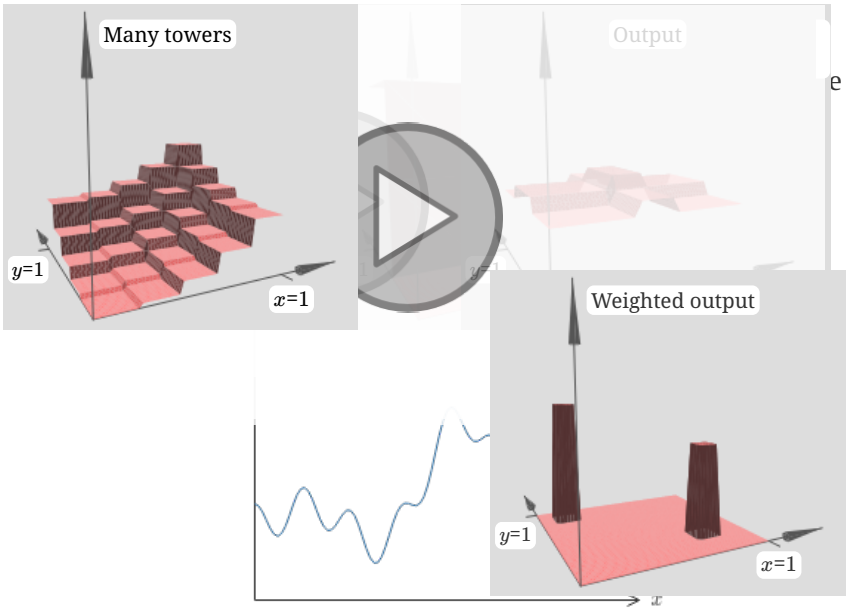
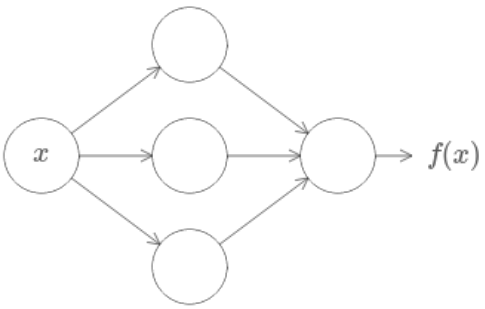
CHAPTER 4

A visual proof that neural nets can compute any function



No matter what the function, there is guaranteed to be a neural network so that for every possible input, x, the value $f(x)$ (or some close approximation) is output from the network, e.g.:



This result holds even if the function has many inputs, $f = f(x_1, \ldots, x_m)$, and many outputs. For instance, here's a network computing a function with $m = 3$ inputs and $n = 2$ outputs:

If you benefit from the book, please make a small donation. I suggest $5, but you can choose the amount.

Donate

Alternately, you can make a donation by sending me Bitcoin, at address

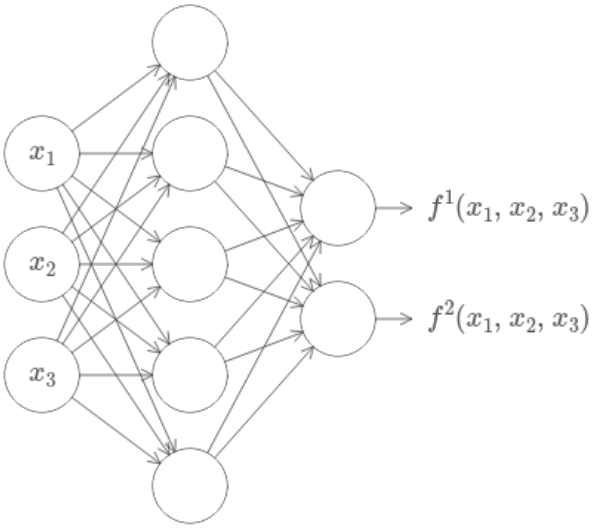
1Kd6tXH5SDAmiFb49J9hknG5pqj7KStSAx

Sponsors

Thanks to all the supporters who made the book possible, with especial thanks to Pavel Dudrenov. Thanks also to all the contributors to the Bugfinder Hall of Fame.

Resources

Michael Nielsen on Twitter



This result tells us that neural networks have a kind of *universality*. No matter what function we want to compute, we know that there is a neural network which can do the job.

What's more, this universality theorem holds even if we restrict our networks to have just a single layer intermediate between the input and the output neurons - a so-called single hidden layer. So even very simple network architectures can be extremely powerful.

The universality theorem is well known by people who use neural networks. But why it's true is not so widely understood. Most of the explanations available are quite technical. For instance, one of the original papers proving the result* did so using the Hahn-Banach theorem, the Riesz Representation theorem, and some Fourier analysis. If you're a mathematician the argument is not difficult to follow, but it's not so easy for most people. That's a pity, since the underlying reasons for universality are simple and beautiful.

In this chapter I give a simple and mostly visual explanation of the universality theorem. We'll go step by step through the underlying ideas. You'll understand why it's true that neural networks can compute any function. You'll understand some of the limitations of the result. And you'll understand how the result relates to deep neural networks.

To follow the material in the chapter, you do not need to have read earlier chapters in this book. Instead, the chapter is structured to be enjoyable as a self-contained essay. Provided you have just a little basic familiarity with neural networks, you should be able to follow the explanation. I will, however, provide occasional links to earlier material, to help fill in any gaps in your knowledge.

Book FAQ

Code repository

Michael Nielsen's project announcement mailing list

Deep Learning, book by Ian Goodfellow, Yoshua Bengio, and Aaron Courville

cognitivemedium.com

By Michael Nielsen / Dec 2019

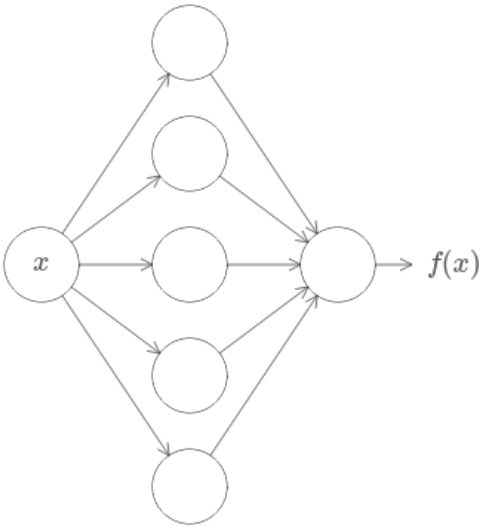
Universality theorems are a commonplace in computer science, so much so that we sometimes forget how astonishing they are. But it's worth reminding ourselves: the ability to compute an arbitrary function is truly remarkable. Almost any process you can imagine can be thought of as function computation. Consider the problem of naming a piece of music based on a short sample of the piece. That can be thought of as computing a function. Or consider the problem of translating a Chinese text into English. Again, that can be thought of as computing a function*. Or consider the problem of taking an mp4 movie file and generating a description of the plot of the movie, and a discussion of the quality of the acting. Again, that can be thought of as a kind of function computation*. Universality means that, in principle, neural networks can do all these things and many more.

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. This limitation applies also to traditional universality theorems for models such as Boolean circuits. But, as we've seen earlier in the book, neural networks have powerful algorithms for learning functions. That combination of learning algorithms + universality is an attractive mix. Up to now, the book has focused on the learning algorithms. In this chapter, we focus on universality, and what it means.

Two caveats

Before explaining why the universality theorem is true, I want to mention two caveats to the informal statement "a neural network can compute any function".

First, this doesn't mean that a network can be used to *exactly* compute any function. Rather, we can get an *approximation* that is as good as we want. By increasing the number of hidden neurons we can improve the approximation. For instance, earlier I illustrated a network computing some function $f(x)$ using three hidden neurons. For most functions only a low-quality approximation will be possible using three hidden neurons. By increasing the number of hidden neurons (say, to five) we can typically get a better approximation:



And we can do still better by further increasing the number of hidden neurons.

