CS 5600/6600: F20: Intelligent Systems

ANN Universality Theorem

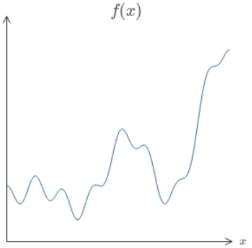
Vladimir Kulyukin
Department of Computer Science
Utah State University

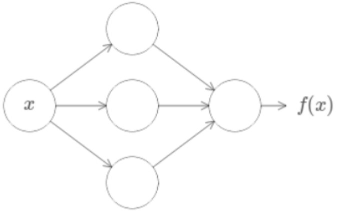
ANN Universality Theorem

ANNs can compute any function.

ANN Universality Theorem in 1D

If there is a 1D function, then there is a 1-input and 1-output ANN that computes it.





Two Qualifications about ANN Universality

Given a function $f(x)$ and some desired accuracy $\epsilon > 0$, there exists a neural network whose output $g(x)$ satisfies $|g(x) - f(x)| < \epsilon$, for all inputs x.

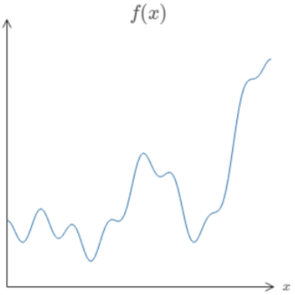
For a neural network to exist, the function $f(x)$ must be continuous. In practice, this is not an important limitation, because continouous approximates are often “good enough.”

Another Formulation of the ANN Universality Theorem

Any continuous function can be approximated with an ANN with at most two hidden layers to any desired precision.

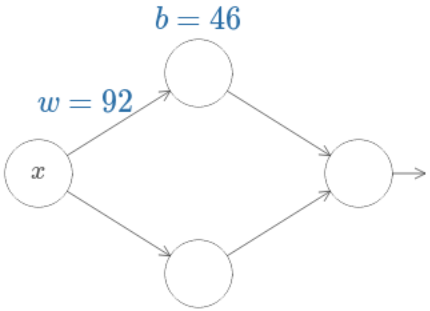
Approximation of 1D Functions

Let's start by approximating 1D functions and then extend our proof to multi-dimensional functions.



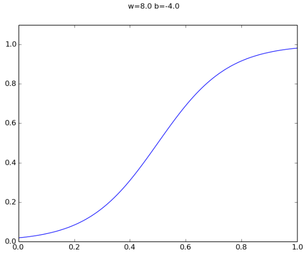
Approximation of 1D Functions

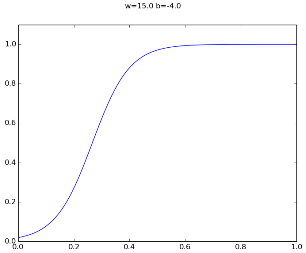
Let's start with an ANN with just two hidden neurons and try manipulating the output of the upper neuron by manipulating its bias and input weight.

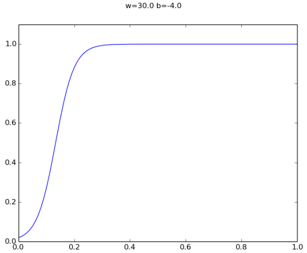


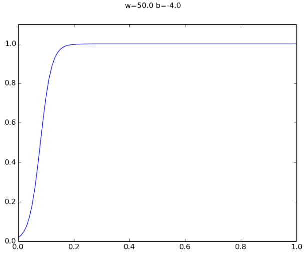
Hidden Neuron's Output

Let's keep the bias fixed at -4.0 and change the input weight.



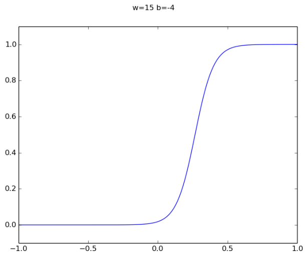


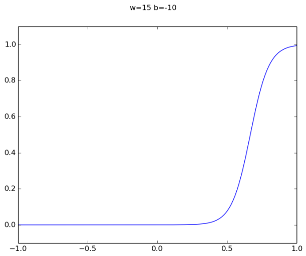


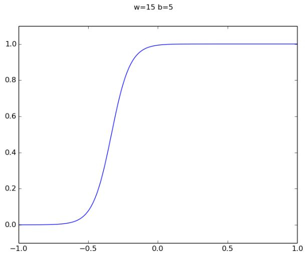


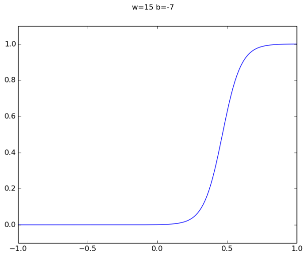
Hidden Neuron's Output as Function of Weight and Bias

Let's keep the weight fixed and change the bias.









