Basics of Feed-Forward Neural Networks

Author: Nigel Nelson

In this lab, we will start to create a feed-forward neural network from scratch. We begin with the very basic computational unit, a perceptron, and then we will add more layers and increase the complexity of our network. Along the way, we will learn how a perceptron works, the benefits of adding more layers, the kind of transformations necessary for learning complex features and relationships from data, and why an object-oriented paradigm is useful for easier management of our neural network framework.

We will implement everything from scratch in Python using helpful libraries such as NumPy and PyTorch (without using the autograd feature of PyTorch). The purpose of this lab and the following lab series is to learn how neural networks work starting from the most basic computational units and proceeding to deeper and more networks. This will help us better understand how other popular deep learning frameworks, such as PyTorch, work underneath. You should be able to easily understand and implement everything in this lab. If you are having trouble consult with your instructors as the next lab series will assume a perfect understanding of the basic feed-forward neural network material.

The recommended Python version for this implementation is 3.7. Recommended reading: sections 4.1 and 4.2 of the book (https://www.d2l.ai/chapter_multilayer-perceptrons/index.html (https://www.d2l.ai/chapter_multilayer-perceptrons/index.html)).

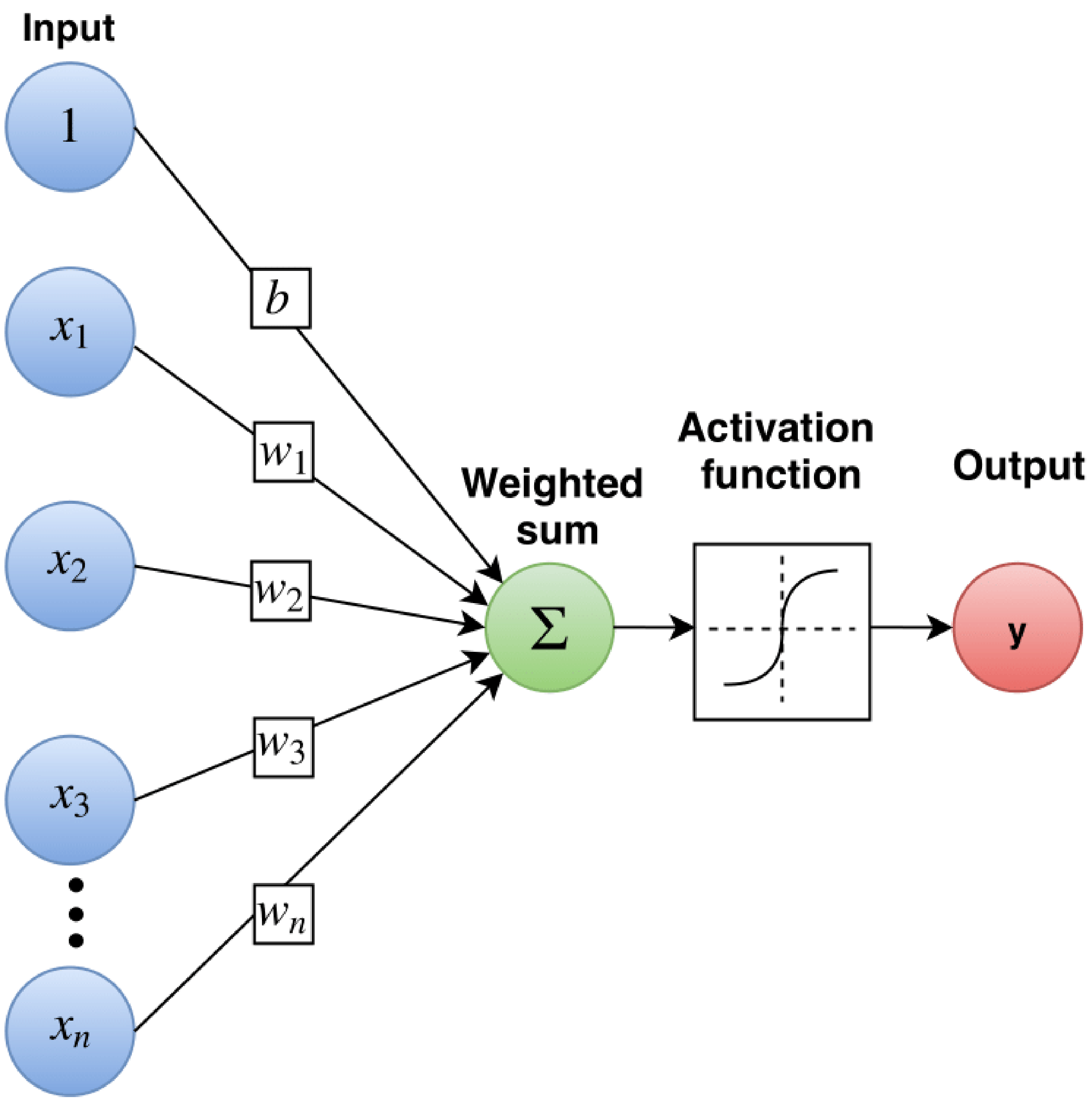
Perceptron

A perceptron or artificial neuron is the most basic processing unit of feed-forward neural networks. A perceptron can be modeled as a single-layer neural network with an input vector $\mathbf{x} \in \mathbb{R}^n$, a bias b, a vector of trainable weights $\mathbf{w} \in \mathbb{R}^n$, and an output unit y. Given the input $\mathbf{x}$, the output y is computed by an activation function $f(\cdot)$ as follows:

$$y(\mathbf{x}; \Theta) = f\left(\left(\sum_{i=1}^{n} x_i w_i\right) + b\right) = f(\mathbf{w}^\intercal \mathbf{x} + b)\,,$$

where $\Theta = \{\mathbf{w}, b\}$ represents the trainable parameter set.

