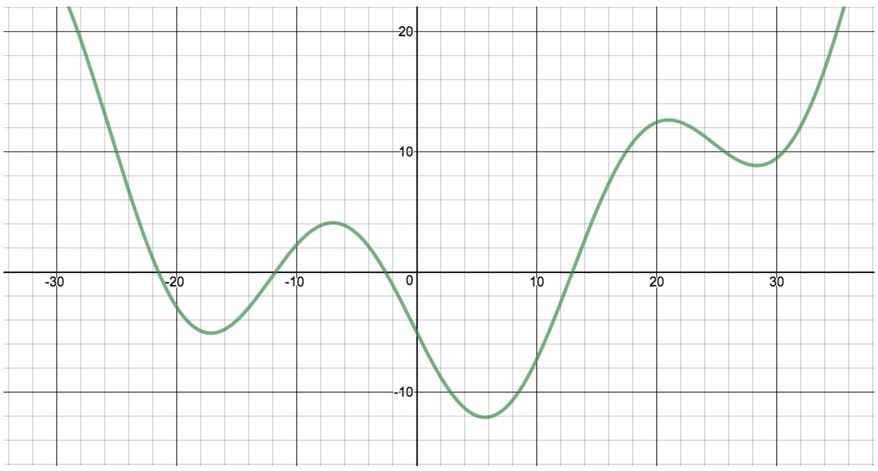
Can neural networks solve any problem?

Visualizing the Universal Approximation Theorem

Brendan Fortuner Follow
Mar 7, 2017 · 6 min read



Let's see you try modeling that, neural network!

At some point in your deep learning journey you probably came across the Universal Approximation Theorem.

> *A feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly.*
>
> *—Ian Goodfellow, DLB*

This is an incredible statement. If you accept most classes of problems can be reduced to functions, this statement implies a neural network can, in theory, solve any problem. If human intelligence can be modeled with functions (exceedingly complex ones perhaps), then we have the tools to reproduce human intelligence today. Neural nets might be AI's version of the Babbage analytical engine (1822) whereas the Terminator needs a Macbook Pro, but still. Perhaps UAT explains why deep learning has been so successfully tackling "hard problems" in AI —image recognition, machine translation, speech-to-text, etc.

TLDR; I prove to myself visually and empirically that UAT holds for a non-trivial function *(x^3+x^2-x-1)* using a single hidden layer and 6 neurons. I pretend I'm a neural network and try to "learn" the correct weights on my own. I also verify this works in code.

How can neural nets possibly model any function?

This question stumped me for a long time and I couldn't find a great explanation online. Most explanations with a sentence-to-equation ratio above .57 would read something like:

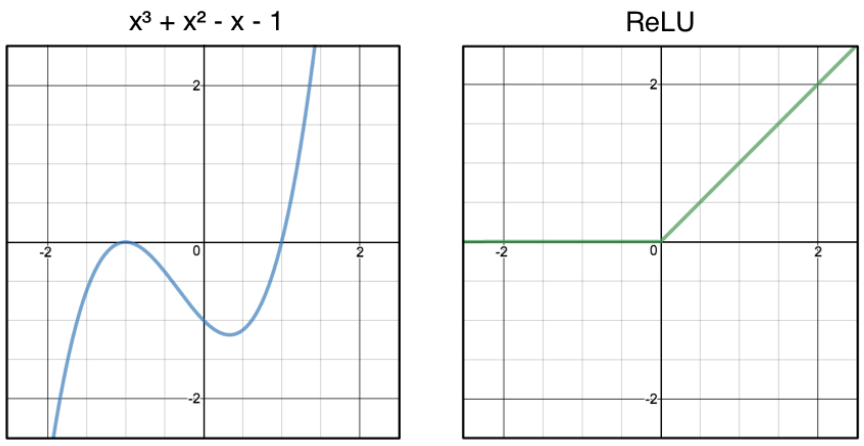
> *Introducing non-linearity via an activation function allows us to approximate any function. It's quite simple, really. —Elon Musk*

So nonlinear activation functions are the secret sauce? Can we really model *any* function by chaining together a bunch of Sigmoid activations? How about the ReLU function? Surely not—it has the word *linear* in it! Rectified Linear Units!

