**Mathematics of Neurons and models:**

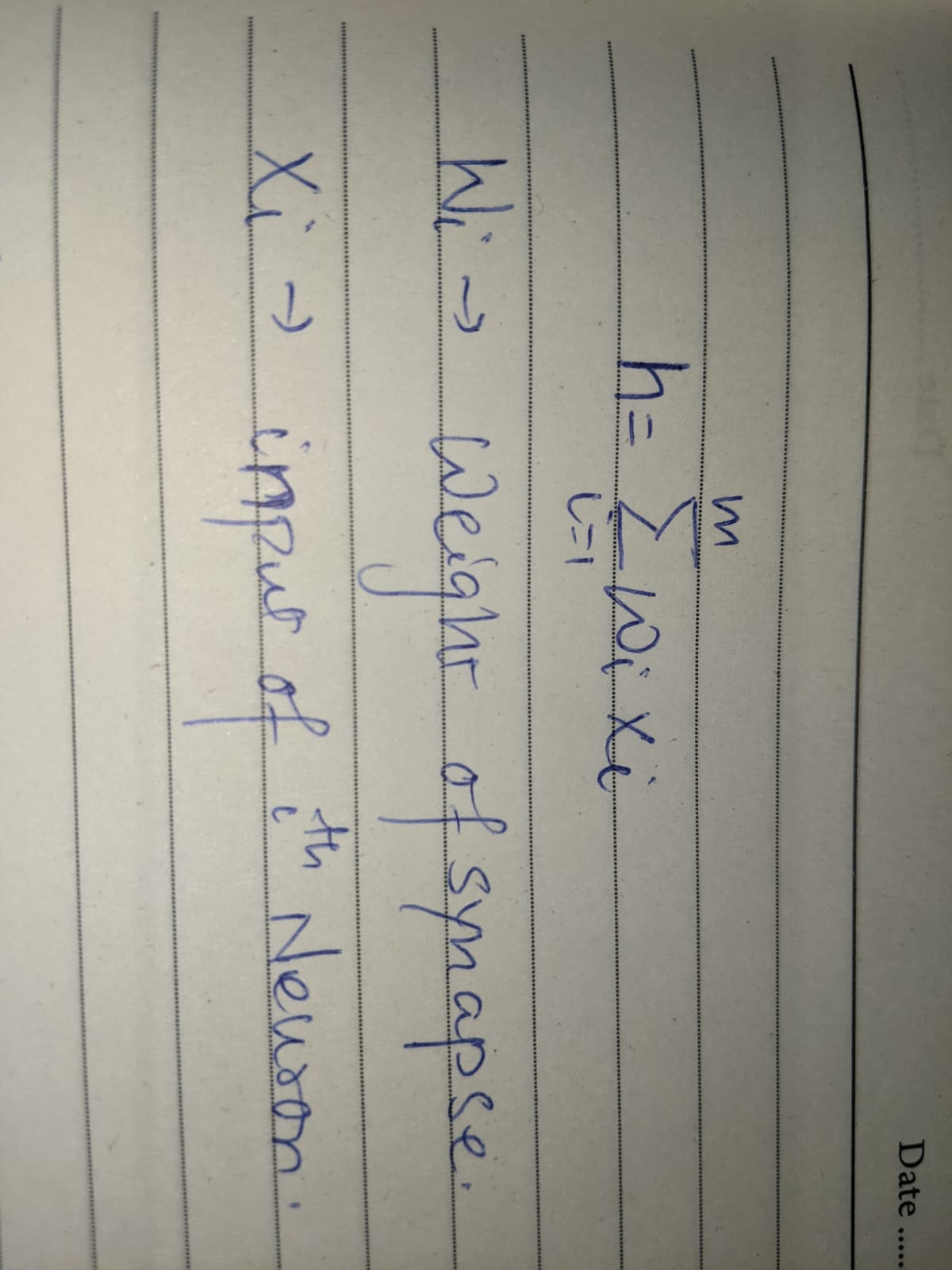
An important task for learning deep learning is to learn math & behind the scenes statistics! Before you can actually start to write code. In this blog we will discuss and learn about the math of neural nets from 60’s & 70’s.

The idea of neurons traces back to **1940’s**, well the neurons did exist but only **ON PAPER.** Because there was not good computers to solve even basic problems with such math heavy tasks. **Mc Culloch & Pitts** presented the idea for a neuron, but only math existed, no computer or algorithm to train a model.

