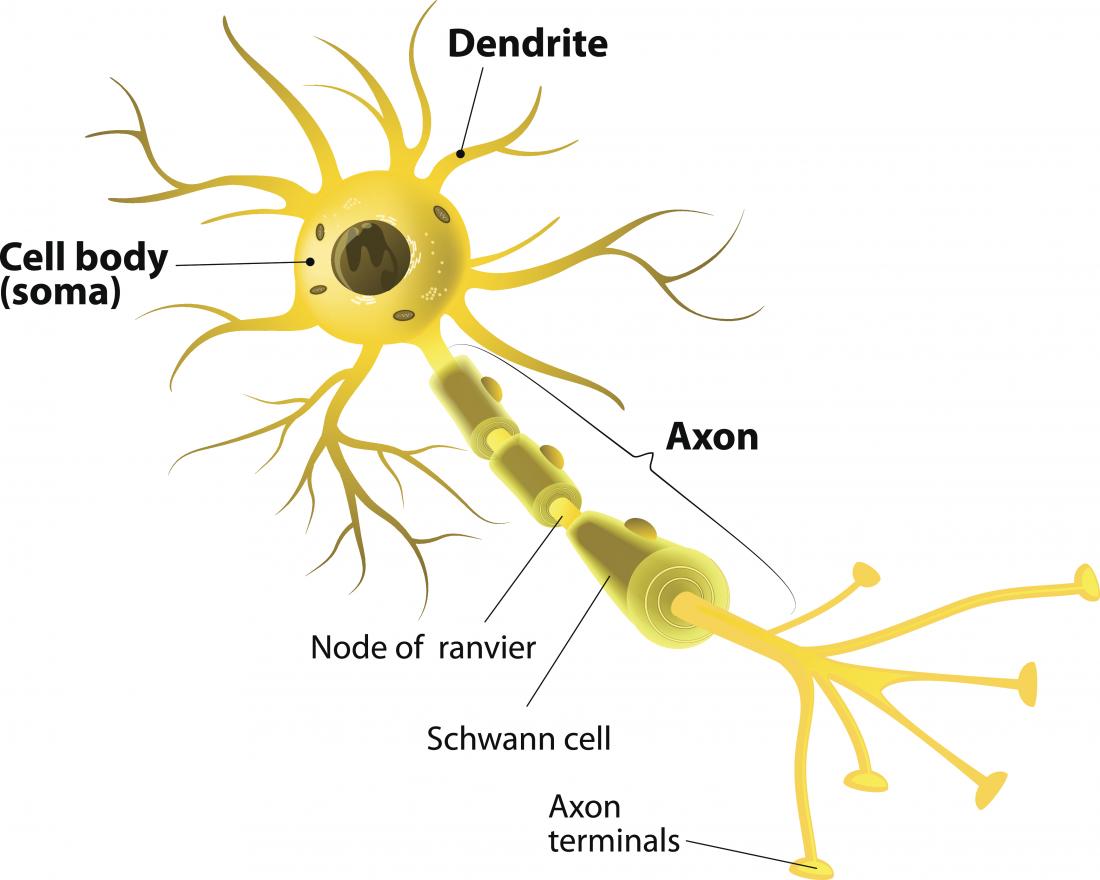
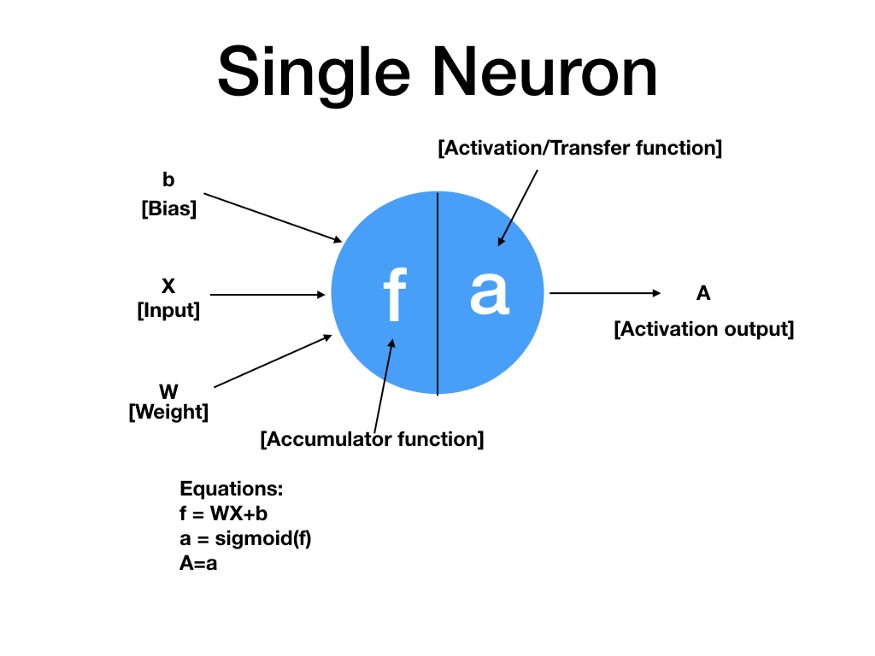
**What is a neural network neuron?**

