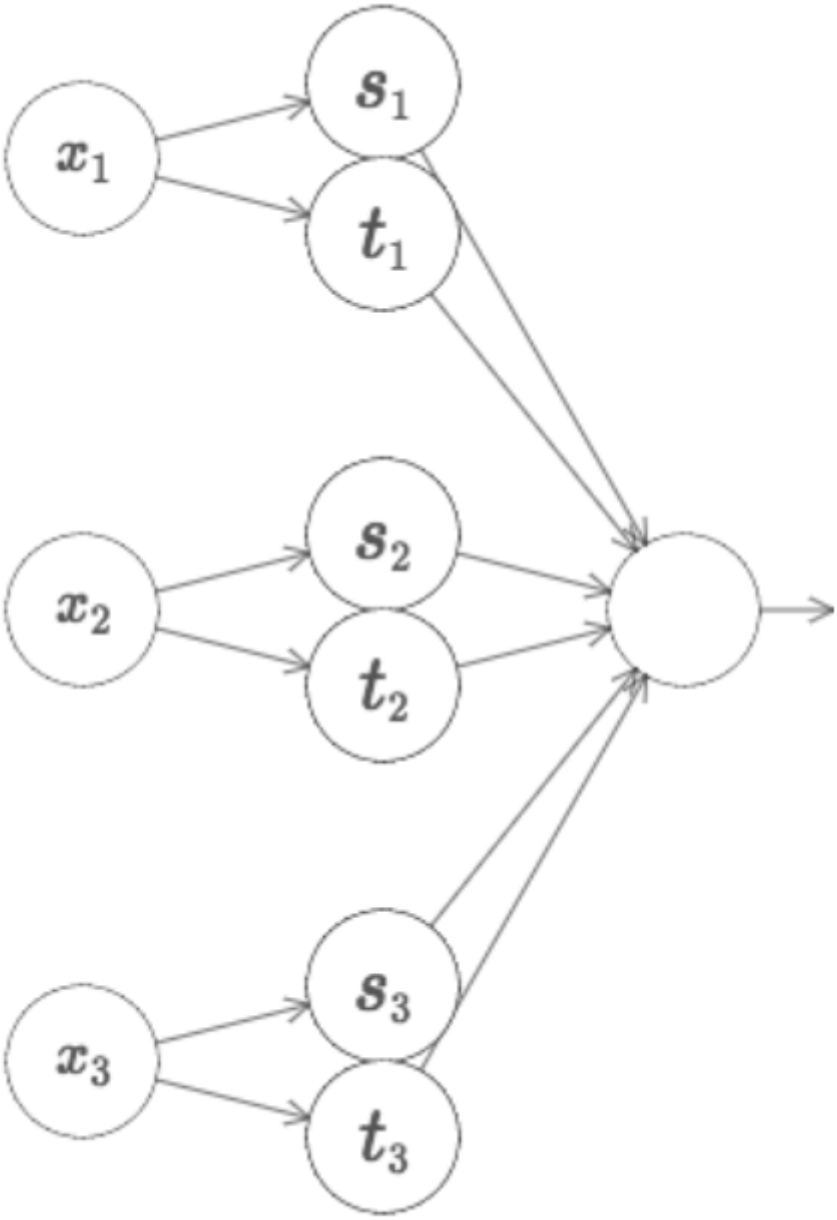
- we can construct many towers thus approximating a function:



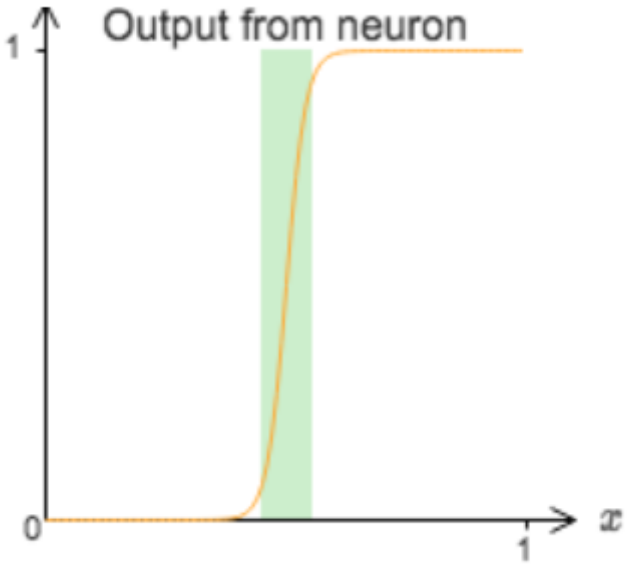
- like this:



- and we can also scale the dimension of inputs by just adding more blocks:



- Technically, the step function here isn't really a step function. things can go bad if the input is in this region:



- we can get around the above issue by constructing M functions that have their bumps slightly moved with each function approximating $f(x)/M$. The output is the sum of these. this will approximate well and get around this weird region problem.
- The explanation for universality we've discussed is certainly not a practical prescription for how to compute using neural networks! The focus is more on showing that it exists. For this reason, I've focused mostly on trying to make the construction clear and easy to follow, and not on optimizing the details of the construction.
- So the right question to ask is not whether any particular function is computable, but rather what's a *good* way to compute the function.
- Most times, like in language translation or image recognition etc, we don't even know what the function to approximate is, so forget using the method we've taken here.
- Given this, you might wonder why we would ever be interested in deep networks, i.e., networks with many hidden layers. Can't we simply replace those networks with shallow, single hidden layer networks? While in principle that's possible, there are good practical reasons to use deep networks. As argued in Chapter 1, deep networks have a hierarchical structure which makes them particularly well adapted to learn the hierarchies of knowledge that seem to be useful in solving real-world problems. Put more concretely, when attacking problems such as image recognition, it helps to use a system that understands not just individual pixels, but also increasingly more complex concepts.