The figure below shows a schematic view of a single output perceptron. Each input value x_i is multiplied by a weight factor w_i. The weighted sum added to the bias is then passed through an activation function to obtain the output, y.



The vector $\mathbf{x}$ represents one sample of our data and each element x_i represents a feature. Thus, $\mathbf{x}$ is often referred to as a feature vector. These features can represent different measurements depending on the application. For example, if we are trying to predict if a patient is at high risk of cardiac disease then each element of $\mathbf{x}$ might contain vital signs such as diastolic and systolic blood pressure, heart rate, blood sugar levels, etc. In another application where we are trying to predict if a tissue biopsy is cancerous or not using mid-infrared imaging then each element of $\mathbf{x}$ can represent the amount of mid-infrared light absorbed at a particular wavelength. The output y in the applications above could contain values of 0 or 1, indicating if the patient is at high risk of cardiac disease or if the tissue biopsy is cancerous or not.

Now, let us begin implementing our first artificial neuron.

Implementation

Let's assume that our feature vector contains measurements of body temperature pressure, pulse oximeter reading, and presence of cough or not. Then for a 'healthy' patient our input sample might look like $\mathbf{x} = \begin{bmatrix} 98.6 \\ 95 \\ 0 \end{bmatrix}$. Let's say that we are trying to 'predict' the probability of a patient being positive with COVID-19 based on the above measurements.

Each element of our input vector is associated with a unique weight. Let the vector of weights be $\mathbf{w} = \begin{bmatrix} 0.03 \\ 0.55 \\ 0.88 \end{bmatrix}$. Each artifical neuron is also associated with a unique bias. Let the bias be $b = 2.9$.
Assuming a linear activation function write the code to produce and print the output y given the above input vector $\mathbf{x}$, weights $\mathbf{w}$, and the bias b using the above perceptron model. Do not use any NumPy or PyTorch functions. Use a Python variables for each mathematical variable and use Python lists for vectors.