I eventually discovered Michael Neilson's tutorial which is so good it nearly renders this post obsolete (I highly encourage you to read it!), but for now let's travel back in time and pretend Michael took his family to Disneyland that day instead of writing the world's greatest tutorial on neural networks ever. Thanks, Michael ;)

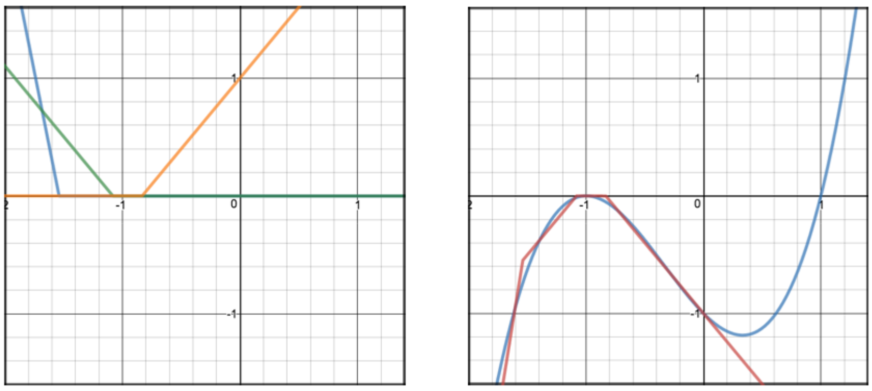
Performing gradient descent by hand

I realized early on I wasn't going to win this battle perusing mathematical proofs, so I decided to take an experimental approach. I went to Desmos and started chaining together ReLU activation functions to see if I could build something interesting-looking. After each attempt I tweaked my functions to make them look more like the target—sound familiar?



Left: target function **Right:** raw materials

I chose x^3+x^2-x-1 as my target function. Using only ReLU `max(0,x)` , I iteratively tried different combinations of ReLUs until I had an output that roughly resembled the target. Here are the results I achieved taking the weighted sum of 3 ReLUs.



Left: 3 ReLU functions **Right**: Weighted sum of the 3 ReLU functions

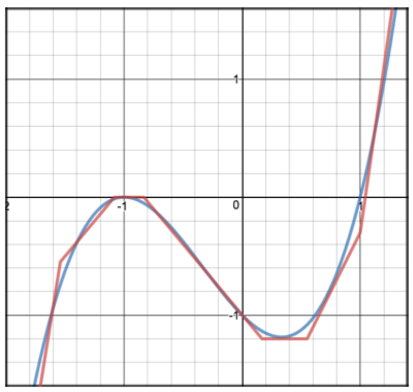
Not bad? The left chart shows the ReLU functions. The right chart shows the output of my model compared to the target. You can think of each ReLU function as a neuron. So combining 3 ReLU functions is like training a network of 3 hidden neurons. Here are the equations I used to generate these charts.

$$
\begin{array}{ll}
\text{Neuron Activations} & \begin{aligned} n_1(x) &= Relu(-5x-7.7) \\ n_2(x) &= Relu(-1.2x-1.3) \\ n_3(x) &= Relu(1.2x+1) \end{aligned} \\
\text{Weighted Output} & Z(x) = -n_1(x) - n_2(x) - n_3(x)
\end{array}
$$

Each neuron's output equals ReLU wrapped around the weighted input ***wx* + *b***.

I found I could shift the ReLU function left and right by changing the bias and adjust the slope by changing the weight. I combined these 3 functions into a final sum of weighted inputs (**Z**) which is standard practice in most neural networks.

The negative signs in **Z** represent the final layer's weights which I set to -1 in order to "flip" the graph across the x-axis to match our concave target. After playing around a bit more I finally arrived at the following 7 equations that, together, roughly approximate x^3+x^2-x-1.