To make this statement more precise, suppose we're given a function $f(x)$ which we'd like to compute to within some desired accuracy $\epsilon > 0$. The guarantee is that by using enough hidden neurons we can always find a neural network whose output $g(x)$ satisfies $|g(x) - f(x)| < \epsilon$, for all inputs x. In other words, the approximation will be good to within the desired accuracy for every possible input.

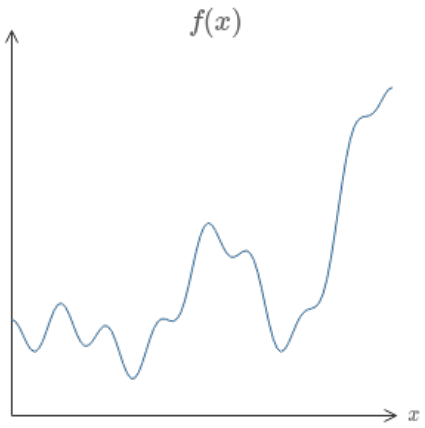
The second caveat is that the class of functions which can be approximated in the way described are the *continuous* functions. If a function is discontinuous, i.e., makes sudden, sharp jumps, then it won't in general be possible to approximate using a neural net. This is not surprising, since our neural networks compute continuous functions of their input. 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. If that's so, then we can use a neural network. In practice, this is not usually an important limitation.

Summing up, a more precise statement of the universality theorem is that neural networks with a single hidden layer can be used to approximate any continuous function to any desired precision. In this chapter we'll actually prove a slightly weaker version of this result, using two hidden layers instead of one. In the problems I'll briefly outline how the explanation can, with a few tweaks, be adapted to give a proof which uses only a single hidden layer.

Universality with one input and one

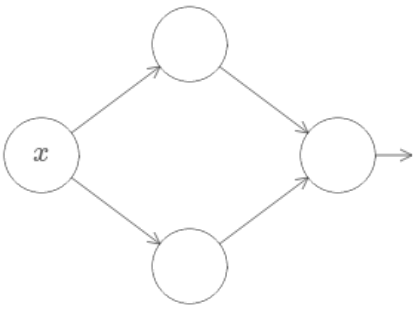
output

To understand why the universality theorem is true, let's start by understanding how to construct a neural network which approximates a function with just one input and one output:

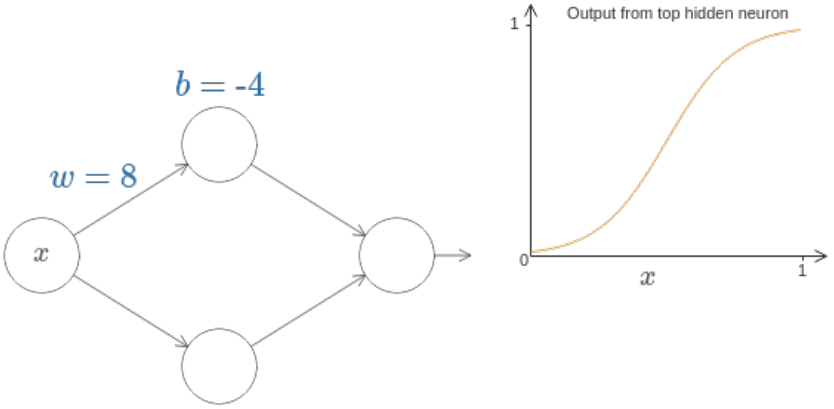


It turns out that this is the core of the problem of universality. Once we've understood this special case it's actually pretty easy to extend to functions with many inputs and many outputs.

To build insight into how to construct a network to compute *f*, let's start with a network containing just a single hidden layer, with two hidden neurons, and an output layer containing a single output neuron:



To get a feel for how components in the network work, let's focus on the top hidden neuron. In the diagram below, click on the weight, *w*, and drag the mouse a little ways to the right to increase *w*. You can immediately see how the function computed by the top hidden neuron changes:



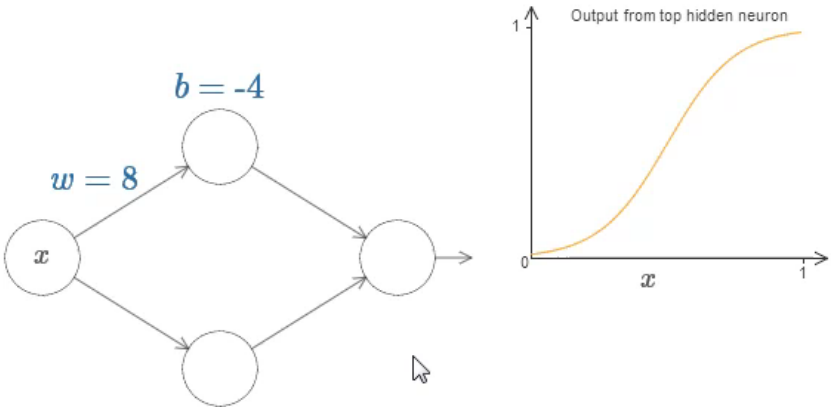
As we learnt earlier in the book, what's being computed by the hidden neuron is $\sigma(wx + b)$, where $\sigma(z) \equiv 1/(1 + e^{-z})$ is the sigmoid function. Up to now, we've made frequent use of this algebraic form. But for the proof of universality we will obtain more insight by ignoring the algebra entirely, and instead manipulating and observing the shape shown in the graph. This won't just give us a better feel for what's going on, it will also give us a proof* of universality that applies to activation functions other than the sigmoid function.

To get started on this proof, try clicking on the bias, b, in the diagram above, and dragging to the right to increase it. You'll see that as the bias increases the graph moves to the left, but its shape doesn't change.

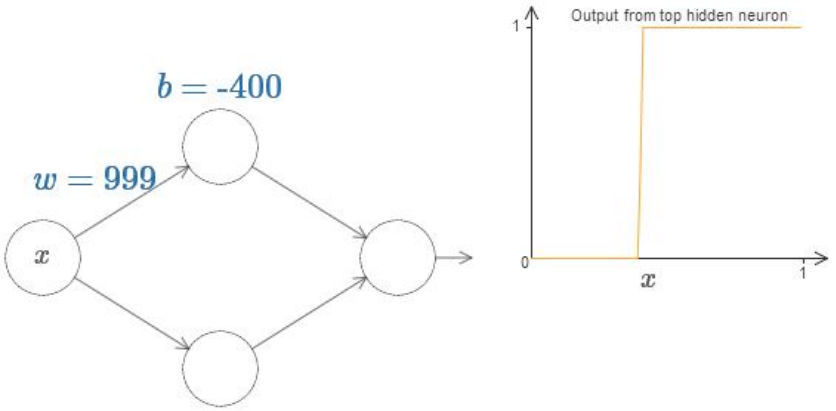
Next, click and drag to the left in order to decrease the bias. You'll see that as the bias decreases the graph moves to the right, but, again, its shape doesn't change.

Next, decrease the weight to around 2 or 3. You'll see that as you decrease the weight, the curve broadens out. You might need to change the bias as well, in order to keep the curve in-frame.

Finally, increase the weight up past $w = 100$. As you do, the curve gets steeper, until eventually it begins to look like a step function. Try to adjust the bias so the step occurs near $x = 0.3$. The following short clip shows what your result should look like. Click on the play button to play (or replay) the video:



We can simplify our analysis quite a bit by increasing the weight so much that the output really is a step function, to a very good approximation. Below I've plotted the output from the top hidden neuron when the weight is $w = 999$. Note that this plot is static, and you can't change parameters such as the weight.



