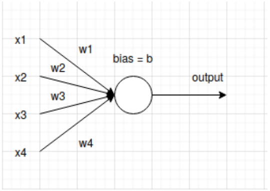
A Visual Proof of the Universal Approximation Theorem

Somnath Sikdar

March 11, 2020

Neurons and Neural Networks

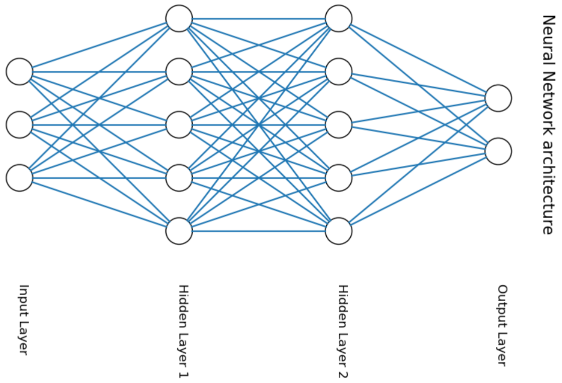
- Neural networks are comprised of neurons.
- A single neuron can accept an arbitrary number of inputs $x_1, \ldots, x_k$.



- The output of a neuron is $\Psi(\sum_{i=1}^{k} w_i x_i + b)$, where $w_1, \ldots, w_k$ are the input weights, b is the bias and Ψ is called the activation function.
- A typical activation function is the sigmoid function

$$\sigma(x_1, \ldots, x_k) = \frac{1}{1 + e^{-(\sum_i w_i x_i + b)}}.$$

Feedforward Neural Networks



- A network with m input nodes and n output neurons compute functions from $\mathbf{R}^m$ to $\mathbf{R}^n$.
- What is this class of functions?

The Universal Approximation Theorem

It turns out that:

> *Feedforward networks can compute any continuous function from $\mathbf{R}^m$ to $\mathbf{R}^n$ to any desired degree of accuracy.*

First shown by:

- George Cybenko. *Approximation by superpositions of a sigmoidal function*. Mathematics of Control, Signals and Systems, Volume 2, 1989.
- Kurt Hornik, Maxwell Stinchcombe, Halbert White. *Multilayer feedforward networks are universal approximators*. Neural Networks, Volume 2, Issue 5, 1989.

Hornik et al.

> ... *that standard multilayer feedforward networks with as few as one hidden layer using arbitrary squashing functions are capable of approximating any Borel measurable function from one finite dimensional space to another to any desired degree of accuracy, provided sufficiently many hidden units are available.*

What we will show

Feedforward networks with

- two hidden layers with sigmoid neurons
- an output layer with linear neurons

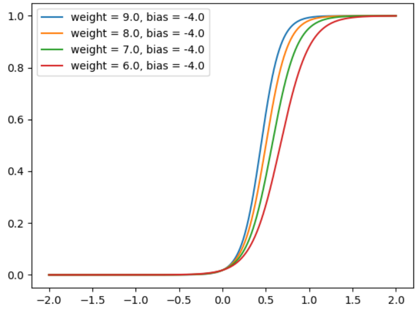
can approximate any continuous function from one real vector space to another.

Comparison with Hornik et al.

- Continuous functions are Borel functions but not all Borel functions are continuous.
- Any activation function $\Psi\colon \mathbf{R} \to [0, 1]$ that is non-decreasing with $\lim_{x\to\infty} \Psi(x) = 1$ and $\lim_{x\to-\infty} \Psi(x) = 0$ will do. Can have countably many discontinuities.
- **One** hidden layer with activation functions and the output layer with neurons with a linear activation.

Single Sigmoid Neurons

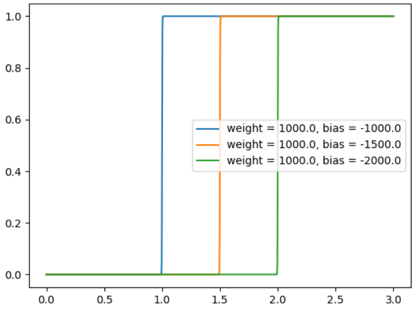
$$\sigma(x) = \frac{1}{1 + e^{-(wx+b)}}.$$



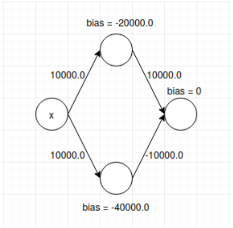
- The sigmoid squishes values to the interval $[0, 1]$.
- At the point $-b/w$, the function attains the value $1/2$.