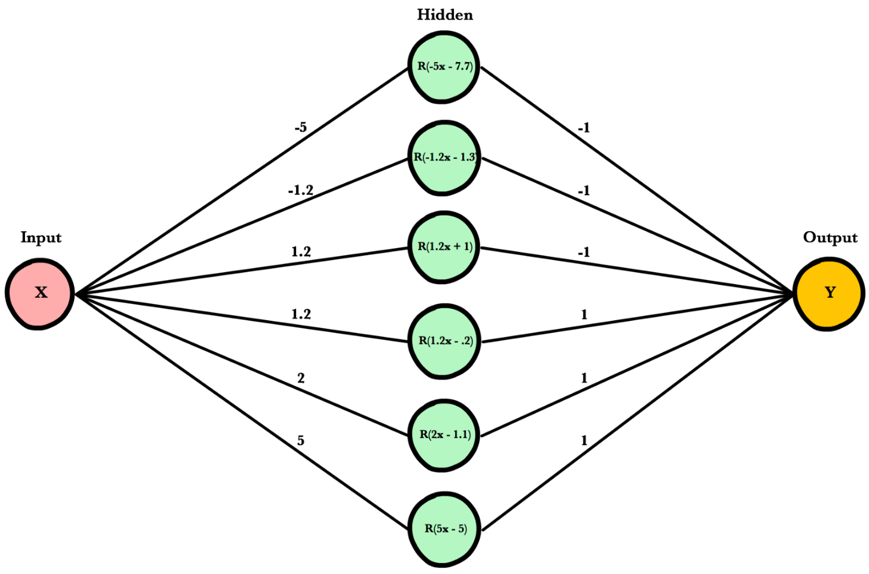
$$
\begin{aligned}
n_1(x) &= Relu(-5x - 7.7)\\
n_2(x) &= Relu(-1.2x - 1.3)\\
n_3(x) &= Relu(1.2x + 1)\\
n_4(x) &= Relu(1.2x - .2)\\
n_5(x) &= Relu(2x - 1.1)\\
n_6(x) &= Relu(5x - 5)
\end{aligned}
$$

$$
\begin{aligned}
Z(x) = &-n_1(x) - n_2(x) - n_3(x)\\
&+ n_4(x) + n_5(x) + n_6(x)
\end{aligned}
$$

So visually at least, it looks like it IS possible to model non-trivial functions with a single hidden layer and a handful of neurons. Pretty cool.

Hard-coding my weights into a real network

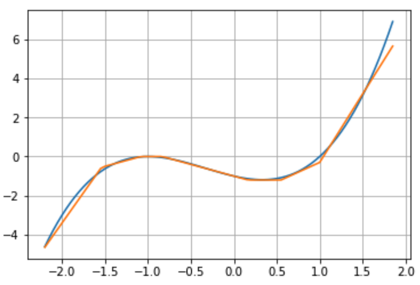
Here is a diagram of a neural network initialized with my fake weights and biases. If you give this network a dataset that resembles x^3+x^2-x-1, it *should* be able approximate the correct output for inputs between -2 and 2.

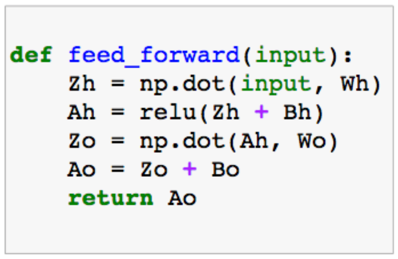


That last statement, *approximate the correct output for any input between -2 and 2,* is key. The Universal Approximation Theorem states that a neural network with 1 hidden layer can approximate any ***continuous*** function for inputs within a ***specific range***. If the function jumps around or has large gaps, we won't be able to approximate it. Additionally, if we train a network on inputs between 10 and 20, we can't say for sure whether it will work on inputs between 40 and 50.

Do my weights actually work?

I wanted to prove programmatically the weights I came up with actually work when plugged into a basic neural network with one hidden layer and 6 neurons. Instead of training the network to learn weights, however, I replaced its default weights and biases with my hand-picked values. The method `feed_forward()` below takes a vector of inputs (e.g. [-2,-1,0,1..]) and outputs a vector of predictions using my weights. Here's what happened:

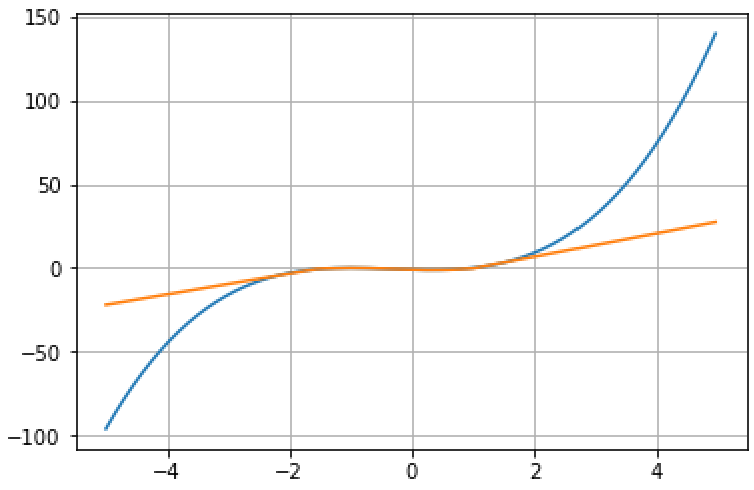




```
def feed_forward(input):
    Zh = np.dot(input, Wh)
    Ah = relu(Zh + Bh)
    Zo = np.dot(Ah, Wo)
    Ao = Zo + Bo
    return Ao
```