A neuron in an artificial neural network is an elementary block that mimics the functionality of human neurons. However, with stark difference that neurons/perceptron are enabled with memory storage for a very short duration of time and serve as an elementary computational block unlike the human neurons that allow for memory storage, computation and signalling. Also, unlike Human neuron where inputs may be termed as dendrites and outputs are via Axioms, the input and output lines for perceptron’s are termed as weights.



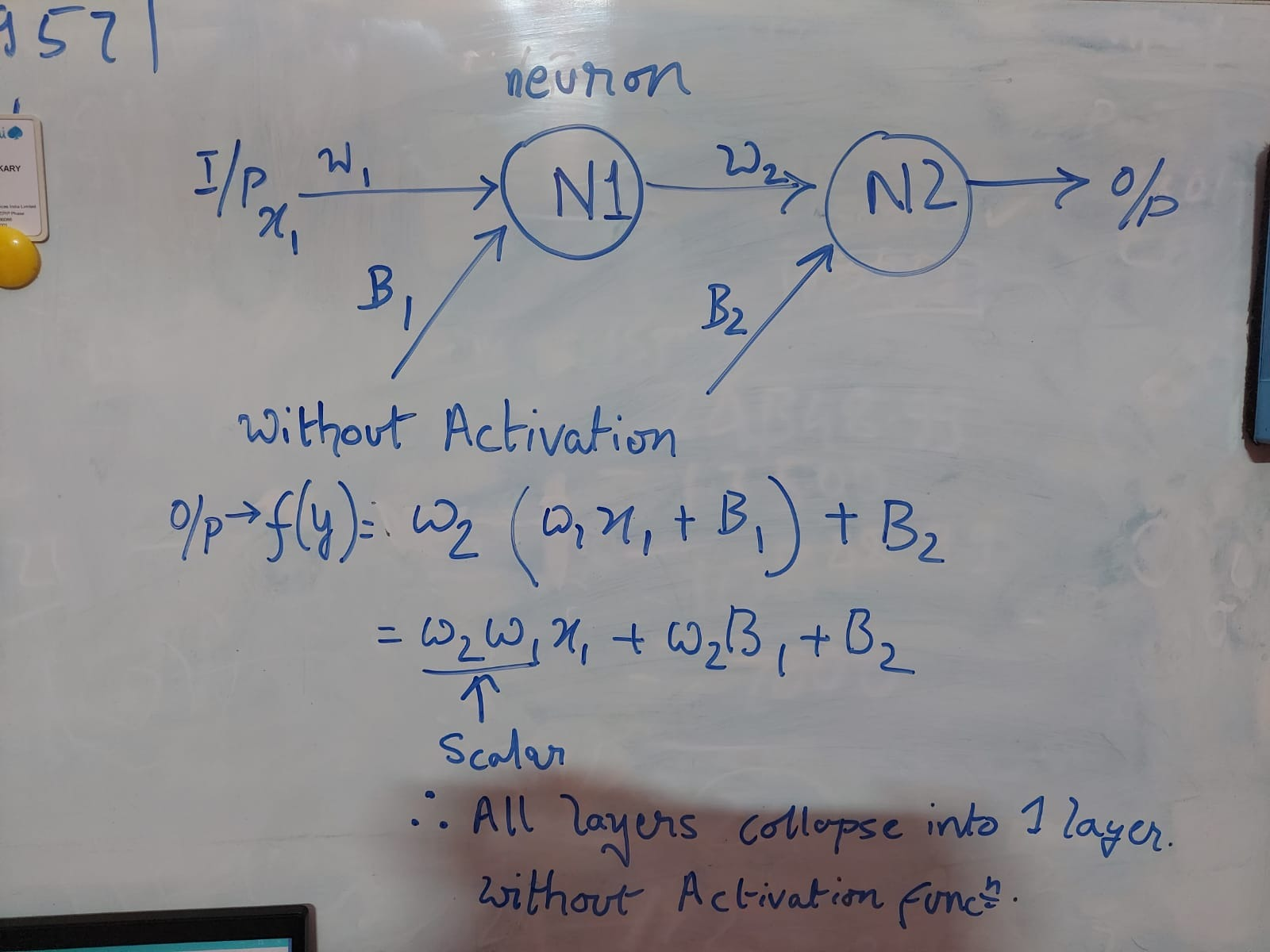


Each neuron is fed with an input along with associated weight and bias. A neuron has two functions:

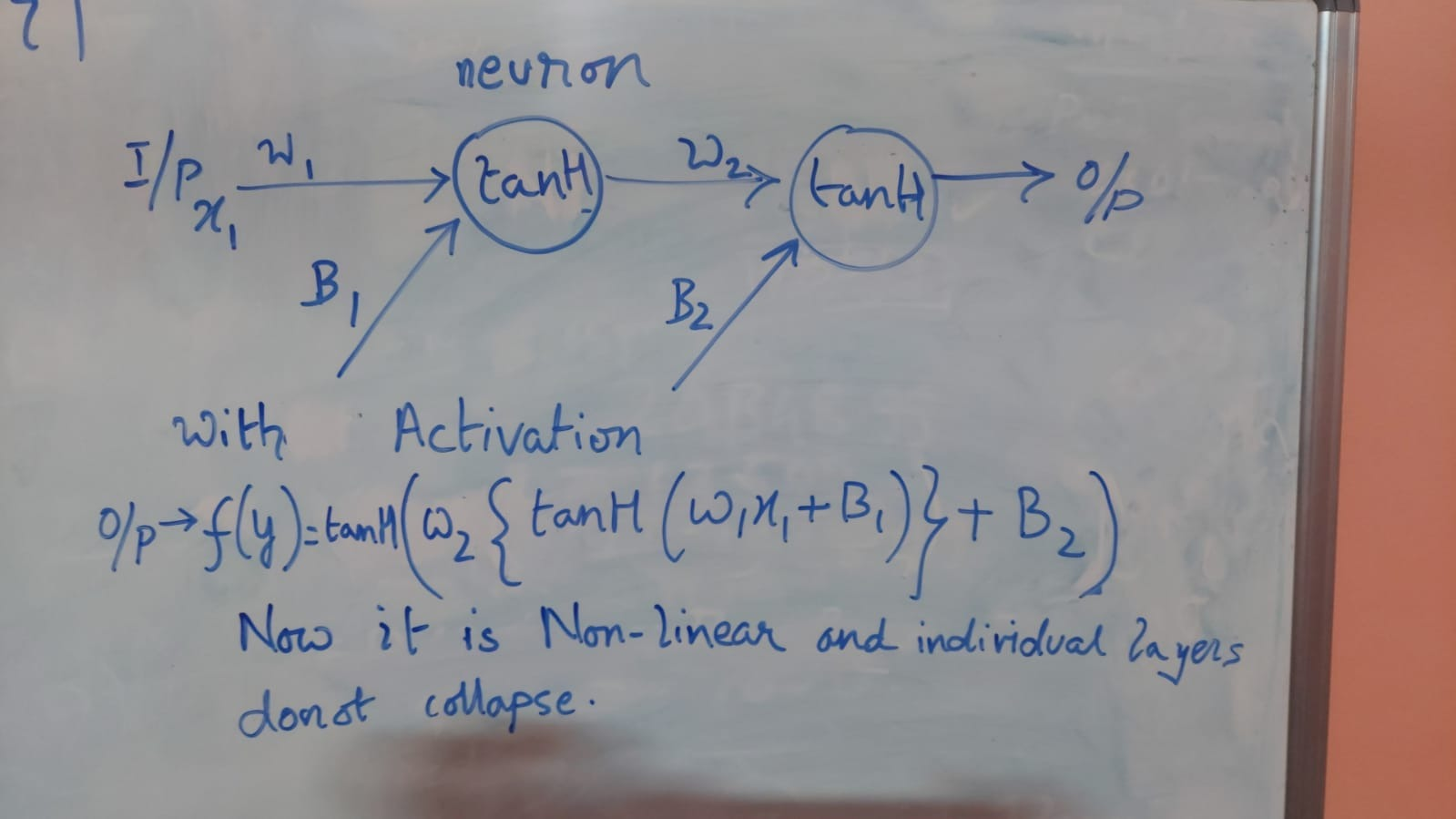
1) Accumulator function: It essentially is the weighted sum of input along with a bias added to it.  
2) Activation function: Activation functions are non-linear function. And as the name suggests is a function to decide whether output of a node will be actively participating in the overall output of the model or not.