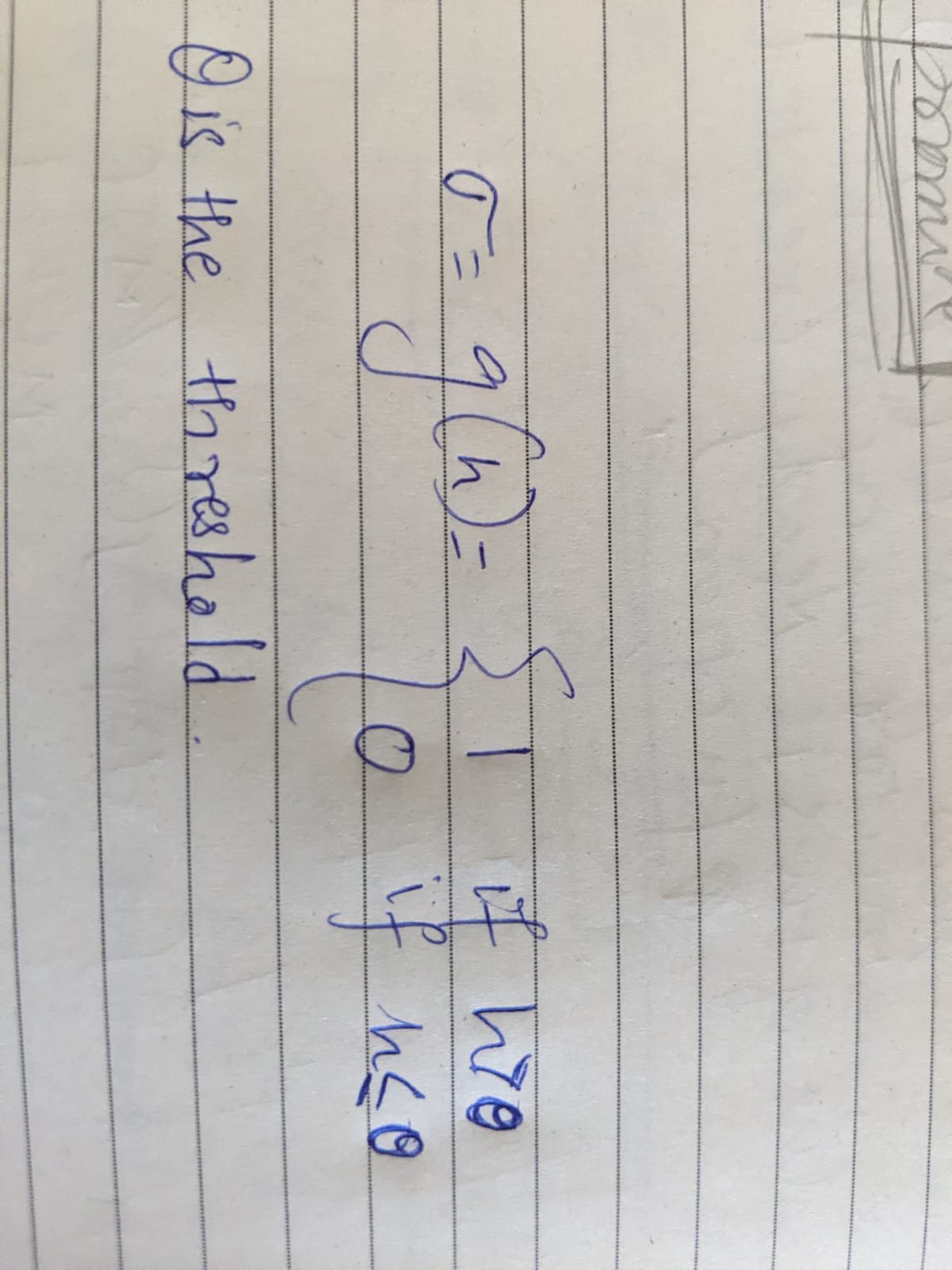
In our first section we discuss math behind Mc Culloch and Pitts Neuron. The idea was to accurately represent the nervous system just on a singular level.

The approach was very simple, and we describe it as:

1. a set of weighted inputs, wi, that correspond to synapses
2. an adder that sums the input signals.
3. an activation function(usually a threshold function), that decides whether the neuron fires(spikes!) for the current inputs.



where wi is the weight at the **synapse** for ith input and xi is the input from ith neuron into synapse. Now to decide if the **neuron fires or not**, we need a **threshold** (θ). If the weighted sum of the inputs(h) is greater than the threshold, the neuron fires(i.e output is 1).