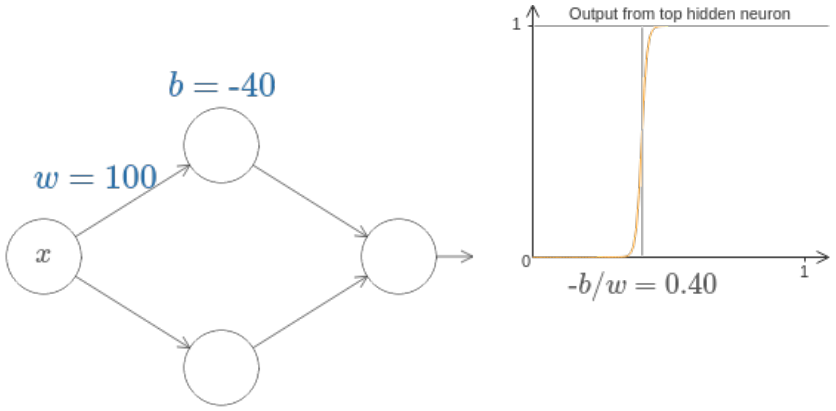
It's actually quite a bit easier to work with step functions than general sigmoid functions. The reason is that in the output layer we add up contributions from all the hidden neurons. It's easy to analyze the sum of a bunch of step functions, but rather more difficult to reason about what happens when you add up a bunch of sigmoid shaped curves. And so it makes things much easier to assume that our hidden neurons are outputting step functions. More concretely, we do this by fixing the weight w to be some very large value, and then setting the position of the step by modifying the bias. Of course, treating the output as a step function is an approximation, but it's a very good approximation, and for now we'll treat it as exact. I'll come back later to discuss the impact of deviations from this approximation.

At what value of x does the step occur? Put another way, how does the position of the step depend upon the weight and bias?

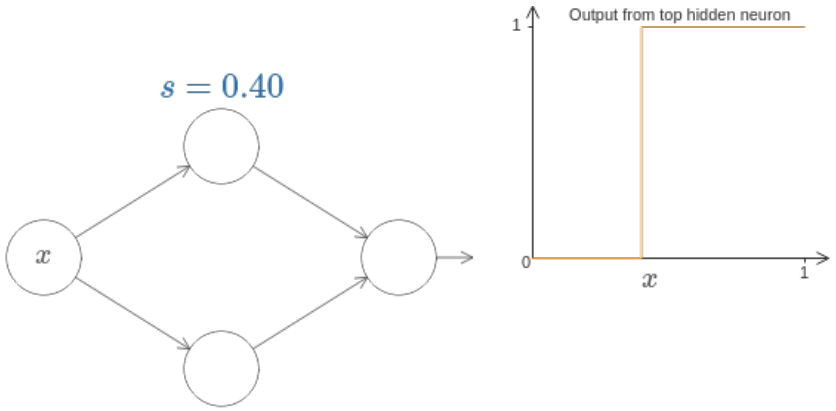
To answer this question, try modifying the weight and bias in the diagram

above (you may need to scroll back a bit). Can you figure out how the position of the step depends on w and b? With a little work you should be able to convince yourself that the position of the step is *proportional* to b, and *inversely proportional* to w.

In fact, the step is at position $s = -b/w$, as you can see by modifying the weight and bias in the following diagram:

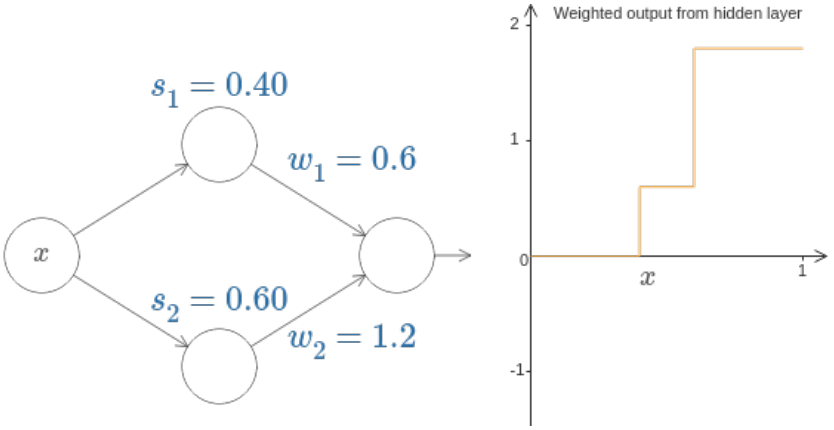


It will greatly simplify our lives to describe hidden neurons using just a single parameter, s, which is the step position, $s = -b/w$. Try modifying s in the following diagram, in order to get used to the new parameterization:



As noted above, we've implicitly set the weight w on the input to be some large value - big enough that the step function is a very good approximation. We can easily convert a neuron parameterized in this way back into the conventional model, by choosing the bias $b = -ws$.

Up to now we've been focusing on the output from just the top hidden neuron. Let's take a look at the behavior of the entire network. In particular, we'll suppose the hidden neurons are computing step functions parameterized by step points s_1 (top neuron) and s_2 (bottom neuron). And they'll have respective output weights w_1 and w_2. Here's the network:



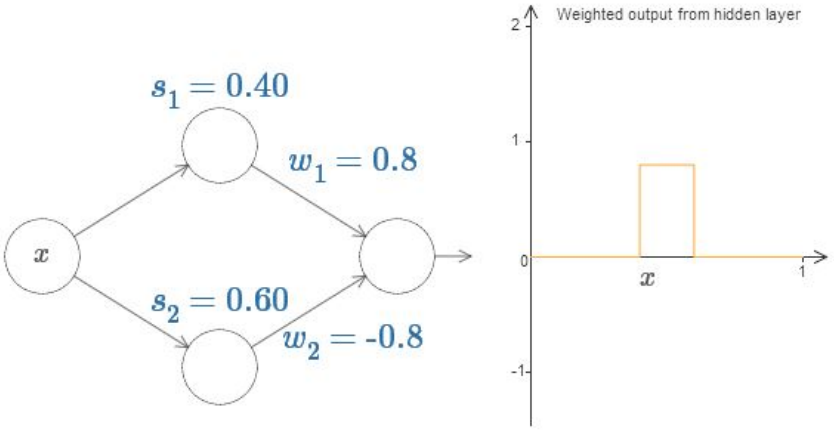
What's being plotted on the right is the *weighted output* $w_1 a_1 + w_2 a_2$ from the hidden layer. Here, a_1 and a_2 are the outputs from the top and bottom hidden neurons, respectively*. These outputs are denoted with *a*s because they're often known as the neurons' *activations*.

Try increasing and decreasing the step point s_1 of the top hidden neuron. Get a feel for how this changes the weighted output from the hidden layer. It's particularly worth understanding what happens when s_1 goes past s_2. You'll see that the graph changes shape when this happens, since we have moved from a situation where the top hidden neuron is the first to be activated to a situation where the bottom hidden neuron is the first to be activated.

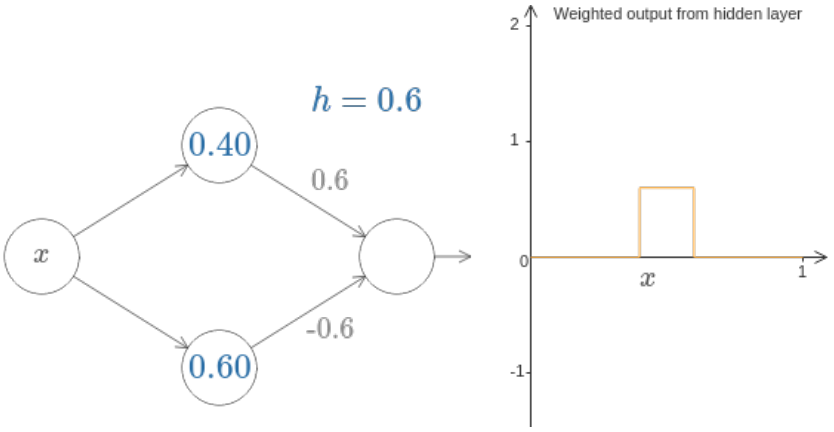
Similarly, try manipulating the step point s_2 of the bottom hidden neuron, and get a feel for how this changes the combined output from the hidden neurons.