For the activation function, use ReLU. This can be computed as `x * (x > 0)` in Python.

In [1]:
```
x = [98.6, 95, 0]
w = [0.03, 0.55, 0.88]
b = 2.9

def ReLU(x):
    return x * (x > 0)
```

In [2]:
```
y = ReLU(sum([x*w for x,w in zip(x,w)]) + b)
print(f'The output of the perceptron is {y}')
```

```
The output of the perceptron is 58.108000000000004
```

Question 1: How many parameters does our simple model contain? Be specific.

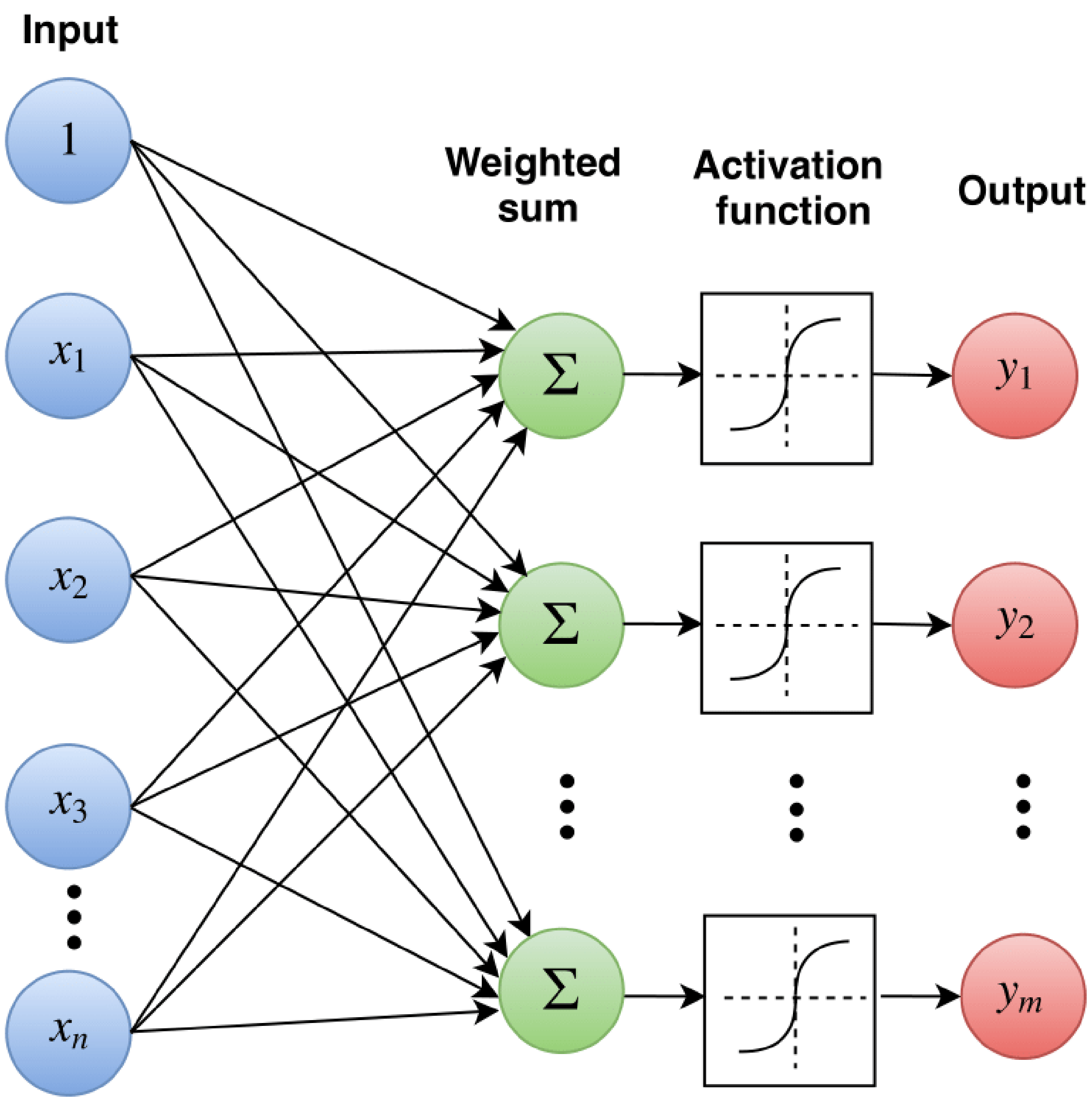
Ans: The model contains 4 parameters. This is because there is a weight vector, which contains 3 weight parameters, and a single bias parameter, resulting in 4 total parameters.