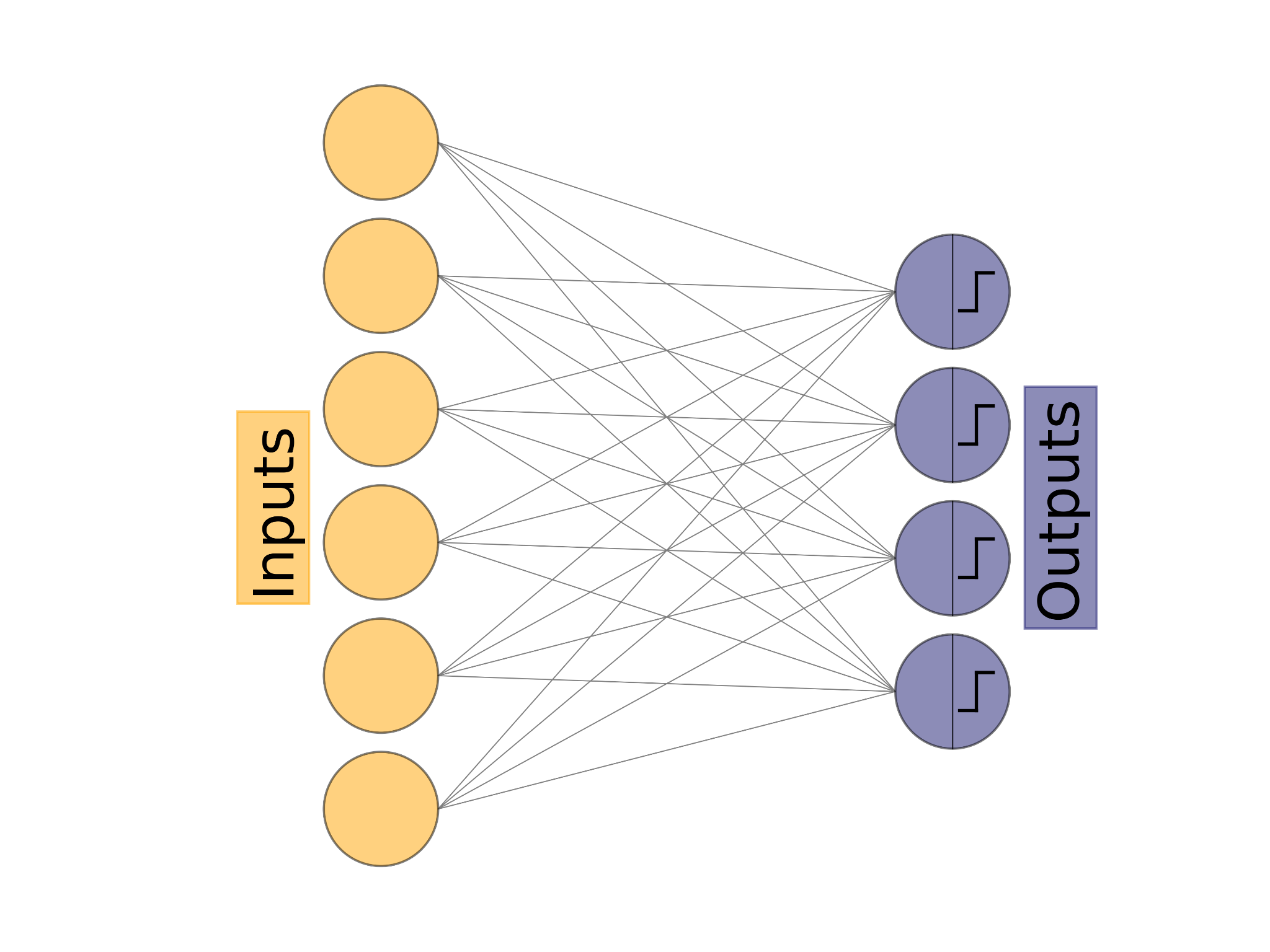


They successfully attempted to mimic the neuron of a human, but there were some limitations:

1. It assumes only linear summation, there maybe non linear summations also.
2. It was only a single entity, but real neurons in human body fire millions of time on a response.
3. The neuron were not able to update themselves, accept any feedback.

This was a bare attempt in modelling how neurons work, but was a significant contribution for further researches.

After a few years came the idea of single layer **PERCEPTRONS**, which were just a collection of Mc Cullloch & pitts Neurons which were completely independent of each other.