Now an Activation function plays a very critical task of converting the Acumulator function into non-linear space so that the layers might not collapse into 1 layer .

So essentially without activation function :



With Activation Function :



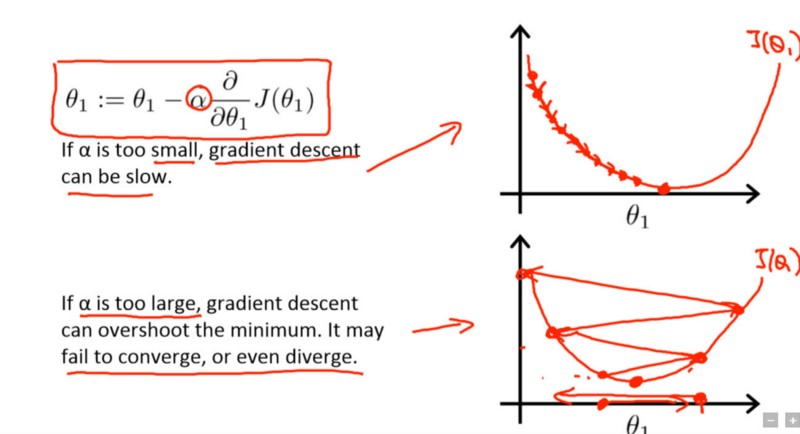
**What is the use of the learning rate?**

First off, what is a learning rate?

Learning rate is a hyper-parameter that controls how much we are adjusting the weights of our network with respect the loss gradient. The lower the value, the slower we travel along the downward slope. While this might be a good idea (using a low learning rate) in terms of making sure that we do not miss any local minima, it could also mean that we’ll be taking a long time to converge — especially if we get stuck on a plateau region.

The following formula shows the relationship.

New Weight = Old weight – Learning Rate \* Gradient.



There are is reason also another reason why the LR should be small:

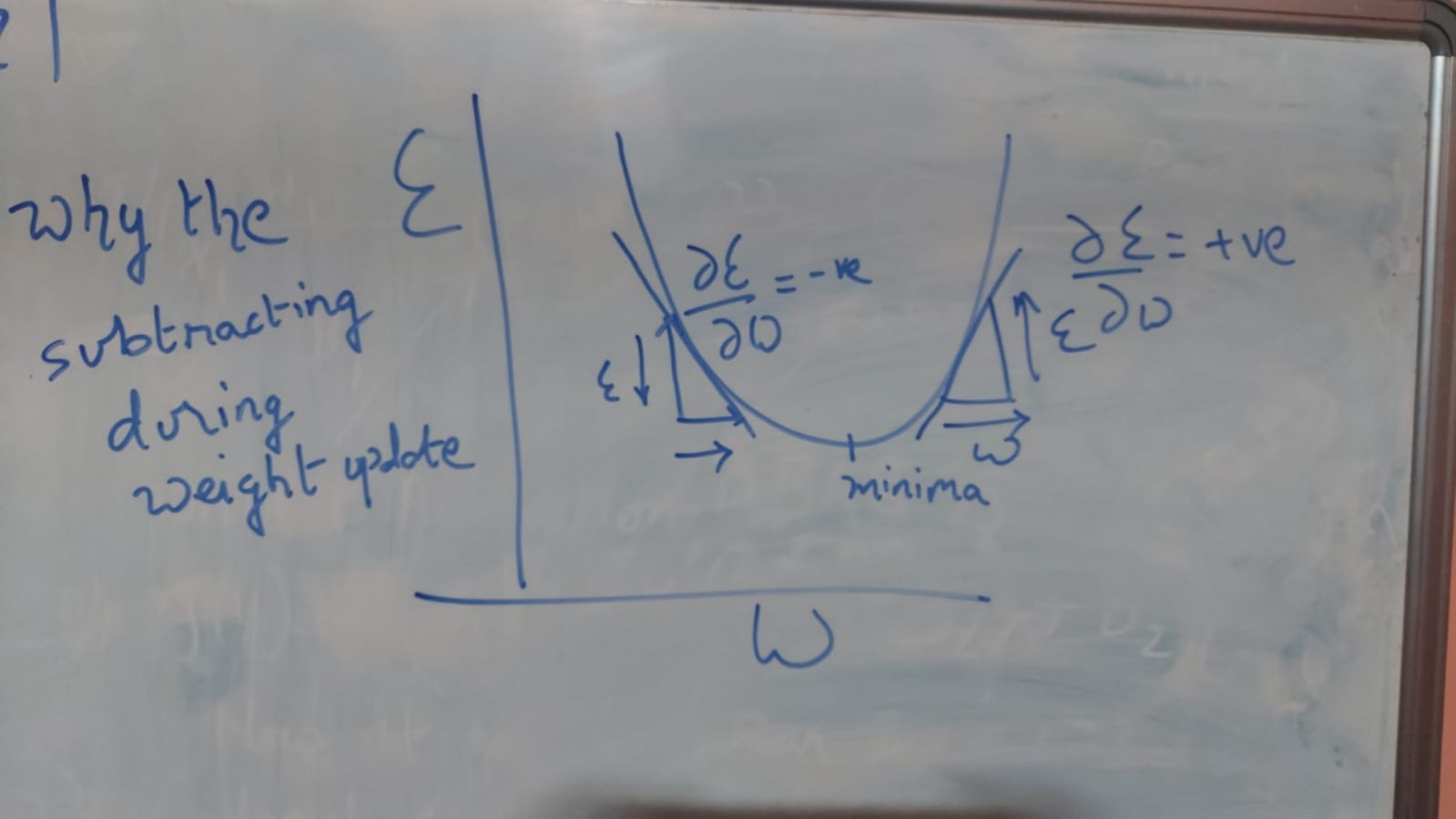
1. So that the update is so small that we can consider weights updation on every layer to be independent of the next as the impact would be too small to even notice.

As for why it is subtractive and not an additive property is so that we reduce the loss at every step and do not overshoot the minima.

So in case the gradient is -ve we are moving in the right direction and the weight gets updated by addition.

If in case the gradient is +ve then we reduce the weight to reduce the error or loss.