Question 2: Recall that we were hoping to 'predict' the probability of a patient being positive with COVID-19. Does the output make sense? If not, elaborate on how you could fix it.

Ans: Due to the fact that we are attempting to predict the **probability** of a patient being positive with COVID-19, the output *~58.1* does not make complete sense. It is common place to have probabilities reported in the range of 0-1, where .2 would correspond to a 20% probability of an event occurring. In order to introduce this standard notation of probability, one could apply the Sigmoid function, which takes any input, and will map that input between the range of 0-1.

Perceptron with Multiple Outputs

The perceptron model above has only one output. However, in most applications, we need multiple outputs. For example, in a classification problem, we would expect the model to output a vector $\mathbf{y}$, where each y_i represents the probability of a sample belonging to a particular class i. The figure below shows a schematic view of a multiple output feed-forward neural network. Each input value x_i is multiplied by a weight factor W_{ij}, where W_{ij} denotes a connection weight between the input node x_i and the output node y_j. The weighted sum is added to the bias and then passed through an activation function to obtain the output, y_j.



Given an input $\mathbf{x} \in \mathbb{R}^n$ this can be modeled as:

$$y_j(\mathbf{x};\Theta) = f\left(\left(\sum_{i=1}^{n} x_i W_{ij}\right) + b_j\right) = f(\mathbf{w}_j^\intercal \mathbf{x} + b_j)\,,$$

where the parameter set here is $\Theta = \{\mathbf{W} \in \mathbb{R}^{n\times m}, \mathbf{b} \in \mathbb{R}^m\}$ and $\mathbf{w}_j$ denotes the j^{th} column of $\mathbf{W}$.

Implementation

Let $\mathbf{x} = \begin{bmatrix} 98.6 \\ 95 \\ 0 \\ 1 \end{bmatrix}$. Let the output vector $\mathbf{y} \in \mathbb{R}^3$, i.e. consisting of 3 outputs. Let the weights associated with each output node y_i be $\mathbf{w_1} = \begin{bmatrix} 0.03 \\ 0.55 \\ 0.88 \\ 0.73 \end{bmatrix}$, $\mathbf{w_2} = \begin{bmatrix} 0.48 \\ 0.31 \\ 0.28 \\ -0.9 \end{bmatrix}$, $\mathbf{w_3} = \begin{bmatrix} 0.77 \\ 0.54 \\ 0.37 \\ 0.44 \end{bmatrix}$. Let the bias vector be $\mathbf{b} = \begin{bmatrix} 2.9 \\ 6.1 \\ 3.3 \end{bmatrix}$. Note that a single bias is associated with each output node y_i.

Given the above inputs write the code to print the output vector $\mathbf{y}$. Use a Python variables for each mathematical variable and use Python lists for vectors.