Each neuron has its own weights which it multiplies with its input and adds them to decide whether to fire or not depending on its own threshold. The inputs are the number of features(usually columns) our data has. The number of neurons can be varied and is usually the total unique classifying target values. We represent a particular weight as Wij where i is the input it is coming from and j is the neuron it is going into. So W32 is the weight that connects the input node 3 to neuron 2. The input will be stored as a vector and so the output.

Now our agenda is to have a **target vector** and an **output vector**, where we compare our outputs with our targets, and we check neurons with wrong output, because those are the neurons whose **weights need to be adjusted**.

How do you learn weights?

Suppose **kth** neuron gets the **wrong** answer, it has **m weights** connected to it(one for each input node). The weights we need to change is Wik (where i runs from 1 to m).

Now we **know which weights to change**, but by **HOW MUCH TO CHANGE** them by?

Before answering that, let’s figure out if a weight is too high or low(i.e do we need to increase it or decrease it?).

We first say that **bigger weights tend a neuron to fire**(as they help it get over the threshold) and **smaller weights tend to not fire**. So if a neuron fires when it isn't supposed to, there are some weights which are bigger than they should be, or if it doesn't fire when it should(some weights are too small!).

Read that again…

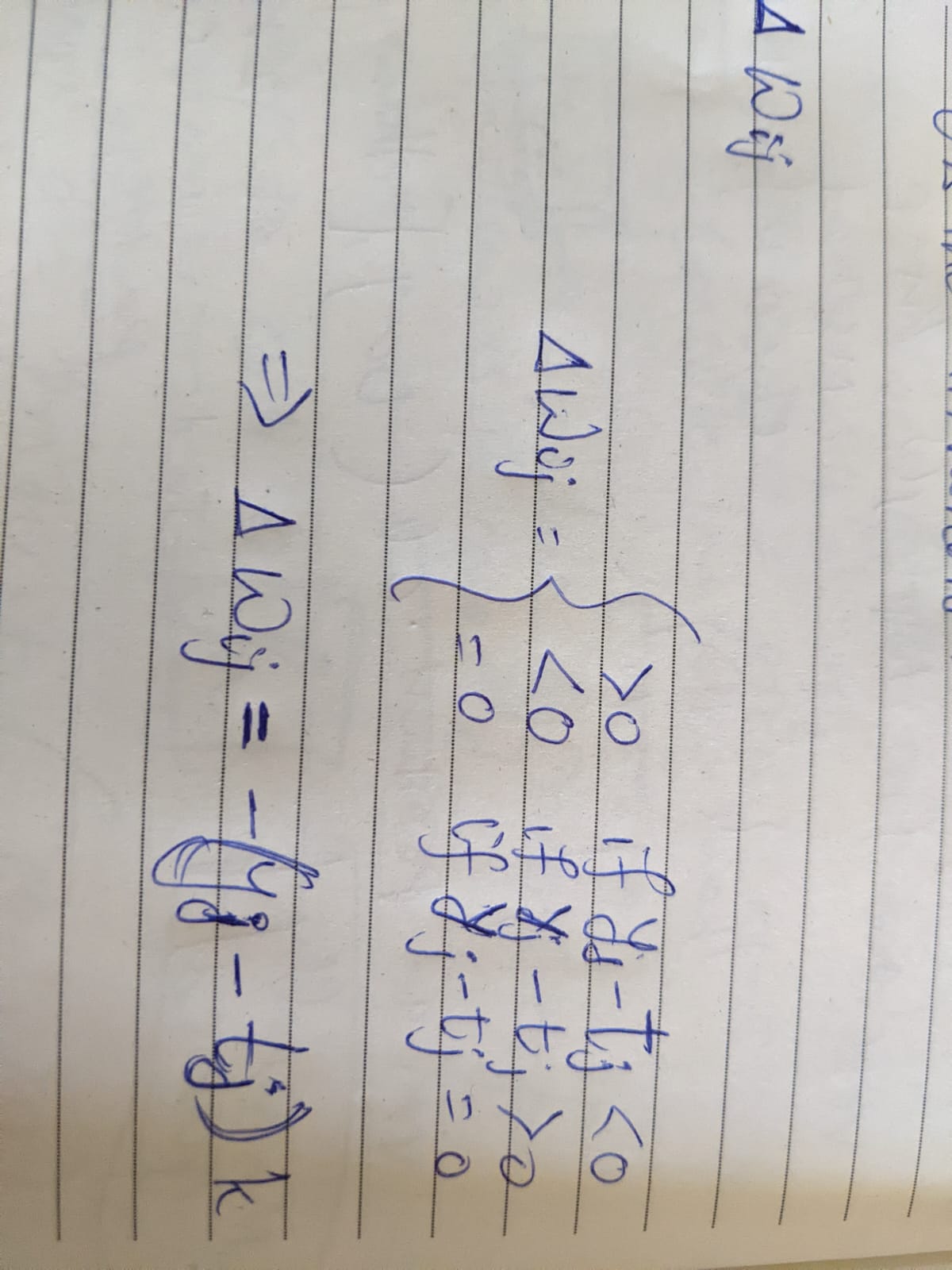
For that neuron we can calculate

**yk - tk**

(i.e the **difference between** the output of the neuron and the actual truth).