Try increasing and decreasing each of the output weights. Notice how this rescales the contribution from the respective hidden neurons. What happens when one of the weights is zero?

Finally, try setting w_1 to be 0.8 and w_2 to be -0.8. You get a "bump" function, which starts at point s_1, ends at point s_2, and has height 0.8. For instance, the weighted output might look like this:



Of course, we can rescale the bump to have any height at all. Let's use a single parameter, *h*, to denote the height. To reduce clutter I'll also remove the "s_1 = …" and "w_1 = …" notations.



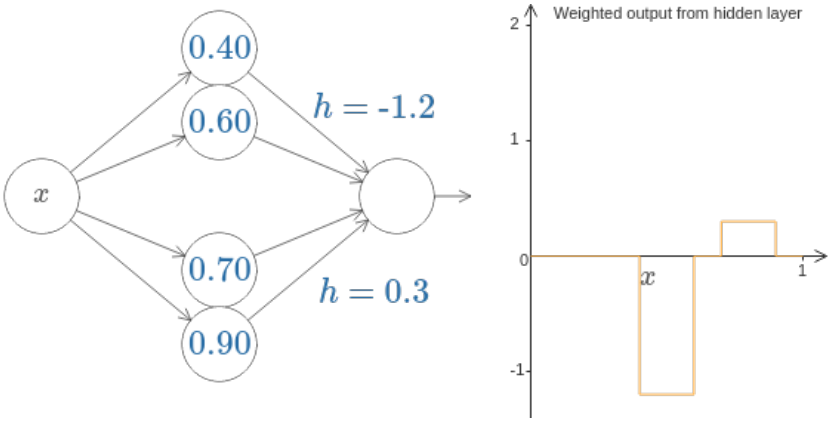
Try changing the value of *h* up and down, to see how the height of the bump changes. Try changing the height so it's negative, and observe what happens. And try changing the step points to see how that changes the shape of the bump.

You'll notice, by the way, that we're using our neurons in a way that can be thought of not just in graphical terms, but in more conventional programming terms, as a kind of `if-then-else` statement, e.g.:

```
if input >= step point:
    add 1 to the weighted output
else:
    add 0 to the weighted output
```

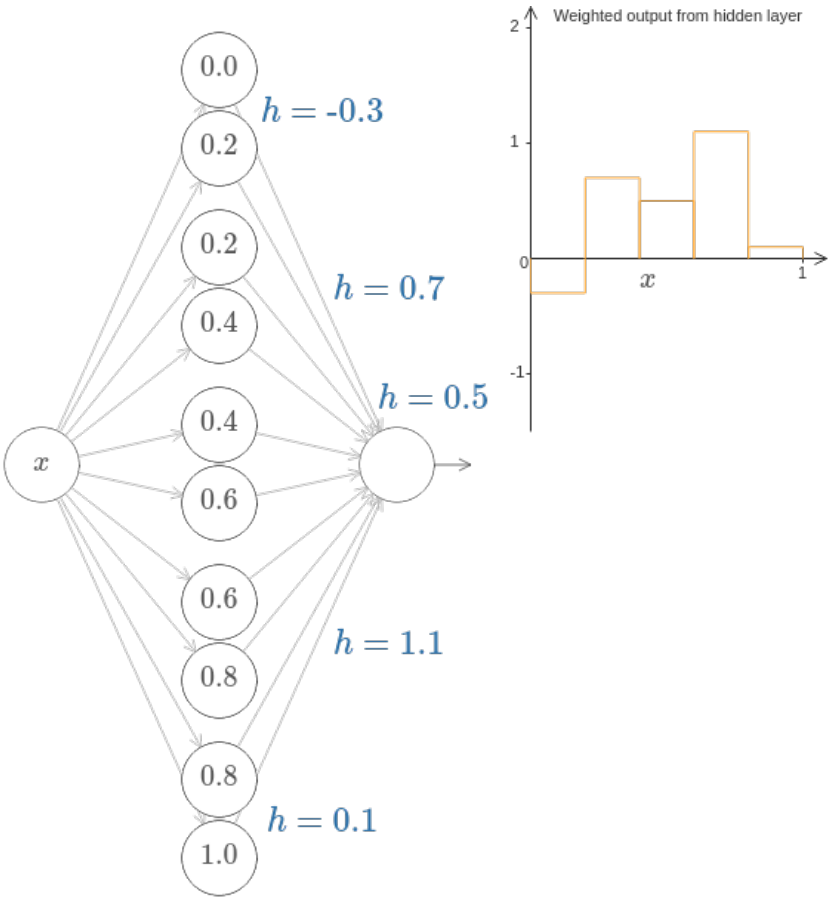
For the most part I'm going to stick with the graphical point of view. But in what follows you may sometimes find it helpful to switch points of view, and think about things in terms of `if-then-else`.

We can use our bump-making trick to get two bumps, by gluing two pairs of hidden neurons together into the same network:



I've suppressed the weights here, simply writing the *h* values for each pair of hidden neurons. Try increasing and decreasing both *h* values, and observe how it changes the graph. Move the bumps around by changing the step points.

More generally, we can use this idea to get as many peaks as we want, of any height. In particular, we can divide the interval [0, 1] up into a large number, *N*, of subintervals, and use *N* pairs of hidden neurons to set up peaks of any desired height. Let's see how this works for $N = 5$. That's quite a few neurons, so I'm going to pack things in a bit. Apologies for the complexity of the diagram: I could hide the complexity by abstracting away further, but I think it's worth putting up with a little complexity, for the sake of getting a more concrete feel for how these networks work.



You can see that there are five pairs of hidden neurons. The step points for the respective pairs of neurons are 0, $1/5$, then $1/5$, $2/5$, and so on, out to $4/5$, $5/5$. These values are fixed - they make it so we get five evenly spaced bumps on the graph.

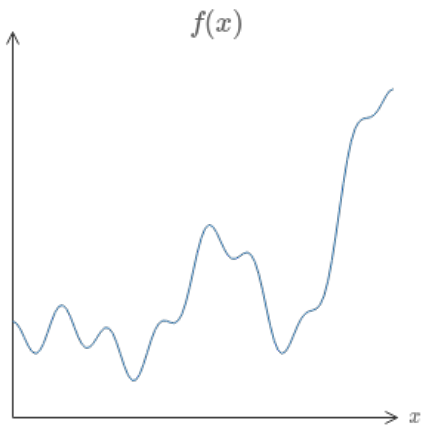
Each pair of neurons has a value of h associated to it. Remember, the connections output from the neurons have weights h and $-h$ (not marked). Click on one of the h values, and drag the mouse to the right or left to change the value. As you do so, watch the function change. By changing the output weights we're actually *designing* the function!

Contrariwise, try clicking on the graph, and dragging up or down to change the height of any of the bump functions. As you change the heights, you can see the corresponding change in h values. And, although it's not shown, there is also a change in the corresponding output weights, which are $+h$ and $-h$.

In other words, we can directly manipulate the function appearing in the graph on the right, and see that reflected in the h values on the left. A fun thing to do is to hold the mouse button down and drag the mouse from one side of the graph to the other. As you do this you draw out a function, and get to watch the parameters in the neural network adapt.

Time for a challenge.