In [3]:
```
# TODO: Your code here
x = [98.6, 95, 0 , 1]
w1 = [0.03, 0.55, 0.88, 0.73]
w2 = [0.48, 0.31, 0.28, -0.9]
w3 = [0.77, 0.54, 0.37, 0.44]
b = [2.9, 6.1, 3.3]

def ReLU(x):
    return x * (x > 0)
```

In [4]:
```
y1 = ReLU(sum([x*w1 for x,w1 in zip(x,w1)]) + b[0])
y2 = ReLU(sum([x*w2 for x,w2 in zip(x,w2)]) + b[1])
y3 = ReLU(sum([x*w3 for x,w3 in zip(x,w3)]) + b[2])

print(f'Output for node y1 is {y1}')
print(f'Output for node y2 is {y2}')
print(f'Output for node y3 is {y3}')
```

```
Output for node y1 is 58.838
Output for node y2 is 81.97799999999998
Output for node y3 is 130.96200000000002
```

Now that you understand how to do basic computations with a simple perceptron model manually, we will proceed to implement the same model above using matrix-vector operations utilizing PyTorch functions. Organizing the computations in matrix-vector format notation makes it simpler to understand and implement neural network models.

Write the code to create the same output vector $\mathbf{y}$ as above by expressing the above computations as matrix-vector multiplications and summation with a bias vector using code vectorization in PyTorch. You should get the same output as above, up to floating-point errors. Again use Python variables for the mathematical variables, but use PyTorch arrays for vectors and matrices.

In [5]:
```
import torch
import numpy as np
```

In [6]:
```
x = torch.tensor([98.6, 95, 0 , 1])
W = torch.tensor([[0.03, 0.55, 0.88, 0.73],
                  [0.48, 0.31, 0.28, -0.9],
                  [0.77, 0.54, 0.37, 0.44]])
b = torch.tensor([2.9, 6.1, 3.3])

def ReLU(x):
    return x * (x > 0)
```

In [7]:
```
y = ReLU((x @ W.T) + b)
print(f'Output for node y1 is {y[0]}')
print(f'Output for node y2 is {y[1]}')
print(f'Output for node y3 is {y[2]}')
```

```
Output for node y1 is 58.8380012512207
Output for node y2 is 81.97799682617188
Output for node y3 is 130.96200561523438
```

Question 3: Explain what each of the dimensions of the matrix of weights $\mathbf{W}$ and the vector of biases $\mathbf{b}$ represent?

Ans: The dimensions of the weights matrix is 3X4, this is because there are 3 output nodes, each of which multiply the 4 input nodes by a unique weight vector. This is such that the first row of weights, corresponds to the weights for the first output node, the second row corresponds to weights for the second output node, ect. The bias vector has a dimension of 1X3 because there are 3 output nodes, and each one has a bias associated with it.

Question 4: What is the total number of parameters for this model?

Ans: This model has a weight matrix with a dimension of 3X4, which means it has 12 weight parameters. In addition, it has a bias vector containing 3 bias parameters. As such, this model has a total of 15 parameters.