This equation can act as a possible **error function**.

the weights Wij will be changed by ΔWij



ΔWij = −(yj−tj)k

where k is the constant which gives the amount by which each weight needs to be changed.

If the inputs are negative, then switch values, we’ll need to reduce the weights to fire and increase to not fire.

ΔWij = −k(yj − tj) xi

The parameter **k** is called the **learning rate** and is often represented by **η** instead of **k**.

ΔWij=−η(yj−tj) xi

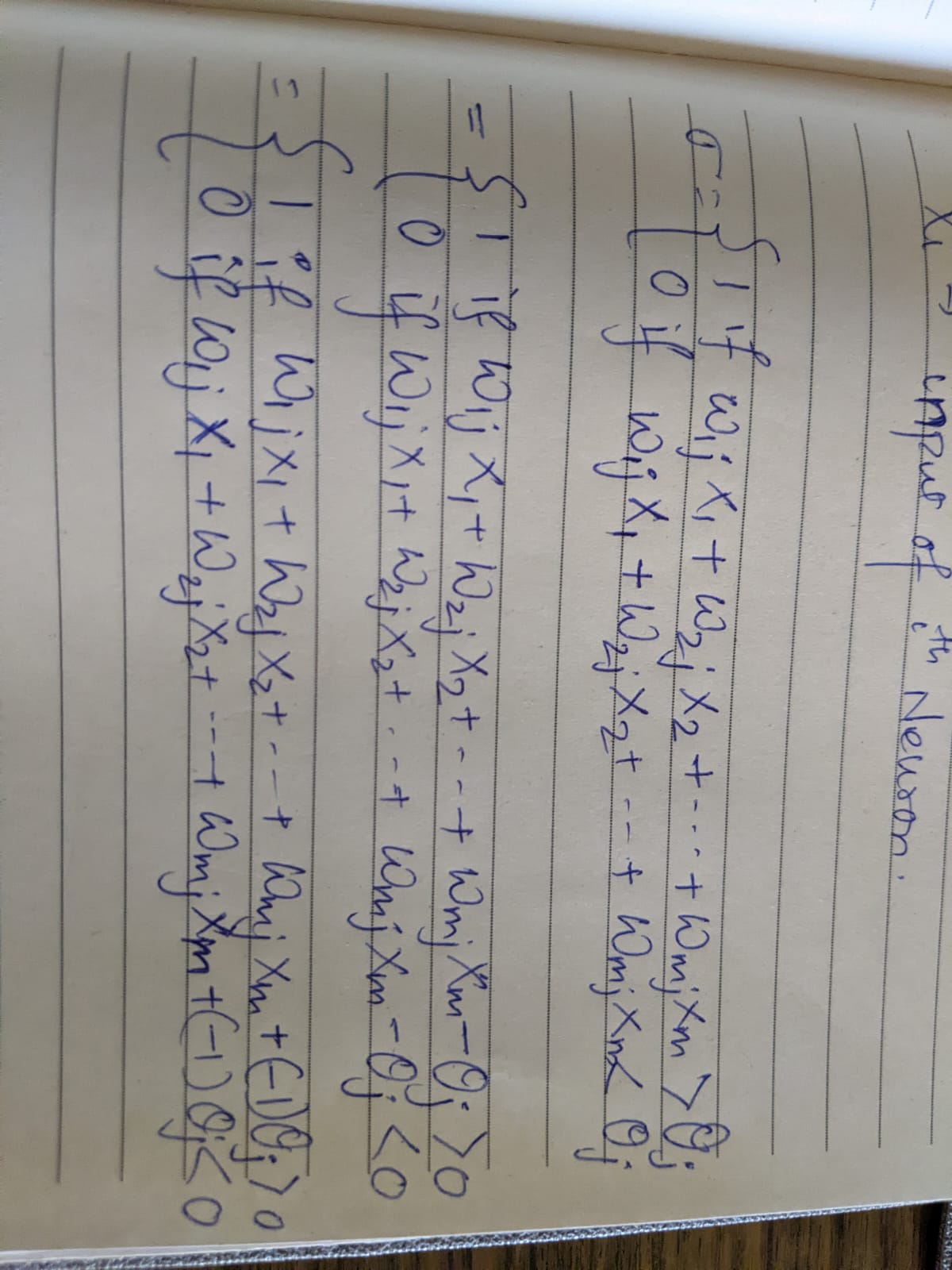
Wij ← Wij − η(yj−tj)xi