Zh,Zo: weighted inputs . Wh,Wo: weights. Bh,Bo: biases. Ah,Ao: activations

Look at that! I'm a genius. It's exactly what we wanted. But what if our boss asks us to expand the range beyond -2 to 2? What if she wants -5 to 5?



x = np.arange(-5, 5, .1)

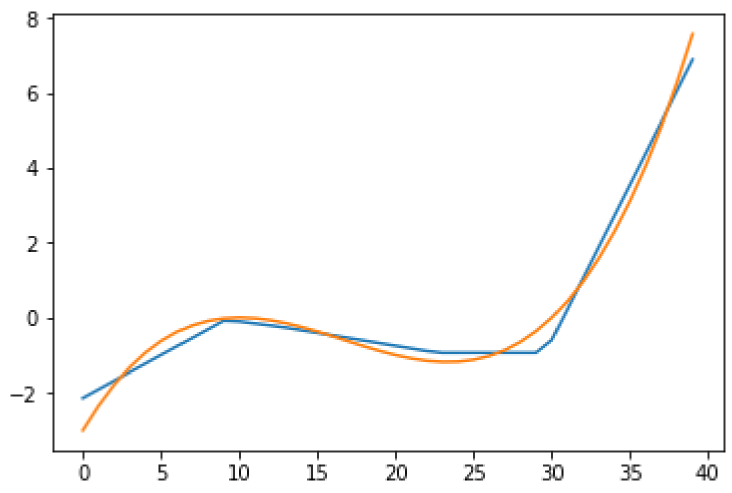
Ah, not so great. But this actually supports our earlier conclusion: a neural network with one hidden layer can approximate any continuous function but only for inputs within a ***specific range***. If you train a network on inputs between -2 and 2, like we did, then it will work well for inputs in a similar range, but you can't expect it to generalize to other inputs without retraining the model or adding more hidden neurons.

Can my weights be learned?

Now we know Brendan Fortuner can learn these weights on his own, but can a real-world neural network with 1 hidden layer and 6 neurons also learn these parameters or others that lead to the same result? Let's use scikit-neuralnetwork to test this theory. We'll design a network so it works for regression problems and configure it to use ReLU, Stochastic Gradient Descent, and Mean Squared Error—the usual gang.

```python
from sknn.mlp import Regressor
from sknn.mlp import Layer

# Design Network
hiddenLayer = Layer("Rectifier", units=6)
outputLayer = Layer("Linear", units=1)
nn = Regressor([hiddenLayer, outputLayer], learning_ru
               learning_rate=.001,batch_size=5,loss_ty

# Generate Data
def cubic(x):
    return x**3 + x**2 - x - 1

def get_cubic_data(start,end,step_size):
    X = np.arange(start, end, step_size)
    X.shape = (len(X),1)
    y = np.array([cubic(X[i]) for i in range(len(X))])
    y.shape = (len(y),1)
    return X,y

```

It worked!



Here are the weights Scikit learned. They're clearly different from my own, but the order of magnitude is the same.

```
hiddenLayerWeights = [
 [-1.5272, -1.0567, -0.2828, 1.0399, 0.1243, 2.2446]
]
finalLayerWeights = [
  [-2.2466],
  [ 1.0707],
  [-1.0643],
  [ 1.8229],
  [-0.4581],
  [ 2.9386]
]
```

Perhaps if I rewrote this blog post 100,000 times I would have come to these parameters on my own, but for now we can only speculate. Maybe someday I'll take the derivative of my up-votes and update my sentences in the direction that maximizes views.

. . .

Please hit the ♥ button below if you enjoyed reading!