Hidden Neuron's Output as Function of Weight and Bias

As the weight decreases, the output graph broadens out.

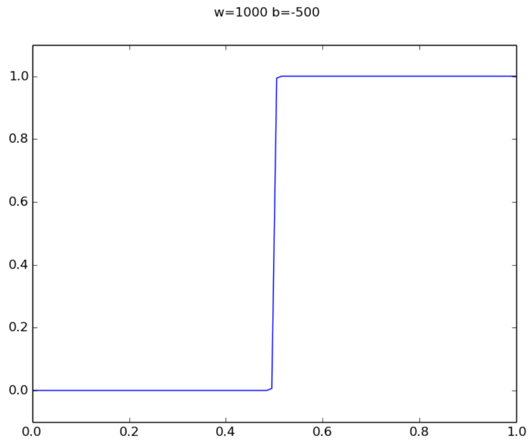
As the weight increases, the output graph becomes steeper.

As the bias increases, the output graph moves left without any shape change.

As the bias decreases, the output graph moves right without any shape change.

Example

Let's turn the neuron's output into a step function that starts at 0.5 by setting $w = 1000$ and $b = -500$.



Turning Hidden Neurons into Step Functions

Since we can change the output of each hidden neuron into a step function by changing weights and biases, we can assume that the output of each hidden neuron is a step function.

Why step functions? Because it is much easier to manipulate step functions than sigmoids.

Step Position as Function of Weight and Bias

Let s be the position of the step. Then

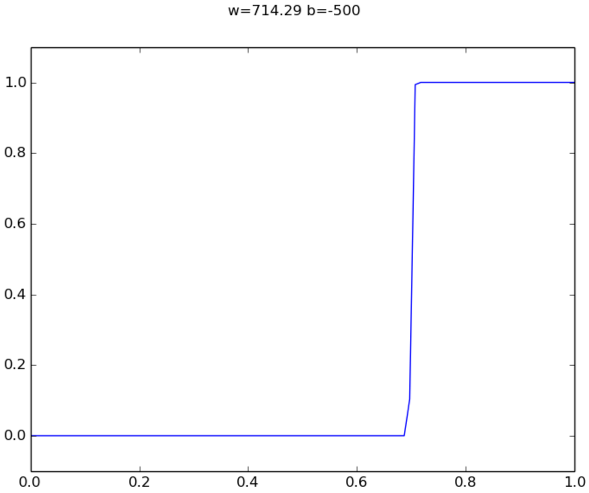
$$s = -\frac{b}{w} = f(b, w).$$

Exercise

What is the neuron's input weight when its bias is -500 and its step starts at 0.7?

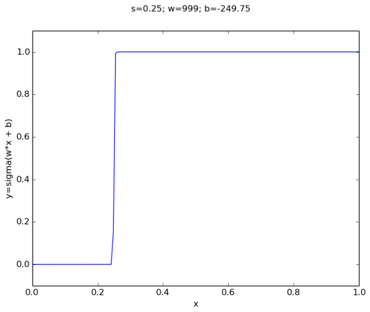
Exercise

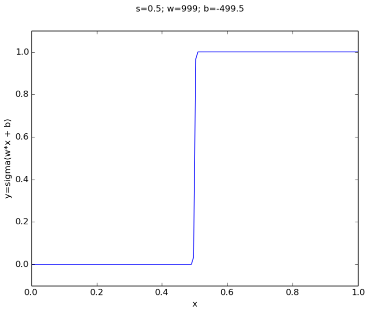
The neuron's weight is $w = -b/s = -(-500/0.7) \approx 714.29$.