The **learning rate is an important parameter** which needs to be tuned to get better **accuracy**. Too high value might change the weights more than they were needed to and a too low will take too long to train. It is often used in the range of 10-4 < η < 1.

Now that we have figured out the weights, we haven’t discussed another important parameter, the **threshold**. **What threshold to choose for what problem**? That is where a **Bias node** comes into play.

**h=W1jx1+W2jx2+⋯+Wmjxm**

**Xk** is the kth input feature.

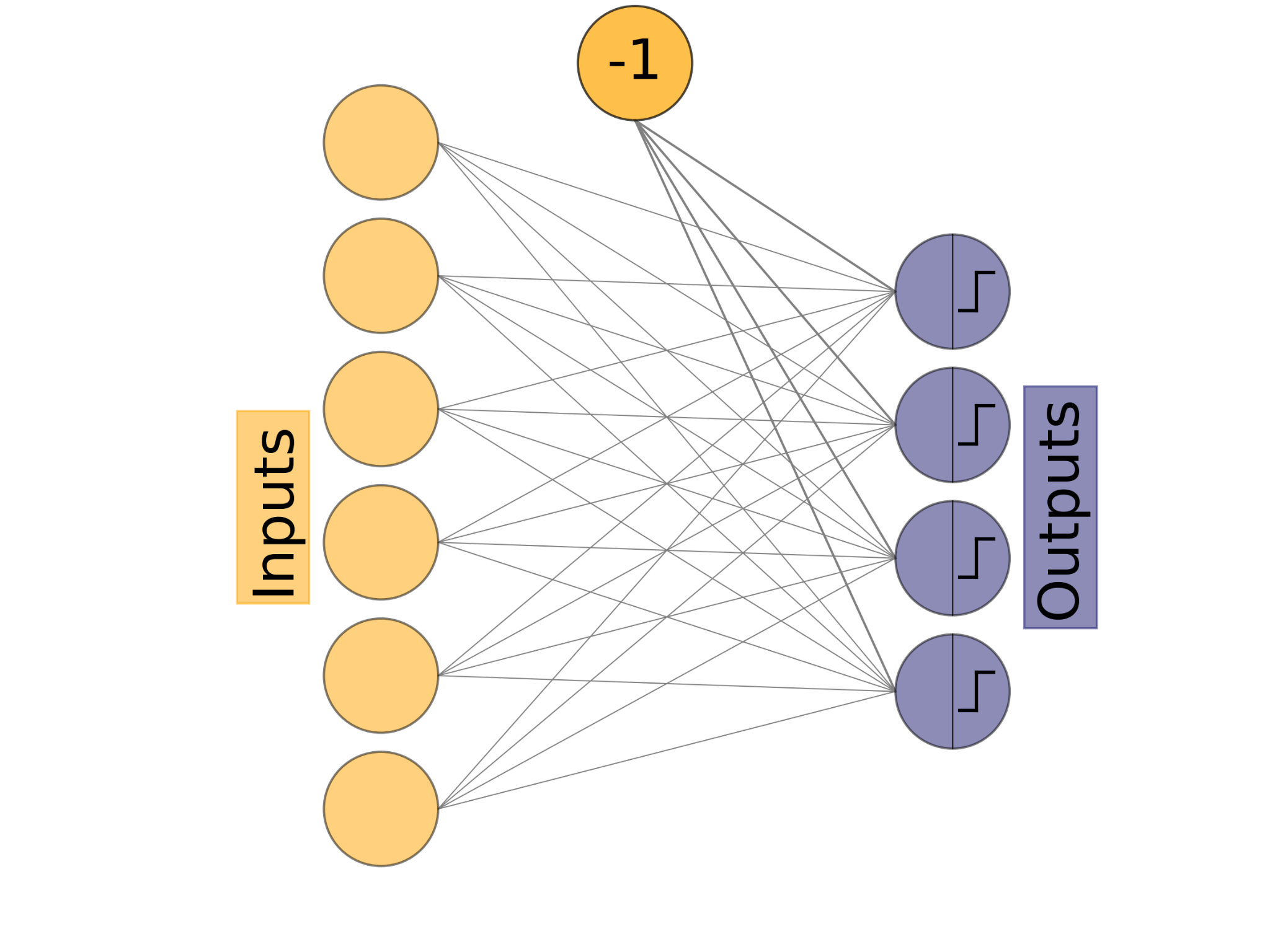


So θ can be another learned weight, we consider an extra input feature which is −1 and our new threshold is 0.