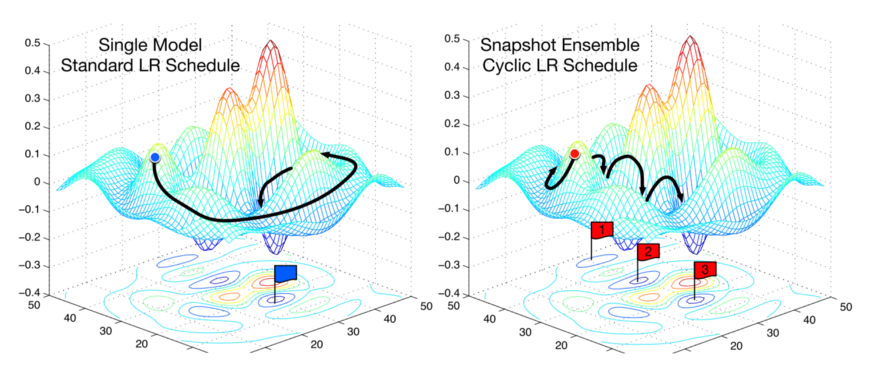
Diagram below illustrates it.



Typically learning rates are configured naively at random by the user. At best, the user would leverage on past experiences (or other types of learning material) to gain the intuition on what is the best value to use in setting learning rates.

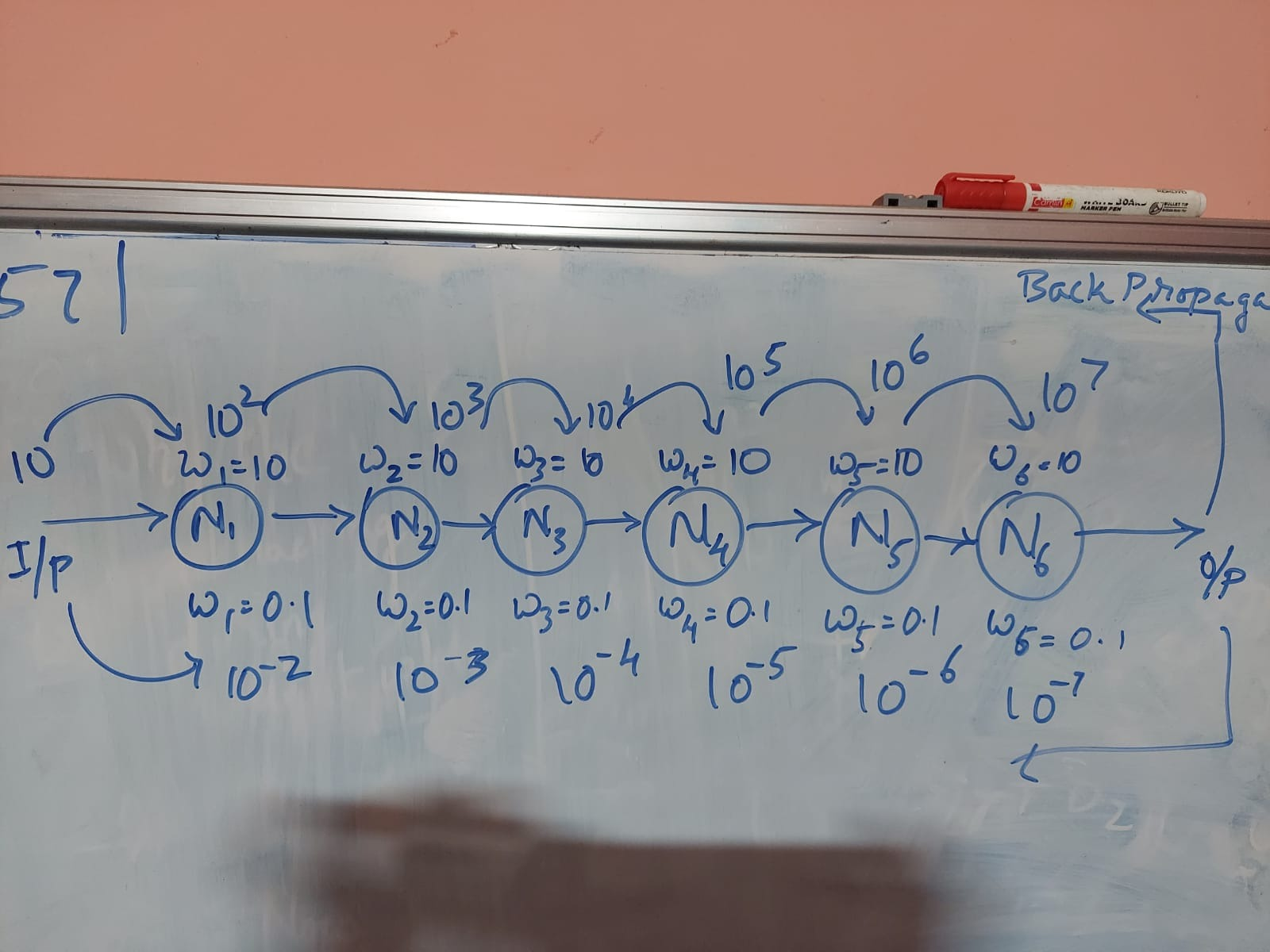
Over the years we have however come up with different approaches to balance the speed and accuracy of convergence to optimal minima using different learning rate principles delailed in how to achieve super convergence .

