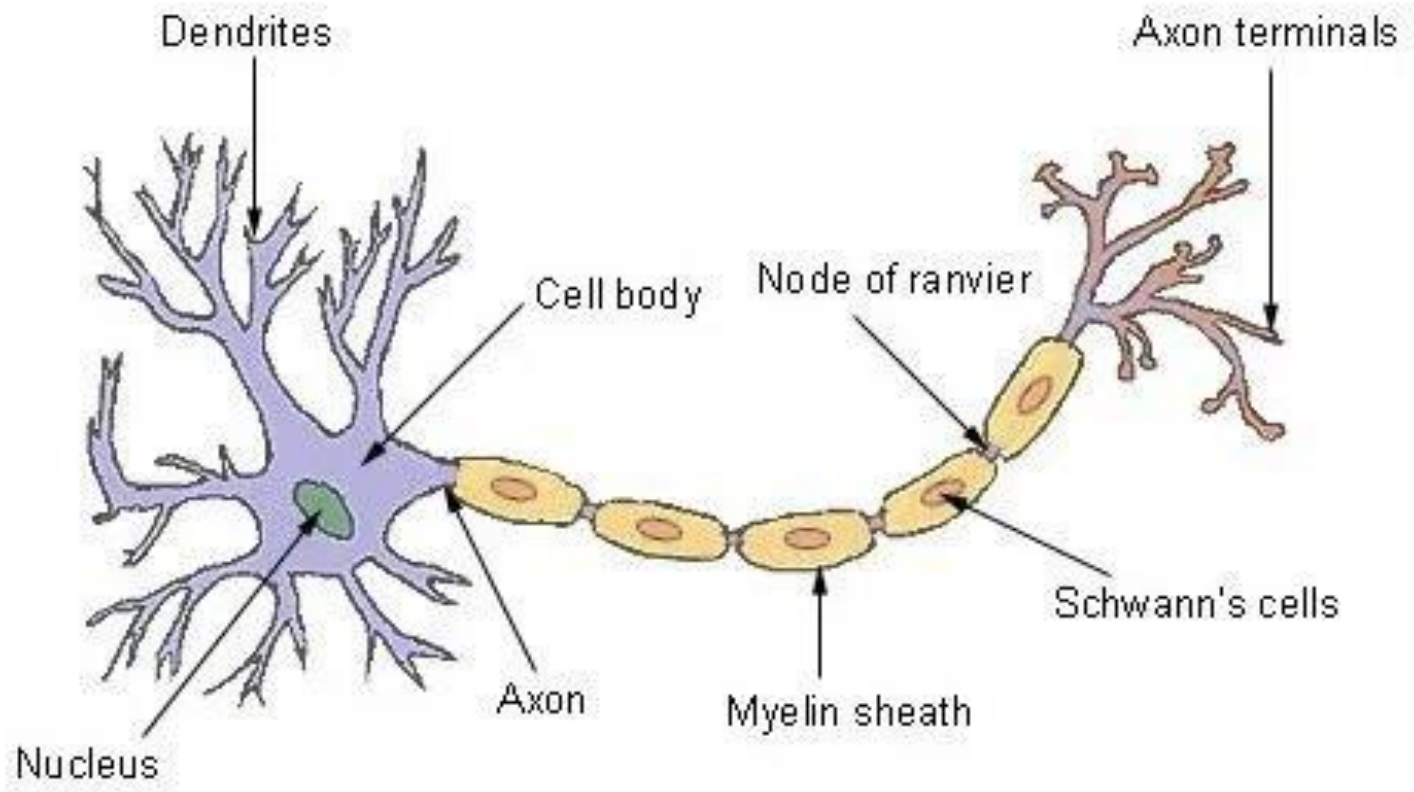
NEURAL NETWORK

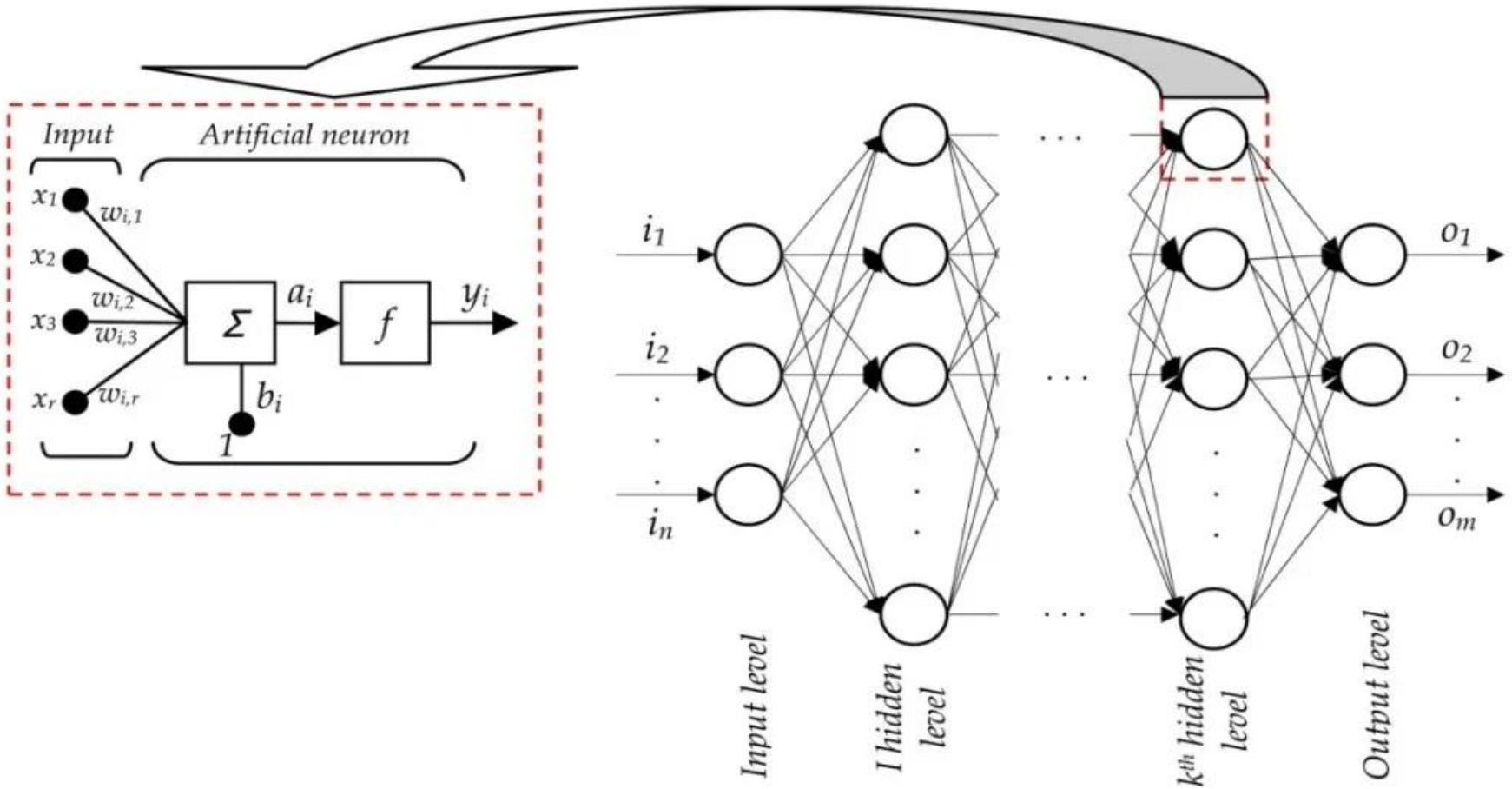
ALOK

The major components are:

- **Dendrites-** It takes input from other neurons in form of an electrical impulse
- **Cell Body**– It generate inferences from those inputs and decide what action to take
- **Axon terminals**– It transmit outputs in form of electrical impulse



1. **Input Layer:** The training observations are fed through these neurons
2. **Hidden Layers:** These are the intermediate layers between input and output which help the Neural Network learn the complicated relationships involved in data.
3. **Output Layer:** The final output is extracted from previous two layers. For Example: In case of a classification problem with 5 classes, the output later will have 5 neurons.