*What if all the Xi’s are zero?*

The **Bias Node** also helps us **overcome the all-zero input problem**.

“If all the inputs in an example are Zero then no matter how the weights change, it won’t change the output, but the bias node will change and make changes necessary for correct output.”



That is all we have to learn for the math of a perceptron. Now we try to put everything together.

We have 3 steps to complete:

1. Initialize the weights randomly
2. Train the model

2.1 Update weights

1. Inference/ Recall
2. Initialize the weights randomly

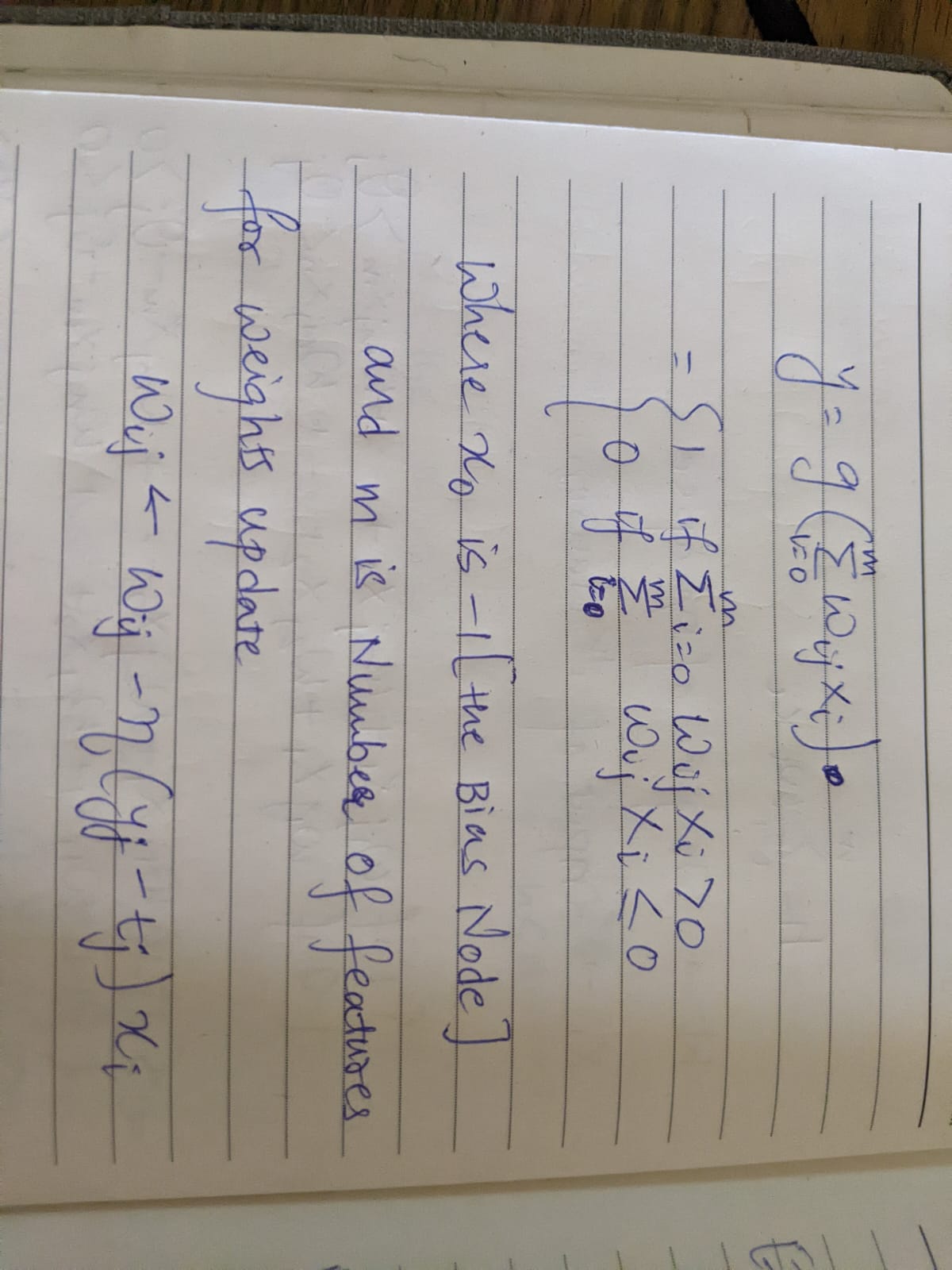
Weights are initialised randomly in a very small range, weights could be positive or negative.