More Layers

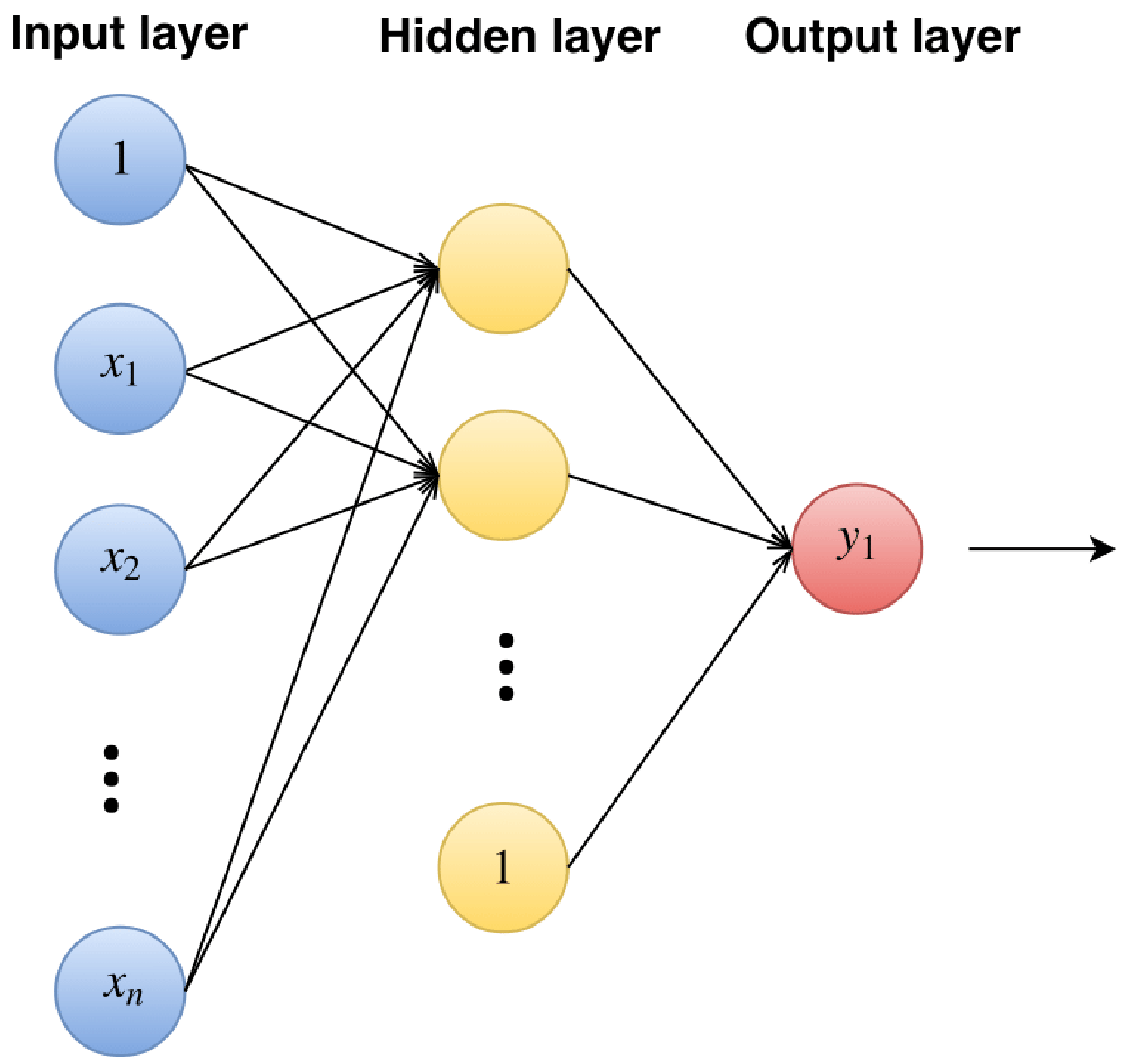
A single-layer perceptron network still represents a linear classifier, even if we were to use nonlinear activation functions. This limitation can be overcome by multi-layer neural networks in combination with nonlinear activation functions, which introduce one or more 'hidden' layers between the input and output layers. Multi-layer neural networks are composed of several simple artificial neurons such that the output of one acts as the input of another. A multi-layer neural network can be represented by a composition function. For a two-layer network with only one output, the composition function can be written as

$$y_j(\mathbf{x};\Theta) = f^{(2)}\left(\sum_{k=1}^{h} W_{kj}^{(2)} * f^{(1)}\left(\left(\sum_{i=1}^{n} W_{ik}^{(1)} * x_i\right) + b_k^{(1)}\right) + b_j^{(2)}\right)$$

where h is the number of units in the hidden layer and the set of unknown parameters is $\Theta = \{\mathbf{W}^{(1)} \in R^{n\times h}, \mathbf{W}^{(2)} \in R^{h\times 1}\}$. In general, for $L-1$ hidden layers the composition function, omitting the bias terms, can be written as

$$y_j(\mathbf{x};\Theta) = f^{(L)}\left(\sum_{k} W_{kj}^{L} * f^{L-1}\left(\sum_{l} W_{lk}^{L-1} * f^{L-2}\left(\cdots f^{1}\left(\sum_{i} W_{iz}^{1} * x_i\right)\right)\right)\right)$$

The figure below illustrates a feed-forward neural network composed of an input layer, a hidden layer, and an output layer. In this illustration, the multi-layer neural network has one input layer and one output unit. In most models, the number of hidden layers and output units is more than one.