Hidden Neuron's Output as Function of Step Position

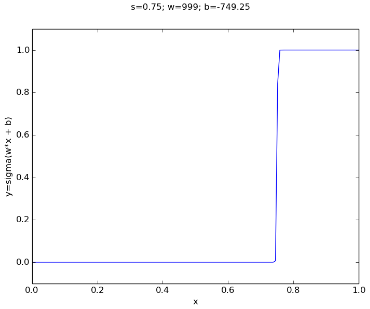
Let's keep the weight fixed and change the bias. We can fix the weight to some large value and then compute the bias from a given step position.

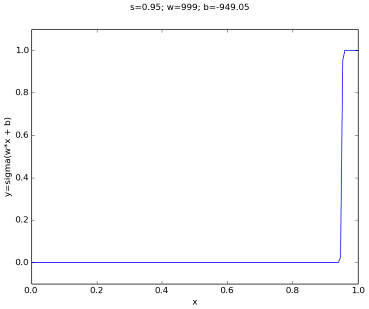




Hidden Neuron's Output as Function of Step Position

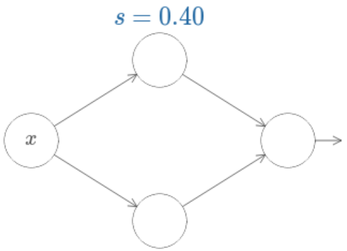
Let's keep the weight fixed and change the bias. We can fix the weight to some large value and then compute the bias from a given step position.

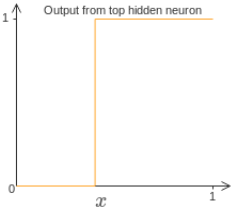




Hidden Neuron's Output as Function of Step Position

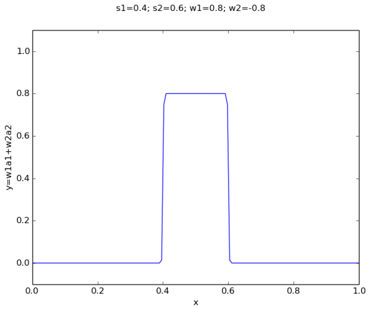
A takeaway from this discussion is that instead of using two parameters, w and b, we can use s as the single parameter to describe a neuron.

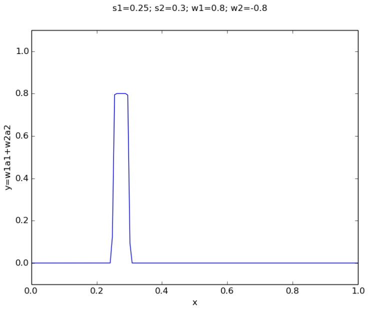




Simulating Bumps of Various Widths

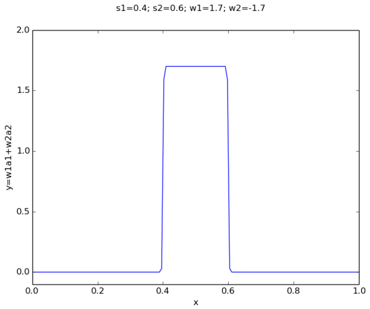
The plot of the weighted output $w_1 \cdot a_1 + w_2 \cdot a_2$ of the hidden layer, where a_1 and a_2 are the activations of the two hidden neurons.

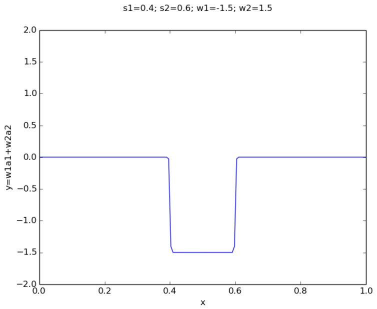




Simulating Bumps of Various Heights

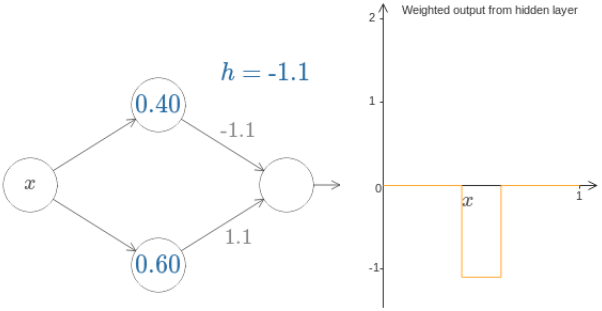
The plot of the weighted output $w_1 \cdot a_1 + w_2 \cdot a_2$ of the hidden layer, where a_1 and a_2 are the activations of the two hidden neurons.