Let's think back to the function I plotted at the beginning of the chapter:



I didn't say it at the time, but what I plotted is actually the function

$$f(x) = 0.2 + 0.4x^2 + 0.3x\sin(15x) + 0.05\cos(50x),$$

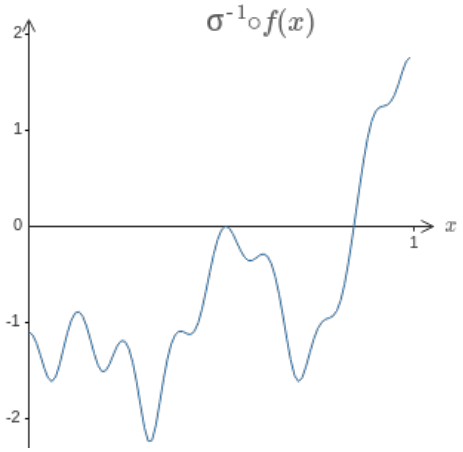
plotted over x from 0 to 1, and with the y axis taking values from 0 to 1.

That's obviously not a trivial function.

You're going to figure out how to compute it using a neural network.

In our networks above we've been analyzing the weighted combination $\sum_j w_j a_j$ output from the hidden neurons. We now know how to get a lot of control over this quantity. But, as I noted earlier, this quantity is not what's output from the network. What's output from the network is $\sigma(\sum_j w_j a_j + b)$ where b is the bias on the output neuron. Is there some way we can achieve control over the actual output from the network?

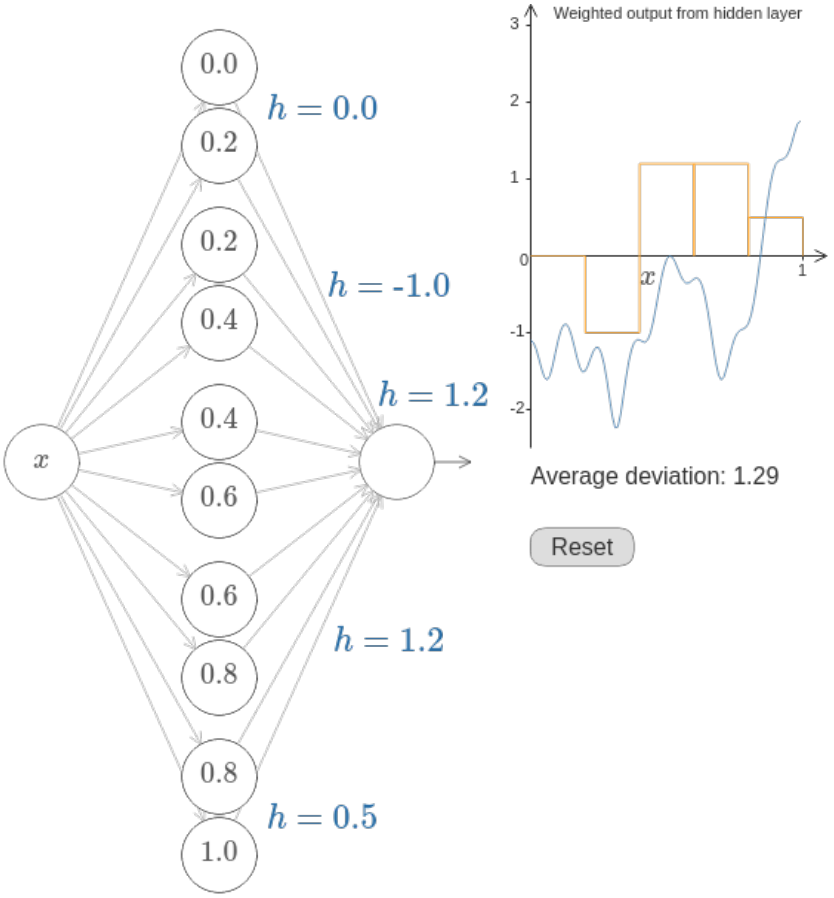
The solution is to design a neural network whose hidden layer has a weighted output given by $\sigma^{-1} \circ f(x)$, where σ^{-1} is just the inverse of the σ function. That is, we want the weighted output from the hidden layer to be:



If we can do this, then the output from the network as a whole will be a good approximation to $f(x)$*.

Your challenge, then, is to design a neural network to approximate the goal function shown just above. To learn as much as possible, I want you to solve the problem twice. The first time, please click on the graph, directly adjusting the heights of the different bump functions. You should find it fairly easy to get a good match to the goal function. How well you're doing is measured by the *average deviation* between the goal function and the function the network is actually computing. Your challenge is to drive the average deviation as *low* as possible. You complete the challenge when you drive the average deviation to 0.40 or below.

Once you've done that, click on "Reset" to randomly re-initialize the bumps. The second time you solve the problem, resist the urge to click on the graph. Instead, modify the h values on the left-hand side, and again attempt to drive the average deviation to 0.40 or below.



You've now figured out all the elements necessary for the network to approximately compute the function $f(x)$! It's only a coarse approximation, but we could easily do much better, merely by increasing the number of pairs of hidden neurons, allowing more bumps.

In particular, it's easy to convert all the data we have found back into the standard parameterization used for neural networks. Let me just recap quickly how that works.

The first layer of weights all have some large, constant value, say $w = 1000$.

The biases on the hidden neurons are just $b = -ws$. So, for instance, for the second hidden neuron $s = 0.2$ becomes $b = -1000 \times 0.2 = -200$.

The final layer of weights are determined by the h values. So, for instance, the value you've chosen above for the first h, $h = 0.0$, means that the output weights from the top two hidden neurons are 0.0 and 0.0, respectively. And so on, for the entire layer of output weights.

Finally, the bias on the output neuron is 0.

That's everything: we now have a complete description of a neural network which does a pretty good job computing our original goal

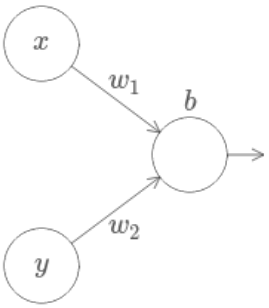
function. And we understand how to improve the quality of the approximation by improving the number of hidden neurons.

What's more, there was nothing special about our original goal function, $f(x) = 0.2 + 0.4x^2 + 0.3\sin(15x) + 0.05\cos(50x)$. We could have used this procedure for any continuous function from [0, 1] to [0, 1]. In essence, we're using our single-layer neural networks to build a lookup table for the function. And we'll be able to build on this idea to provide a general proof of universality.

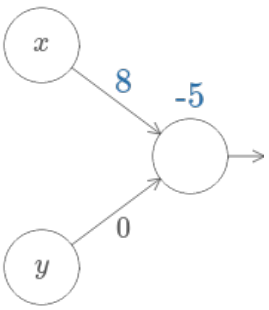
Many input variables

Let's extend our results to the case of many input variables. This sounds complicated, but all the ideas we need can be understood in the case of just two inputs. So let's address the two-input case.

We'll start by considering what happens when we have two inputs to a neuron:



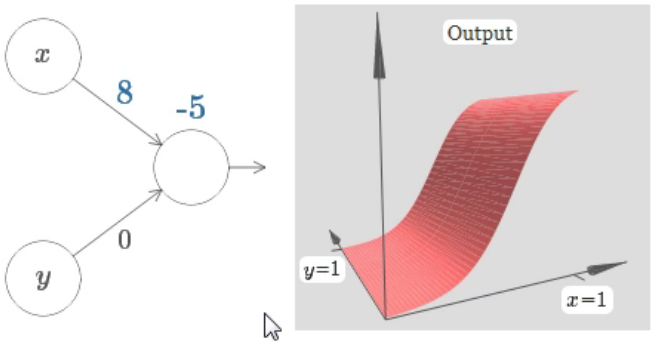
Here, we have inputs x and y, with corresponding weights w_1 and w_2, and a bias b on the neuron. Let's set the weight w_2 to 0, and then play around with the first weight, w_1, and the bias, b, to see how they affect the output from the neuron:



As you can see, with $w_2 = 0$ the input y makes no difference to the output from the neuron. It's as though x is the only input.