1. Learning rate scheduler
2. One cycle learning rate policy
3. Cyclic learning rate



**Why do we go with Neural Network Weight Initialization ?**

Weight Initialization is important to deal with Vanishing and exploding gradient as the back propagation otherwise does not make sense as it is either too large or too small of an update during Back propagation. Hence it is advisable to keep it as close to 0 as possible hence we initialize weights so that weights at individual levels does not explode or vanish.



As seen in the image above there are two extremes possible for instance 10^7 and 10^-7 so even we send weight update in back propagation it will not make sense for each layer.

## Weight Initialization for Sigmoid and Tanh

The current standard approach for initialization of the weights of neural network layers and nodes that use the Sigmoid or TanH activation function is called “glorot” or “xavier” initialization.

It is named for [Xavier Glorot](https://www.linkedin.com/in/xglorot/), currently a research scientist at Google DeepMind, and was described in the 2010 paper by Xavier and Yoshua Bengio titled “[Understanding The Difficulty Of Training Deep Feedforward Neural Networks](http://proceedings.mlr.press/v9/glorot10a.html).”

There are two versions of this weight initialization method, which we will refer to as “xavier” and “normalized xavier.”

### Xavier Weight Initialization

The xavier initialization method is calculated as a random number with a uniform probability distribution (U) between the range -(1/sqrt(n)) and 1/sqrt(n), where n is the number of inputs to the node.

* weight = U [-(1/sqrt(n)), 1/sqrt(n)]

We can implement this directly in Python.

The example below assumes 10 inputs to a node, then calculates the lower and upper bounds of the range and calculates 1,000 initial weight values that could be used for the nodes in a layer or a network that uses the sigmoid or tanh activation function.

After calculating the weights, the lower and upper bounds are printed as are the min, max, mean, and standard deviation of the generated weights.

## Weight Initialization for ReLU

The “xavier” weight initialization was found to have problems when used to initialize networks that use the rectified linear ([ReLU](https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/)) activation function.

As such, a modified version of the approach was developed specifically for nodes and layers that use ReLU activation, popular in the hidden layers of most multilayer Perceptron and convolutional neural network models.

The current standard approach for initialization of the weights of neural network layers and nodes that use the rectified linear (ReLU) activation function is called “he” initialization.

It is named for [Kaiming He](https://www.linkedin.com/in/kaiming-he-90664838/), currently a research scientist at Facebook, and was described in the 2015 paper by Kaiming He, et al. titled “[Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification](https://arxiv.org/abs/1502.01852).”

**HE Weight Initialization**

The he initialization method is calculated as a random number with a Gaussian probability distribution (G) with a mean of 0.0 and a standard deviation of sqrt(2/n), where n is the number of inputs to the node.

* weight = G (0.0, sqrt(2/n))

We can implement this directly in Python.

The example below assumes 10 inputs to a node, then calculates the standard deviation of the Gaussian distribution and calculates 1,000 initial weight values that could be used for the nodes in a layer or a network that uses the ReLU activation function.

After calculating the weights, the calculated standard deviation is printed as are the min, max, mean, and standard deviation of the generated weights.