Lab 8.5 - Universal Approximation Theorem

Kyle Swanson

January 24, 2018

0 Introduction

In Lecture 7 I stated the Universal Approximation Theorem about single layer neural networks:

Theorem 1 (Universal Approximation Theorem) *A feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of* $\mathbb{R}^2$ *(under mild assumptions on the activation function).*

The theorem essentially says that a single layer neural network can approximate any reasonable function. In this lab, we'll be testing the limits of this theorem by using single layer neural network to approximate 3D functions, i.e. functions of two variables $z = f(x, y)$. The input to the network will be the values x and y and the network will learn to predict the value z.

1 Pythagorean Function

The first function we will try to approximate is what I'm calling the "Pythagorean function". This function is based on the idea of Pythagorean triples. A Pythagorean triple is a set of three integers a, b, c such that $a^2 + b^2 = c^2$. We can rewrite this as a 3D function $c = \sqrt{a^2 + b^2}$, and we'll let a, b, c be real numbers. In this section, we will train a neural network to predict c given inputs a and b with the hope that the network will learn to predict the function $c = \sqrt{a^2 + b^2}$.

1.1 Learning the Pythagorean Function

First, we need to build a single layer neural network for this task. Go into `lab8_5.py` and implement the function `build_single_layer_regression_model`. This function should build a keras Sequential model with two Dense layers. The first layer, which is the hidden layer, should have `n_hidden` neurons with `input_dim=2` (our input is the two numbers a and b) and with tanh activation. The second layer, which is the output layer, should have 1 neuron and linear activation. We are using linear activation instead of softmax activation for the final layer because we are trying to output a real number rather than a class (this is a regression problem rather than a classification problem).

Once your implementation is complete, go into `main.py`, uncomment Part 1.1, and run `python main.py`. Your model will be trained on data from the Pythagorean function four times: once for 10 epochs, once for 20 epochs, once for 100 epochs, and once for 500 epochs. Once training is complete, you'll see an image with four plots showing the Pythagorean function and your model's predictions after 10 epochs, 20 epochs, 100 epochs, and 500 epochs. Note how with only 10 or 20 epochs your model's predictions barely look like the cone produced by the Pythagorean function, but by 100 and definitely by 500 epochs the approximation gets to be really good.

This approximation was done with a neural network with a single hidden layer with just 50 neurons, so this is a good demonstration of the Universal Approximation Theorem in action.

1.2 Generalizing the Pythagorean Function

In the plots produced in Part 1.1, we trained on a and b values between -100 and 100 and we plotted the model's predictions for a and b values between -100 and 100. This illustrates that the model can learn to approximate the function for values within its training data, but how well does it generalize to values outside its training data?

In Part 1.2, I've written code which trains the model on a and b values between -100 and 100 but which plots the model's predictions on values between -200 and 200. Uncomment Part 1.2 in `main.py` and run the code. Once training is complete, you'll see an image with two plots. On the left is the Pythagorean function and on the right is your model's predictions. You can see that near the middle (where a and b are between -100 and 100) the model's predictions are a great approximation of the Pythagorean function. However, as soon as a and b exceed the range that the model was trained on, the approximation breaks down and the model's prediction looks nothing like the Pythagorean function. Therefore, our model is a great approximator for the data ranges it is trained on, but it does not generalize. However, the Universal Approximation Theorem says nothing about the network's ability to generalize, so this result does not contradict the Universal Approximation Theorem.

2 Ripple Function

Now we'll test the limits of the Universal Approximation theorem with a different function called the "ripple function". The ripple function is defined as $z = \sin(10 * x^2 + y^2)$.

2.1 Limits of the Universal Approximation Theorem

First we'll try to approximate the ripple function with a single layer neural network like we did with the Pythagorean function. Uncomment Part 2.1 in `main.py` and run the code. After training, you'll see an image with a plot of the ripple function on the left and a plot of your model's predictions on the right. You should see that your model's predictions look nothing like the ripple function.

Clearly, a single hidden layer with 50 neurons is not enough to learn the ripple function. Try adding more neurons to the layer by changing `n_hidden` (under "Parameters" in Part 2.1 in `main.py`) to a large number like 100 or 1000; then try running the code again. Are your model's predictions any better? Experiment with different values of `n_hidden`. According to the Universal Approximation Theorem, our model should be able to approximate the ripple function as long as we make `n_hidden` large enough. However, no matter how many neurons I try, I'm unable to get a good approximation (let me know if you can get a decent approximation). Thus, even though the Universal Approximation Theorem says that it is possible to approximate the function with enough neurons in one layer, it seems that approximating the ripple function must require an exceedingly large number of neurons. So while the theorem may hold in theory, it's not always useful in practice.

2.2 Power of Deep Networks

Now that we've seen the limits of single layer neural networks, we'll see the power of deep neural networks.

Go into `lab8_5.py` and implement the function `build_deep_regression_model`. This function should build a keras Sequential model with `n_layers` Dense layers, each with `n_hidden` neurons and tanh activation. Note that in the first of these layers, you must specify `input_dim=2`. The model should also have one more layer, a Dense layer with 1 neuron and linear activation, at the end.

Once your implementation is complete, go into `main.py`, uncomment Part 2.2, and run the code. This will build a deep neural network with 3 hidden layers, each with 50 neurons, and it will train the network on the ripple function. Once training is complete, you'll see an image with the ripple function plotted on the left and your model's predictions plotted on the right. You can see that your deep network produces an excellent approximation of the ripple function.

Note how a deep network with 3 layers of 50 neurons each (150 neurons total) is able to approximate a function that a single layer with 1000 neurons could not. This demonstrates the power of uses depth in neural networks as opposed to just adding more neurons.

3 Custom Functions

Now it's your turn to try approximating functions! In this part, I've written code that will take a function named `custom`, which you will define in `main.py`, and will approximate it either using your single layer neural network (Part 3.1) or your deep neural network (Part 3.2). Try implementing the function `custom` with different 3D functions and see which ones the single layer network is able to approximate and which ones the deep network is able to approximate. Feel free to play around with the parameters `n_hidden` (number of neurons per layer), `n_layers` (number of layers), and `epochs` (number of epochs). You may also change `low` and `high`, which indicate the lowest and highest values of x and y used when generating data and plotting the functions and your model predictions. You can find suggestions for cool 3D functions here: `https://www.benjoffe.com/code/tools/functions3d/examples` Have fun!