We will now see how to add an additional layer to our model and then how to generalize to any number of layers.

In [8]:

```
# To add another layer we need another set of weights and biases.

W_2 = torch.tensor([[-0.3, 0.66, 0.98],
                   [0.58, -0.4, 0.38],
                   [0.87, 0.69, -0.4]])

b_2 = torch.tensor([3.9, 8.2, 0.8])
```

In [9]:

```
# TODO: Write the code to print the output of a 2-layer feed-forward network
# y, as input to the second layer
y_1 = ReLU((y @ W_2.T) + b_2)

print(f'Output for node y1 is {y_1}')
```

```
Output for node y1 is tensor([168.6969,  59.3004,  56.1691])
```

Question 5: Explain the dimensions of the weight matrix for the second layer with respect to the dimensions of the previous layer and the number of artificial neurons in the second layer. or Why are the dimensions of the weight matrix of the second layer 3x3?

Ans: The reason that the dimensions of the weight matrix for the second layer are 3X3 is because the output from the second layer is a 1X3 vector. As such, the weight matrix must be of dimension 3XN in order for matrix multiplication to be allowed and to properly receive the input from the 3 nodes. Also, due to the fact that 3 output nodes is still desired, the second layer weight matrix's dimensions must be 3X**3**.

Layers to Objects

Now, we have a feed-forward model (with an input layer, one hidden layer, and one output layer with 3 outputs) capable of processing a batch of data. It would be cumbersome and redundant if we had to keep writing the same code for hundreds of layers. So, to make our code more modular, easier to manage, and less redundant we will represent layers using an object-oriented programming paradigm. Let's define classes for representing our layers.

All layer objects should have an `output` instance attribute. Use good object-oriented practices to avoid code duplication. To initialize an instance attribute in Python, write `self.attribute_name = attribute_value` in the initializer (`__init__` method). Don't mention the variable at the top of the class as we would usually do in Java -- this is how you define static attributes in Python.

Rather than each layer taking PyTorch arrays as inputs, it should take `Layer` s as inputs, with each layer having its own name. For example, if your network would take $\mathbf{x}$, $\mathbf{W}$, and $\mathbf{b}$ as inputs, you should have attributes `self.x` , `self.W` , and `self.b` . Then, when you need the values of these inputs, go back and read the output of the previous layer. For example, if your layer needs the value of $\mathbf{W}$, you could read `self.W.output` to get it.

Two more Python OO hints: (1) `class MyClass1(MyClass2)` is not a constructor call. It is specifying the inheritance relationship. The Java equivalent is `class MyClass1 extends MyClass2` . So you don't want to add arguments on this line. An easy mistake to make. (2) You must use `self.` every time you access an instance variable in Python. This is how the language was designed.