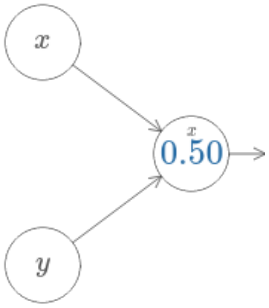
Given this, what do you think happens when we increase the weight w_1 to

$w_1 = 100$, with w_2 remaining 0? If you don't immediately see the answer, ponder the question for a bit, and see if you can figure out what happens. Then try it out and see if you're right. I've shown what happens in the following movie:

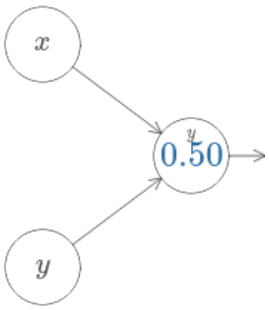


Just as in our earlier discussion, as the input weight gets larger the output approaches a step function. The difference is that now the step function is in three dimensions. Also as before, we can move the location of the step point around by modifying the bias. The actual location of the step point is $s_x \equiv -b/w_1$.

Let's redo the above using the position of the step as the parameter:

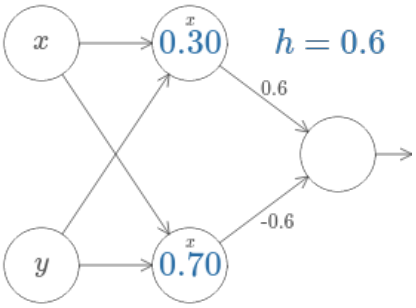


Here, we assume the weight on the x input has some large value - I've used $w_1 = 1000$ - and the weight $w_2 = 0$. The number on the neuron is the step point, and the little x above the number reminds us that the step is in the x direction. Of course, it's also possible to get a step function in the y direction, by making the weight on the y input very large (say, $w_2 = 1000$), and the weight on the x equal to 0, i.e., $w_1 = 0$:



The number on the neuron is again the step point, and in this case the little *y* above the number reminds us that the step is in the *y* direction. I could have explicitly marked the weights on the *x* and *y* inputs, but decided not to, since it would make the diagram rather cluttered. But do keep in mind that the little *y* marker implicitly tells us that the *y* weight is large, and the *x* weight is 0.

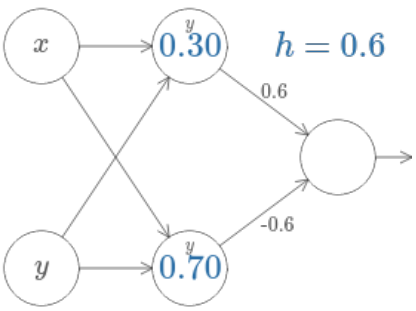
We can use the step functions we've just constructed to compute a three-dimensional bump function. To do this, we use two neurons, each computing a step function in the *x* direction. Then we combine those step functions with weight *h* and $-h$, respectively, where *h* is the desired height of the bump. It's all illustrated in the following diagram:



Try changing the value of the height, *h*. Observe how it relates to the weights in the network. And see how it changes the height of the bump function on the right.

Also, try changing the step point 0.30 associated to the top hidden neuron. Witness how it changes the shape of the bump. What happens when you move it past the step point 0.70 associated to the bottom hidden neuron?

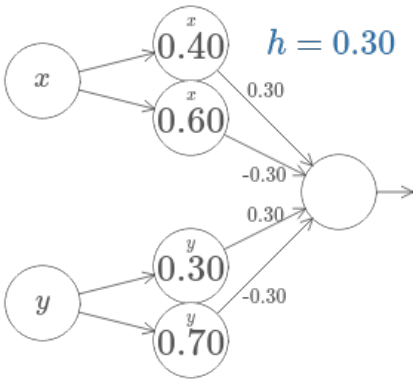
We've figured out how to make a bump function in the *x* direction. Of course, we can easily make a bump function in the *y* direction, by using two step functions in the *y* direction. Recall that we do this by making the weight large on the *y* input, and the weight 0 on the *x* input. Here's the result:



This looks nearly identical to the earlier network! The only thing explicitly

shown as changing is that there's now little *y* markers on our hidden neurons. That reminds us that they're producing *y* step functions, not *x* step functions, and so the weight is very large on the *y* input, and zero on the *x* input, not vice versa. As before, I decided not to show this explicitly, in order to avoid clutter.

Let's consider what happens when we add up two bump functions, one in the *x* direction, the other in the *y* direction, both of height *h*:



To simplify the diagram I've dropped the connections with zero weight. For now, I've left in the little *x* and *y* markers on the hidden neurons, to remind you in what directions the bump functions are being computed. We'll drop even those markers later, since they're implied by the input variable.

Try varying the parameter *h*. As you can see, this causes the output weights to change, and also the heights of both the *x* and *y* bump functions.

What we've built looks a little like a *tower* function:

If we could build such tower functions, then we could use them to approximate arbitrary functions, just by adding up many towers of different heights, and in different locations:

Of course, we haven't yet figured out how to build a tower function. What we have constructed looks like a central tower, of height $2h$, with a surrounding plateau, of height h.

But we can make a tower function. Remember that earlier we saw neurons can be used to implement a type of `if-then-else` statement:

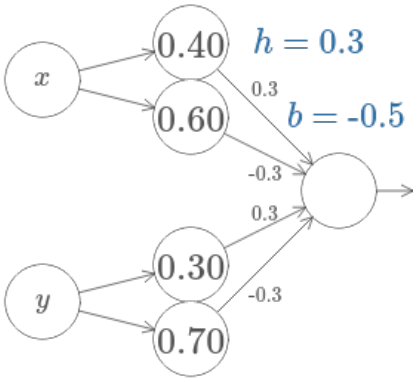
```
if input >= threshold:
    output 1
else:
    output 0
```

That was for a neuron with just a single input. What we want is to apply a similar idea to the combined output from the hidden neurons:

```
if combined output from hidden neurons >= threshold:
    output 1
else:
    output 0
```

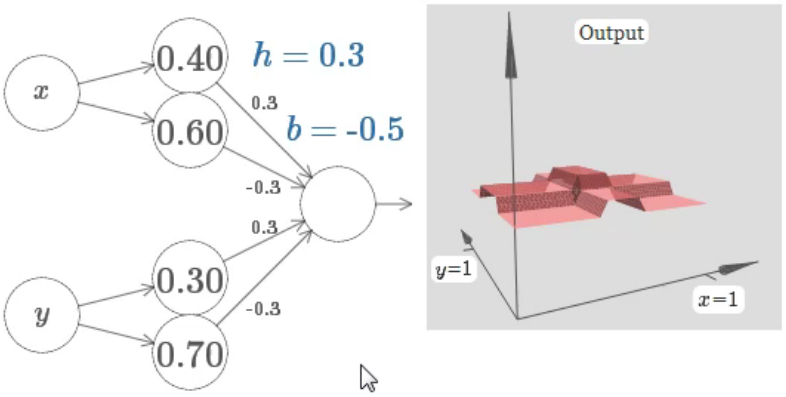
If we choose the `threshold` appropriately - say, a value of $3h/2$, which is sandwiched between the height of the plateau and the height of the central tower - we could squash the plateau down to zero, and leave just the tower standing.

Can you see how to do this? Try experimenting with the following network to figure it out. Note that we're now plotting the output from the entire network, not just the weighted output from the hidden layer. This means we add a bias term to the weighted output from the hidden layer, and apply the sigma function. Can you find values for h and b which produce a tower? This is a bit tricky, so if you think about this for a while and remain stuck, here's two hints: (1) To get the output neuron to show the right kind of `if-then-else` behaviour, we need the input weights (all h or $-h$) to be large; and (2) the value of b determines the scale of the `if-then-else` threshold.