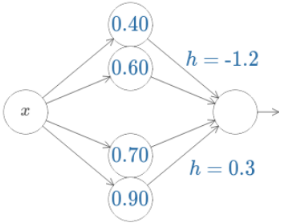
Simulating Bumps of Various Heights

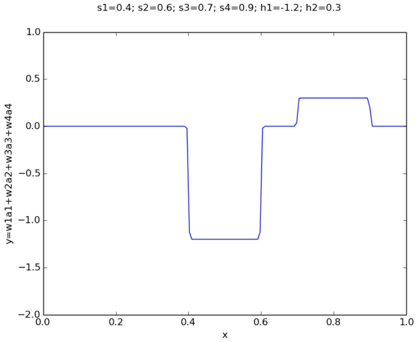
A takeway is that we can use a single height parameter *h* to have our two hidden neuron network to simulate a bump of any height.



Simulating 2 Bumps of Various Heights

The plot of the weighted output $w_1 \cdot a_1 + w_2 \cdot a_2 + w_3 \cdot a_3 + w_4 \cdot a_4$ of the hidden layer, where a_1, a_2, a_3, and a_4 are the activations of the four hidden neurons.





Question

What happens if we add more hidden neuron pairs to the hidden layer?

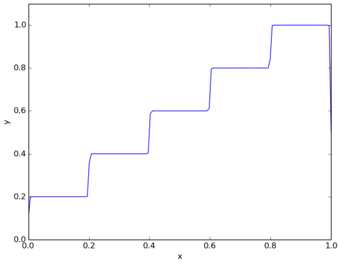
Answer

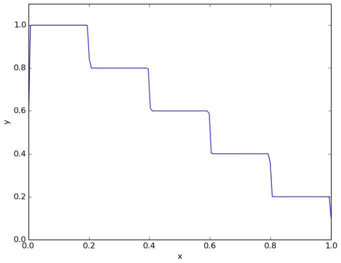
We know that a hidden neuron pair corresponds to exactly one bump.

Our neural network will be able to simulate more bumps. By adding more hidden neuron pairs we can simulate arbitrarily many bumps.

Simulating 5 Bumps

The plot of the weighted output $\sum_{i=1}^{10} w_i \cdot a_i + b$ of the hidden layer, where a_i are the activations of the 10 hidden neurons.





Takeway: Simulating Any Number of Peaks

We can divide the input [0, 1] into any large number, N, of subintervals, and use N pairs of hidden neurons to set up the N peaks of any height.

Approximating 1-Variable Functions

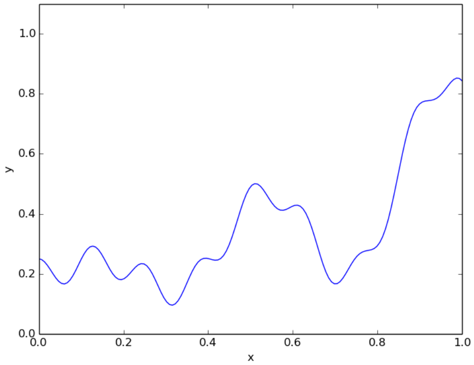
Suppose we have a function $f(x)$. We want the output of the network to be $\sigma(\sum_j w_j a_j + b) = f(x)$. What we need to do is to design the output of the hidden layer to be such that $\sum_j w_j a_j + b = \sigma^{-1}(f(x))$. Then

$$\sigma\left(\sum_j w_j a_j + b\right) = \sigma\left(\sigma^{-1}(f(x))\right) = f(x).$$

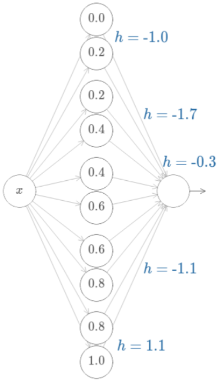
If this approximation is coarse, we can increase the number of intervals N, i.e., the number of hidden neuron pairs.