In [10]:

```
# TODO: Complete the following classes.

class Layer:
    def __init__(self, output_shape):
        """
        TODO: Initialize instance attributes here.
        :param output_shape (tuple): the shape of the output array.  When thi
        it gives the number of output neurons. When this is an array, it give
        of the array of output neurons.
        """
        if not isinstance(output_shape, tuple):
            output_shape = (output_shape,)

        self.output_shape = output_shape




class Input(Layer):
    def __init__(self, output_shape):
        """
        TODO: Accept any arguments specific to this child class.
        """
        Layer.__init__(self, output_shape) # TODO: Pass along any arguments t

    def set(self,value):
        """
        TODO: set the `output` of this array to have value `value`.
        Raise an error if the size of value is unexpected. An `assert()` is f
        """
        assert self.output_shape == value.shape
        self.output = value

    def forward(self):
        """This layer's values do not change during forward propagation."""
        pass


class Linear(Layer):
    def __init__(self, x, W, b):
        """
        TODO: Accept any arguments specific to this child class.

        Raise an error if any of the argument's size do not match as you woul
        """
        Layer.__init__(self, b.output_shape) # TODO: Pass along any arguments
        self.x = x
        self.W = W
        self.b = b

    def ReLU(x):
        return x * (x > 0)

    def forward(self):
        """
        TODO: Set this layer's output based on the outputs of the layers that
        """
        self.output = ReLU((self.x.output @ self.W.output.T) + self.b.output)
```

In [11]:

```
# This example uses your classes to test the output of the entire network.
#
# You may change this example to match your own implementation if you wish.
x = [98.6, 95, 0 , 1]
W = [[0.03, 0.55, 0.88, 0.73],
    [0.48, 0.31, 0.28, -0.9],
    [0.77, 0.54, 0.37, 0.44]]
b = [2.9, 6.1, 3.3]

x_layer = Input(4)
x_layer.set(np.array(x))
W_layer = Input((3, 4))
W_layer.set(np.array(W))
b_layer = Input(3)
b_layer.set(np.array(b))
linear_layer = Linear(x_layer, W_layer, b_layer)

linear_layer.forward()
print('\n output of hidden layer \n', linear_layer.output)
```

```

 output of hidden layer
 [ 58.838  81.978 130.962]
```

This concludes this lab... except for the two **required** parting questions:

Question 6: Summarize what you learned during this lab.

Ans: In this lab I learned about the basics of creating a Neural Network from scratch. This began by first understanding the fundamental processing unit of a feed forward network, a perceptron. I learned that the equation needed to compute the output value of a perceptron is:

$$y(\mathbf{x};\Theta) = f\left(\left(\sum_{i=1}^{n} x_i w_i\right) + b\right) = f(\mathbf{w}^\intercal \mathbf{x} + b)\,,$$

Which once understood makes coding a simple feedforward perceptron very easy if simple python lists are used for the variables. Through this verbose exercise I came to understand that all a feedforward perceptron does, or even a single neuron in a larger network does, is multiply each of its inputs by the corresponding weight, add the corresponding bias, sum the biased output together, and finally apply an activation function to that sum. With this verbose understanding, it made me appreciate matrix and vector libraries such as numpy or PyTorch much more. Another lesson I gained from this lab is how output dimensions are changed layer by layer using simple matrix multiplication. In addition, I learned that in order to string multiple feedforward layers together, one must nest multiple instances of the above function inside one another, where the previous layer in the sequence replaces the x variable with a complete function of its own. Lastly, I learned that using object oriented programming practices in order to code a multi-layer neural network reduces a great deal of redundance, makes the final result more intuitive, and is overall more robust than hand coding each layer as either a numpy array or a PyTorch Tensor.

Question 7: Describe what you liked about this lab *or* what could be improved. (Required.)

Ans:

I enjoyed that thoughtfulness that was put into this lab write up. It was a very well written notebook, and following along with this tutorial/lab was extremely easy because of this. I also appreciated the included mathematical equations as well as the pictures because at times the text was difficult to grasp but the included visuals would often clean up any misunderstandings. In addition, I liked the progression of difficulty of this lab, it made it so that lessons were taught at each step that could be applied to later questions/tasks. One of the only things that I think could be improved is the documentation and instructions given for the Layer, Input, and Linear classes. In hindsight it was not a very difficult implementation, but it was difficult to understand how the whole thing was intended to work together, and I spent a significant amount of time just trying to understand what should go where due to the vagueness of directions. I think that section could benefit from more hints, or even give more examples to run and the correct answers so that students know what to aim for.