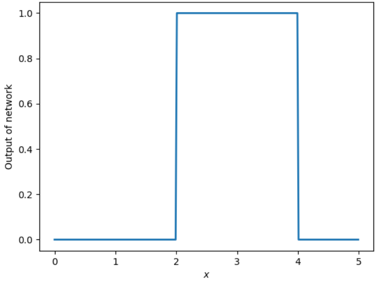
Using Sigmoid Neurons to Approximate Step Functions

If we use a large enough weight, then at the point $-b/w$, we can cause the function to step up from 0 to 1.

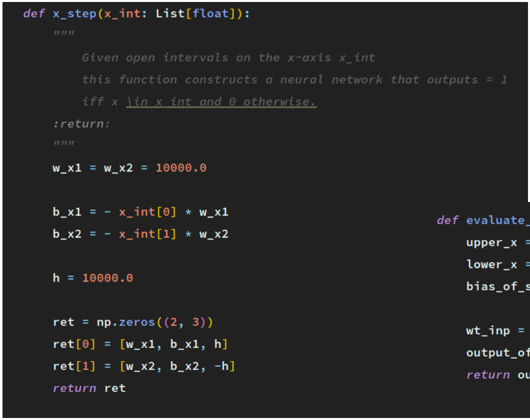


From Step to Rectangle Functions

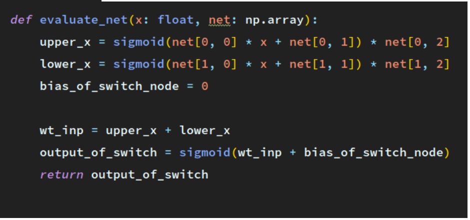




Code for the Rectangle Function

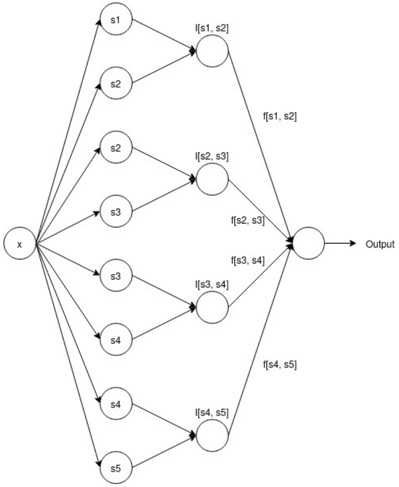


```
def x_step(x_int: List[float]):
    """
        Given open intervals on the x-axis x_int
        this function constructs a neural network that outputs = 1
        iff x \in x_int and 0 otherwise.
    :return:
    """
    w_x1 = w_x2 = 10000.0

    b_x1 = - x_int[0] * w_x1
    b_x2 = - x_int[1] * w_x2

    h = 10000.0

    ret = np.zeros((2, 3))
    ret[0] = [w_x1, b_x1, h]
    ret[1] = [w_x2, b_x2, -h]
    return ret
```



```
def evaluate_net(x: float, net: np.array):
    upper_x = sigmoid(net[0, 0] * x + net[0, 1]) * net[0, 2]
    lower_x = sigmoid(net[1, 0] * x + net[1, 1]) * net[1, 2]
    bias_of_switch_node = 0

    wt_inp = upper_x + lower_x
    output_of_switch = sigmoid(wt_inp + bias_of_switch_node)
    return output_of_switch
```

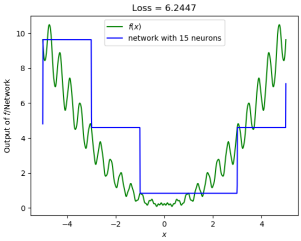
From Rectangle Functions to Univariate Functions

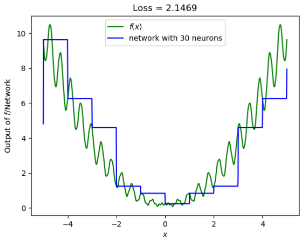


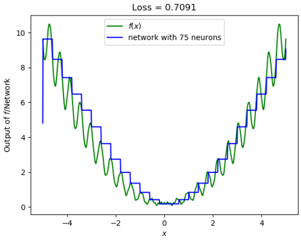
- The output neuron has a linear activation.
- Hidden layers have only sigmoid neurons.
- When $x \in [s_i, s_{i+1}]$ for $i \in \{1, 2, 3, 4\}$, the corresponding sub-network $I[s_1, s_{i+1}]$ outputs a 1; all other sub-networks output 0.
- The final output of the network is then $f[s_i, s_{i+1}]$, the approximate value of f in this interval.