How a Single Neuron works

The different components are:

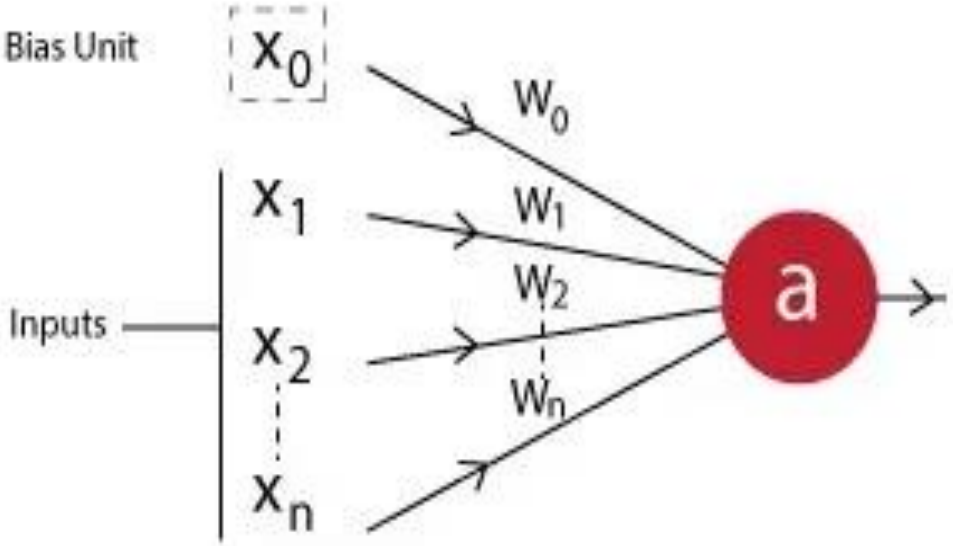
1.x_1, x_2,…, x_N: Inputs to the neuron. These can either be the actual observations from input layer or an intermediate value from one of the hidden layers.

2.x_0: Bias unit. This is a constant value added to the input of the activation function. It works similar to an intercept term and typically has +1 value.

3.w_0,w_1, w_2,…,w_N: Weights on each input. Note that even bias unit has a weight.

4.a: Output of the neuron which is calculated as:

Diagram 1: Single NN Working



$$a = f(\sum_{i=0}^{N} w_i x_i)$$

Here **f** is known an **activation function**. This makes a Neural Network extremely flexible and imparts the capability to estimate complex non-linear relationships in data. It can be a gaussian function, logistic function, hyperbolic function or even a linear function in simple cases.

Lets implement 3 fundamental functions – **OR, AND, NOT** using Neural Networks.

This will help us understand how they work. You can assume these to be like a classification problem where we'll predict the output (0 or 1) for different combination of inputs.

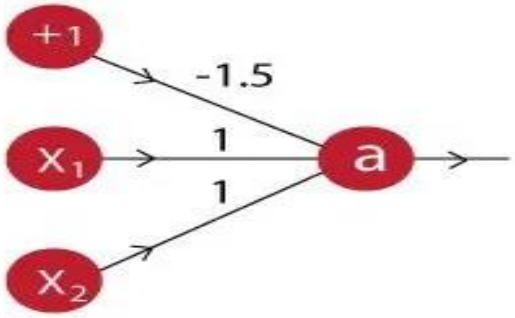
We will model these like linear classifiers with the following activation function:

$$f(x) = \begin{cases} 0, & for\ x < 0 \\ 1, & for\ x \geq 0 \end{cases}$$

Example 1: AND

The AND function can be implemented as:

Diagram 2: AND



The output of this neuron is: **a = f(-1.5 + x1 + x2)**

The truth table for this implementation is:

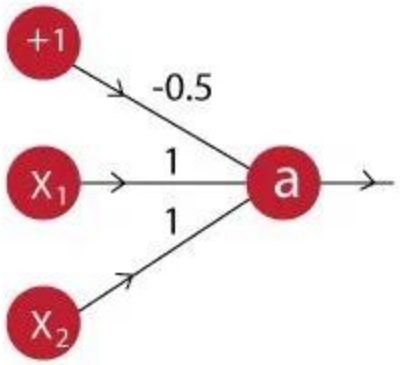
X1	X2	X1 AND X2	(-1.5+X1+X2)	a
0	0	0	-1.5	0
0	1	0	-0.5	0
1	0	0	-0.5	0
1	1	1	0.5	1

Here we can see that the AND function is successfully implemented. Column 'a' complies with 'X1 AND X2'. Note that here the bias unit weight is -1.5. But it's not a fixed value. Intuitively, we can understand it as anything which makes the total value positive only when both x_1 and x_2 are positive. So any value between (-1,-2) would work.

Example 2: OR

The OR function can be implemented as:

Diagram 3: OR



The output of this neuron is:

a = f(-0.5 + x1 + x2)

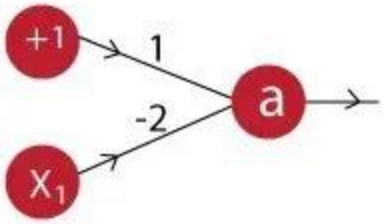
The truth table for this implementation is:

X1	X2	X1 OR X2	(-0.5+X1+X2)	a
0	0	0	-0.5	0
0	1	1	0.5	1
1	0	1	0.5	1
1	1	1	1.5	1

Column ‘a’ complies with ‘X1 OR X2’. We can see that, just by changing the bias unit weight, we can implement an OR function. This is very similar to the one above. Intuitively, you can understand that here, the bias unit is such that the weighted sum will be positive if any of x1 or x2 becomes positive.

- Example 3: NOT
- Just like the previous cases, the NOT function can be implemented as:

Diagram 4: NOT



The output of this neuron is: **a = f(1 – 2*x1)**

The truth table for this implementation is:

X1	NOT X1	(1-2*X1)	a
0	1	1	1
1	0	-1	0

- multi-layer networks

After understanding the working of a single neuron, lets try to understand how a Neural Network can model complex relations using multiple layers.
To understand this further, we will take the example of an **XNOR function**. Just a recap, the truth table of an XNOR function looks like:

X1	X2	X1 XNOR X2
0	0	1
0	1	0
1	0	0
1	1	1

Here we can see that the output is 1 when both inputs are same, otherwise 0. This sort of a relationship **cannot be modeled using a single neuron**. (Don't believe me? Give it a try!) Thus we will use a multi-layer network.
The idea behind using multiple layers is that complex relations can be broken into simpler functions and combined.

- In order for our model to learn, we need to define what is good. Actually, we will define what is bad and try to minimize it.
- We will call the “badness” — error or cost (hence, the cost function). It represents how far off of the true result our model is at some point during training.
- We would love that error to be 0 for all possible inputs

The cost function that we are going to use is called **Cross-Entropy**. It is defined as:

$$H_{y'}(y) = -\sum_i y'_i \log(y_i)$$

Where y is the predicted distribution for our alcohol consumption and y’is the ground truth.

- TensorFlow has a little helper function with the sweet little name softmax_cross_entropy_with_logits. It uses softmax as activation function for our output layer and Cross-Entropy as error function.
- cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=predictions, labels=y))

- ADAM
- Now, for the actual workhorse — Adam (nope, not the from the Bible — although, that would've been fun). Adam is a type of gradient descent optimization algorithm which essentially tries as hard as he can to find proper weights and biases for our network via minimizing the cost function that we specified above. It is well beyond the scope of this post to describe Adam in details, but you can find all the necessary information over here — with tons of nice pictures!

- Using Adam in TensorFlow is quite easy, we just have to specify learning rate (you can fiddle with that one) and pass the cost function we defined above:
- optimizer = tf.train.AdamOptimizer(learning_rate=0.0001).minimize(cost)

- Our model is created by just calling our helper function with the proper arguments:
- predictions = multilayer_perceptron(x, weights, biases, keep_prob)