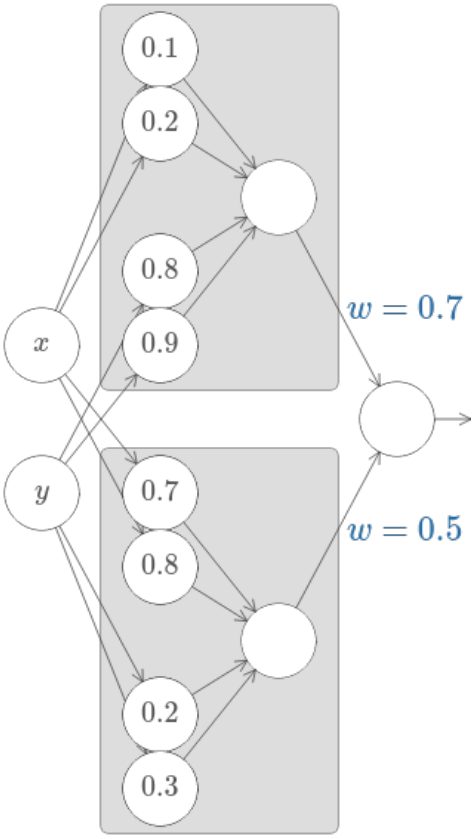
With our initial parameters, the output looks like a flattened version of the earlier diagram, with its tower and plateau. To get the desired behaviour, we increase the parameter *h* until it becomes large. That gives the `if-then-else` thresholding behaviour. Second, to get the threshold right, we'll choose $b \approx -3h/2$. Try it, and see how it works!

Here's what it looks like, when we use $h = 10$:



Even for this relatively modest value of *h*, we get a pretty good tower function. And, of course, we can make it as good as we want by increasing *h* still further, and keeping the bias as $b = -3h/2$.

Let's try gluing two such networks together, in order to compute two different tower functions. To make the respective roles of the two sub-networks clear I've put them in separate boxes, below: each box computes a tower function, using the technique described above. The graph on the right shows the weighted output from the *second* hidden layer, that is, it's a weighted combination of tower functions.



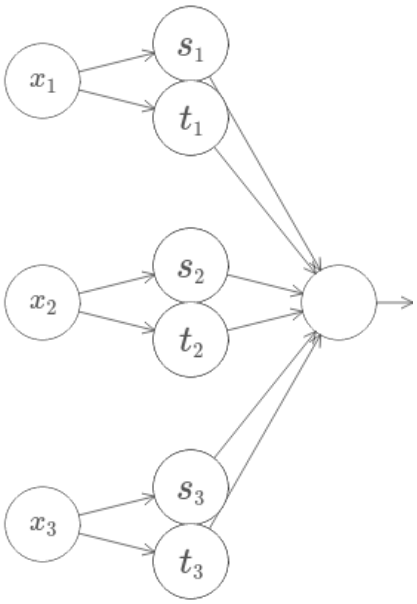
In particular, you can see that by modifying the weights in the final layer you can change the height of the output towers.

The same idea can be used to compute as many towers as we like. We can also make them as thin as we like, and whatever height we like. As a result, we can ensure that the weighted output from the second hidden layer approximates any desired function of two variables:

In particular, by making the weighted output from the second hidden layer a good approximation to $\sigma^{-1} \circ f$, we ensure the output from our network will be a good approximation to any desired function, f.

What about functions of more than two variables?

Let's try three variables x_1, x_2, x_3. The following network can be used to compute a tower function in four dimensions:



Here, the x_1, x_2, x_3 denote inputs to the network. The s_1, t_1 and so on are step points for neurons - that is, all the weights in the first layer are large, and the biases are set to give the step points $s_1, t_1, s_2, \ldots$. The weights in the second layer alternate $+h, -h$, where h is some very large number. And the output bias is $-5h/2$.

This network computes a function which is 1 provided three conditions are met: x_1 is between s_1 and t_1; x_2 is between s_2 and t_2; and x_3 is between s_3 and t_3. The network is 0 everywhere else. That is, it's a kind of tower which is 1 in a little region of input space, and 0 everywhere else.

By gluing together many such networks we can get as many towers as we want, and so approximate an arbitrary function of three variables. Exactly the same idea works in m dimensions. The only change needed is to make the output bias $(-m + 1/2)h$, in order to get the right kind of sandwiching behavior to level the plateau.

Okay, so we now know how to use neural networks to approximate a real-valued function of many variables. What about vector-valued functions $f(x_1, \ldots, x_m) \in R^n$? Of course, such a function can be regarded as just n separate real-valued functions, $f^1(x_1, \ldots, x_m)$, $f^2(x_1, \ldots, x_m)$, and so on. So we create a network approximating f^1, another network for f^2, and so on. And then we simply glue all the networks together. So that's also easy to cope with.

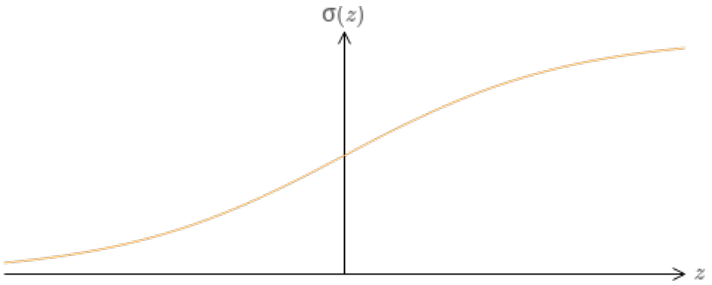
Problem

- We've seen how to use networks with two hidden layers to approximate an arbitrary function. Can you find a proof showing that

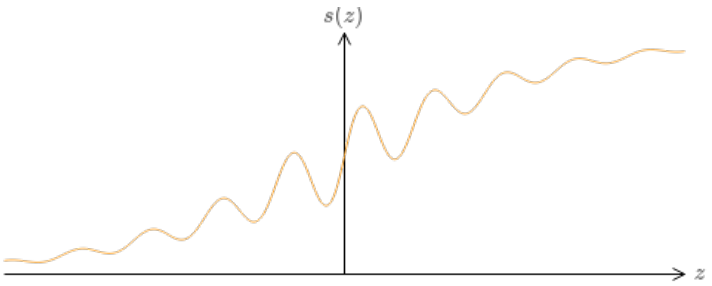
it's possible with just a single hidden layer? As a hint, try working in the case of just two input variables, and showing that: (a) it's possible to get step functions not just in the x or y directions, but in an arbitrary direction; (b) by adding up many of the constructions from part (a) it's possible to approximate a tower function which is circular in shape, rather than rectangular; (c) using these circular towers, it's possible to approximate an arbitrary function. To do part (c) it may help to use ideas from a bit later in this chapter.

Extension beyond sigmoid neurons

We've proved that networks made up of sigmoid neurons can compute any function. Recall that in a sigmoid neuron the inputs $x_1, x_2, \ldots$ result in the output $\sigma(\sum_j w_j x_j + b)$, where w_j are the weights, b is the bias, and σ is the sigmoid function:

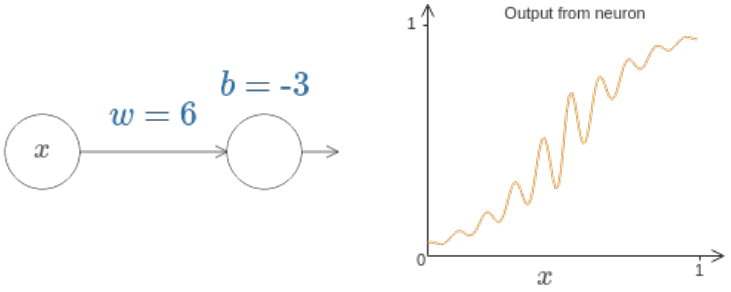


What if we consider a different type of neuron, one using some other activation function, $s(z)$:



That is, we'll assume that if our neurons has inputs $x_1, x_2, \ldots$, weights $w_1, w_2, \ldots$ and bias b, then the output is $s(\sum_j w_j x_j + b)$.