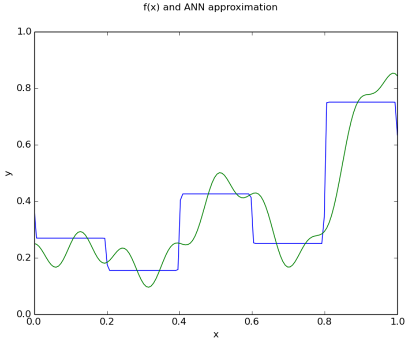
Exercise: Approximating 1-Variable Functions

$$f(x) = 0.2 + 0.4x^2 + 0.3xsin(15x) + 0.05cos(50x).$$



Exercise: Approximating 1-Variable Functions





This is the same ANN (left) where the output of the hidden layer approximates $y = \sigma^{-1}(f(x))$ and the output of the top neuron approximates $f(x)$. The graph (right) shows $f(x)$ (green) and the ann's output (blue).

Approximating 1-Variable Functions

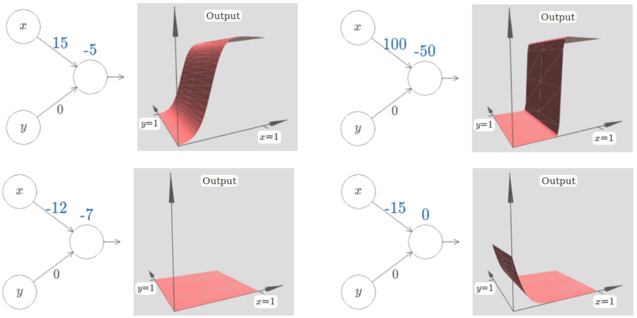
Below are the steps of building an ANN to approximate a function $f(x)$ on $[0, 1]$.

1. Choose a measure of approximation closeness (e.g., average deviation).
2. Build a ANN with k pairs of hidden neurons. This number of pairs corresponds to the number of subintervals on $[0, 1]$.
3. Iteratively modify k (increase/decrease) and the heights of the corresponding bumps until the hidden layer approximates $\sigma^{-1}(f(x))$ within a given threshold.
4. Use this network to approximate $f(x)$.

NB: You don't have to use more than one hidden layer.

Approximating 2-Variable Functions

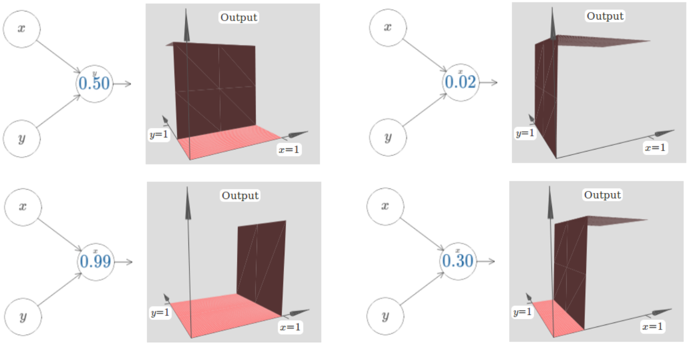
Let's keep the input weight of y and modify the input weight of x and the bias.



Some observations: 1) as the input weight gets larger, the output approaches a step function; 2) the step function is in 3D; 3) the location of the step function along the x-axis is $s_x = -b/w_1$.

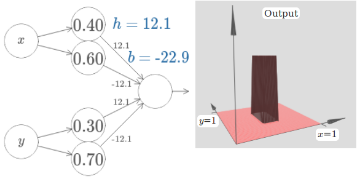
Approximating 2-Variable Functions

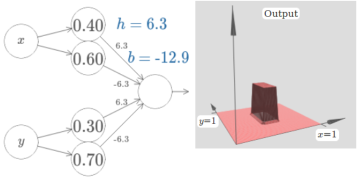
We can set w_1 to some large number, set w_2 to 0 and then express the neuron's output as a step function by manipulating the step parameter s_x.

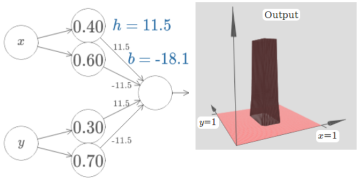


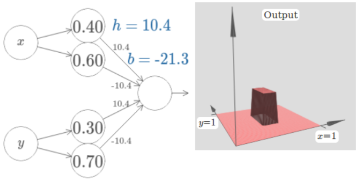
Building Towers

We can do build towers by manipulating not only the h parameter but also the bias b of the output neuron. A good heuristic is $b = -3h/2$.