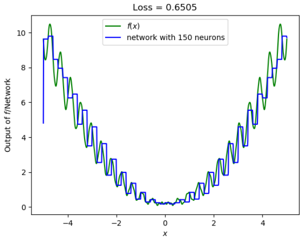
From Rectangle Functions to Univariate Functions

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



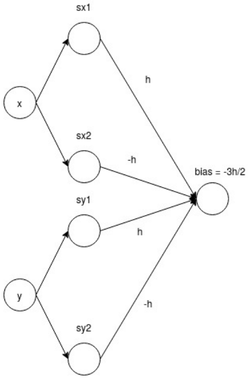


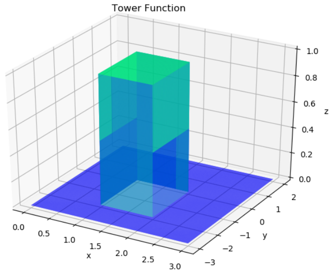




From Univariate to Bivariate Functions

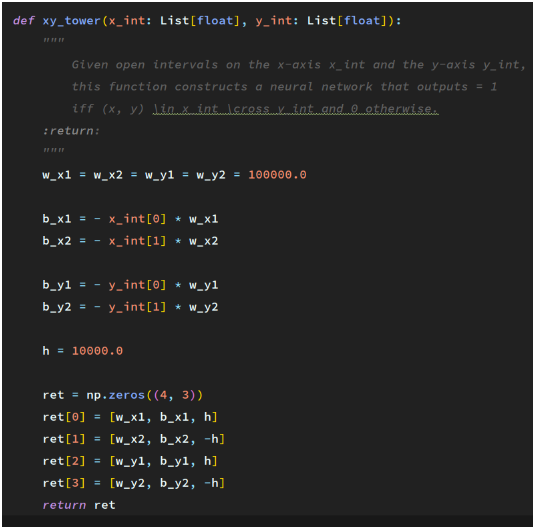
Generalize rectangle functions in the 2D plane:





- $s_x = [1.0, 2.0]; s_y = [-2.0, -0.0];$
- $w_{x1} = w_{x2} = w_{y1} = w_{y2} = 100000.0;$
- $b_{x1} = -100000.0;\ b_{x2} = -200000.0;\ b_{y1} = 200000.0;\ b_{y2} = 0.0;$
- $h = 10000.0$

Code for the Tower Function



```python
def xy_tower(x_int: List[float], y_int: List[float]):
    """
        Given open intervals on the x-axis x_int and the y-axis y_int,
        this function constructs a neural network that outputs = 1
        iff (x, y) \in x_int \cross y_int and 0 otherwise.
    :return:
    """
    w_x1 = w_x2 = w_y1 = w_y2 = 100000.0

    b_x1 = - x_int[0] * w_x1
    b_x2 = - x_int[1] * w_x2

    b_y1 = - y_int[0] * w_y1
    b_y2 = - y_int[1] * w_y2

    h = 10000.0

    ret = np.zeros((4, 3))
    ret[0] = [w_x1, b_x1, h]
    ret[1] = [w_x2, b_x2, -h]
    ret[2] = [w_y1, b_y1, h]
    ret[3] = [w_y2, b_y2, -h]
    return ret
```

Generalizing to Multivariate Functions

- A function f from $\mathbf{R}^m$ to $\mathbf{R}^n$ can be decomposed into n functions $f_1(x_1, \ldots, x_m), \ldots, f_n(x_1, \ldots, x_m)$ from $\mathbf{R}^m \to \mathbf{R}^1$.
- Hence this case can be handled by combining n networks, one for each of the n functions.

Code, References

All code available at:

- `https://github.com/somnath1077/MachineLearningExercises/tree/master/NN_DeepLearning/other_code`

References

- Michael Nielsen. Neural Networks and Deep Learning, Chapter 4. Available at: `http://neuralnetworksanddeeplearning.com/`