We can use this activation function to get a step function, just as we did with the sigmoid. Try ramping up the weight in the following, say to $w = 100$:



Just as with the sigmoid, this causes the activation function to contract, and ultimately it becomes a very good approximation to a step function. Try changing the bias, and you'll see that we can set the position of the step to be wherever we choose. And so we can use all the same tricks as before to compute any desired function.

What properties does $s(z)$ need to satisfy in order for this to work? We do need to assume that $s(z)$ is well-defined as $z \to -\infty$ and $z \to \infty$. These two limits are the two values taken on by our step function. We also need to assume that these limits are different from one another. If they weren't, there'd be no step, simply a flat graph! But provided the activation function $s(z)$ satisfies these properties, neurons based on such an activation function are universal for computation.

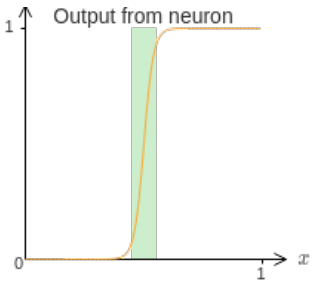
Problems

- Earlier in the book we met another type of neuron known as a rectified linear unit. Explain why such neurons don't satisfy the conditions just given for universality. Find a proof of universality showing that rectified linear units are universal for computation.

- Suppose we consider linear neurons, i.e., neurons with the activation function $s(z) = z$. Explain why linear neurons don't satisfy the conditions just given for universality. Show that such neurons can't be used to do universal computation.

Fixing up the step functions

Up to now, we've been assuming that our neurons can produce step functions exactly. That's a pretty good approximation, but it is only an approximation. In fact, there will be a narrow window of failure, illustrated in the following graph, in which the function behaves very differently from a step function:



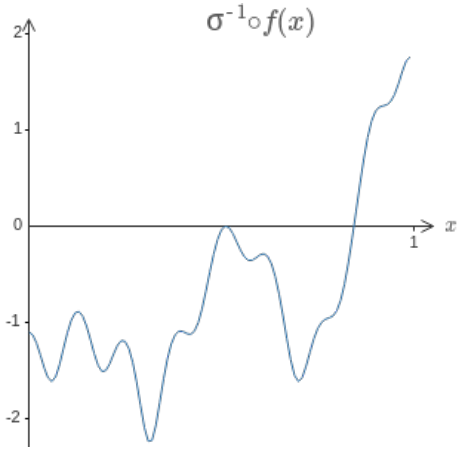
In these windows of failure the explanation I've given for universality will fail.

Now, it's not a terrible failure. By making the weights input to the neurons big enough we can make these windows of failure as small as we like. Certainly, we can make the window much narrower than I've shown above - narrower, indeed, than our eye could see. So perhaps we might not worry too much about this problem.

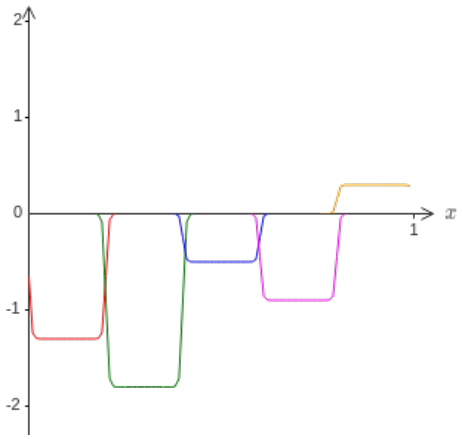
Nonetheless, it'd be nice to have some way of addressing the problem.

In fact, the problem turns out to be easy to fix. Let's look at the fix for neural networks computing functions with just one input and one output. The same ideas work also to address the problem when there are more inputs and outputs.

In particular, suppose we want our network to compute some function, f. As before, we do this by trying to design our network so that the weighted output from our hidden layer of neurons is $\sigma^{-1} \circ f(x)$:

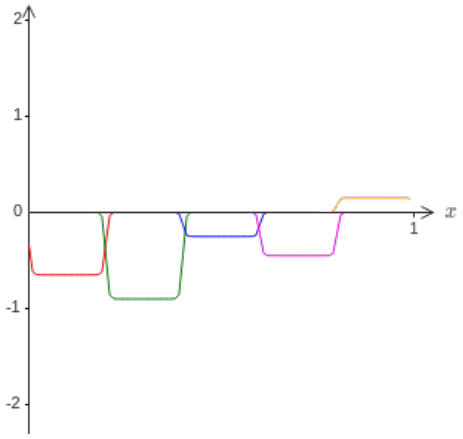


If we were to do this using the technique described earlier, we'd use the hidden neurons to produce a sequence of bump functions:

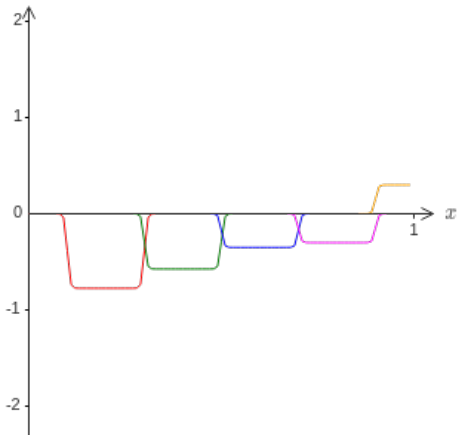


Again, I've exaggerated the size of the windows of failure, in order to make them easier to see. It should be pretty clear that if we add all these bump functions up we'll end up with a reasonable approximation to $\sigma^{-1} \circ f(x)$, except within the windows of failure.

Suppose that instead of using the approximation just described, we use a set of hidden neurons to compute an approximation to *half* our original goal function, i.e., to $\sigma^{-1} \circ f(x)/2$. Of course, this looks just like a scaled down version of the last graph:



And suppose we use another set of hidden neurons to compute an approximation to $\sigma^{-1} \circ f(x)/2$, but with the bases of the bumps *shifted* by half the width of a bump:



Now we have two different approximations to $\sigma^{-1} \circ f(x)/2$. If we add up the two approximations we'll get an overall approximation to $\sigma^{-1} \circ f(x)$. That overall approximation will still have failures in small windows. But the problem will be much less than before. The reason is that points in a failure window for one approximation won't be in a failure window for the other. And so the approximation will be a factor roughly 2 better in those windows.

We could do even better by adding up a large number, M, of overlapping approximations to the function $\sigma^{-1} \circ f(x)/M$. Provided the windows of failure are narrow enough, a point will only ever be in one window of failure. And provided we're using a large enough number M of overlapping approximations, the result will be an excellent overall approximation.

Conclusion

The explanation for universality we've discussed is certainly not a practical prescription for how to compute using neural networks! In this, it's much like proofs of universality for `NAND` gates and the like. 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. However, you may find it a fun and instructive exercise to see if you can improve the construction.

Although the result isn't directly useful in constructing networks, it's important because it takes off the table the question of whether any particular function is computable using a neural network. The answer to that question is always "yes". 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.

The universality construction we've developed uses just two hidden layers to compute an arbitrary function. Furthermore, as we've discussed, it's possible to get the same result with just a single hidden layer. 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: from edges to simple geometric shapes, all the way up through complex, multi-object scenes. In later chapters, we'll see evidence suggesting that deep networks do a better job than shallow networks at learning such hierarchies of knowledge. To sum up: universality tells us that neural networks can compute any function; and empirical evidence suggests that deep networks are the networks best adapted to learn the functions useful in solving many real-world problems.