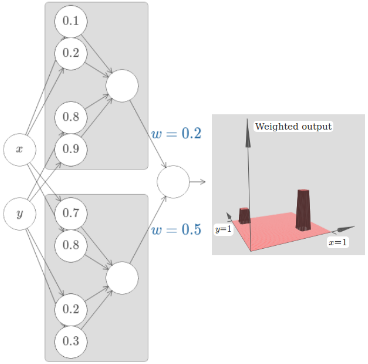


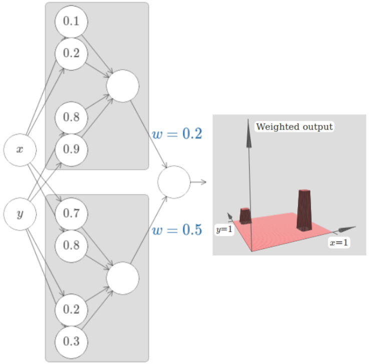




Combining Towers

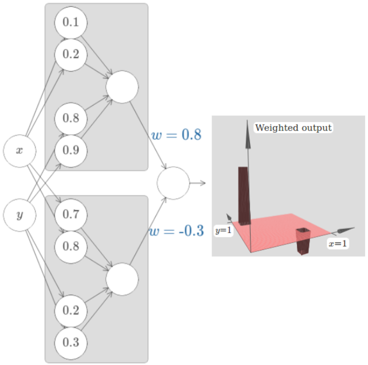
We can combine two tower functions into a single network and control their heights by the input weights to the output neuron.

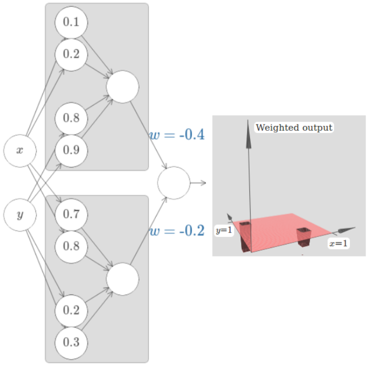




Combining Towers

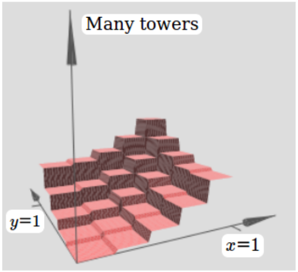
We can combine two tower functions into a single network and control their heights by the input weights to the output neuron.





Many Towers

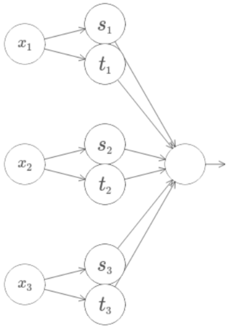
We can compute as many towers as necessary and of whatever widths, lengths, and heights. Consequently, we can ensure that the weighted output from the second hidden layer approximates any function of two variables.



Thus, if we make the weighted output from the second hidden layer approximate $\sigma^{-1} \cdot f$, we make the output of our network approximate any function $f(x_1, x_2)$.

Building 4D Towers

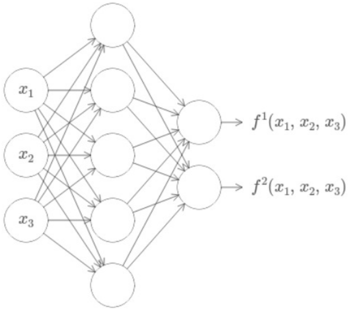
If we have three variables x_1, x_2, x_3, the following network computes a tower in 4D.



s_i, t_i are step points for neurons. The weights in the second layer alternate between $+h, -h$ where h are the heights and the output bias is $-5h/2$.

ANN Universality Theorem in Many Dimensions

The theorem holds for functions with many inputs and outputs. Suppose we have a function $f : R^m \to R^n$. There is an ANN with m inputs and n outputs that computes that function. Here is an example of an ANN that computes $f : R^3 \to R^2$.



Conclusions

The ANN Universality Theorem does not give us practical prescriptions for computing with neural networks: there is no reason why our networks should have only one or two layers.

The ANN Universality Theorem guarantees that any particular function is computable by a network.

Why do we need deeper networks, if every function, in principle, can be computed by a neural network with 1 or 2 hidden layers?