1. Train the model

For T iterations:

For each input vector:

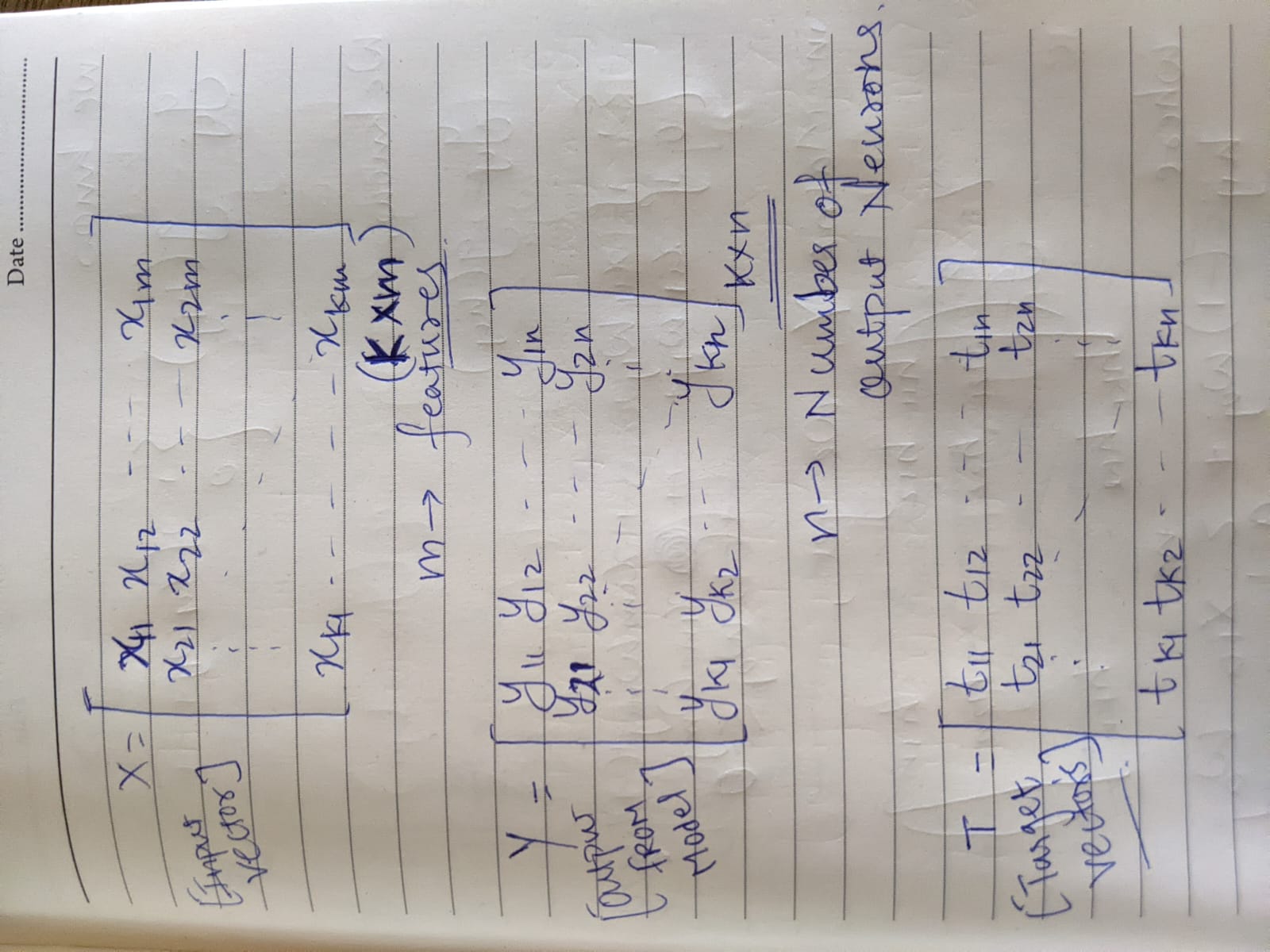
Compute activations & update weights

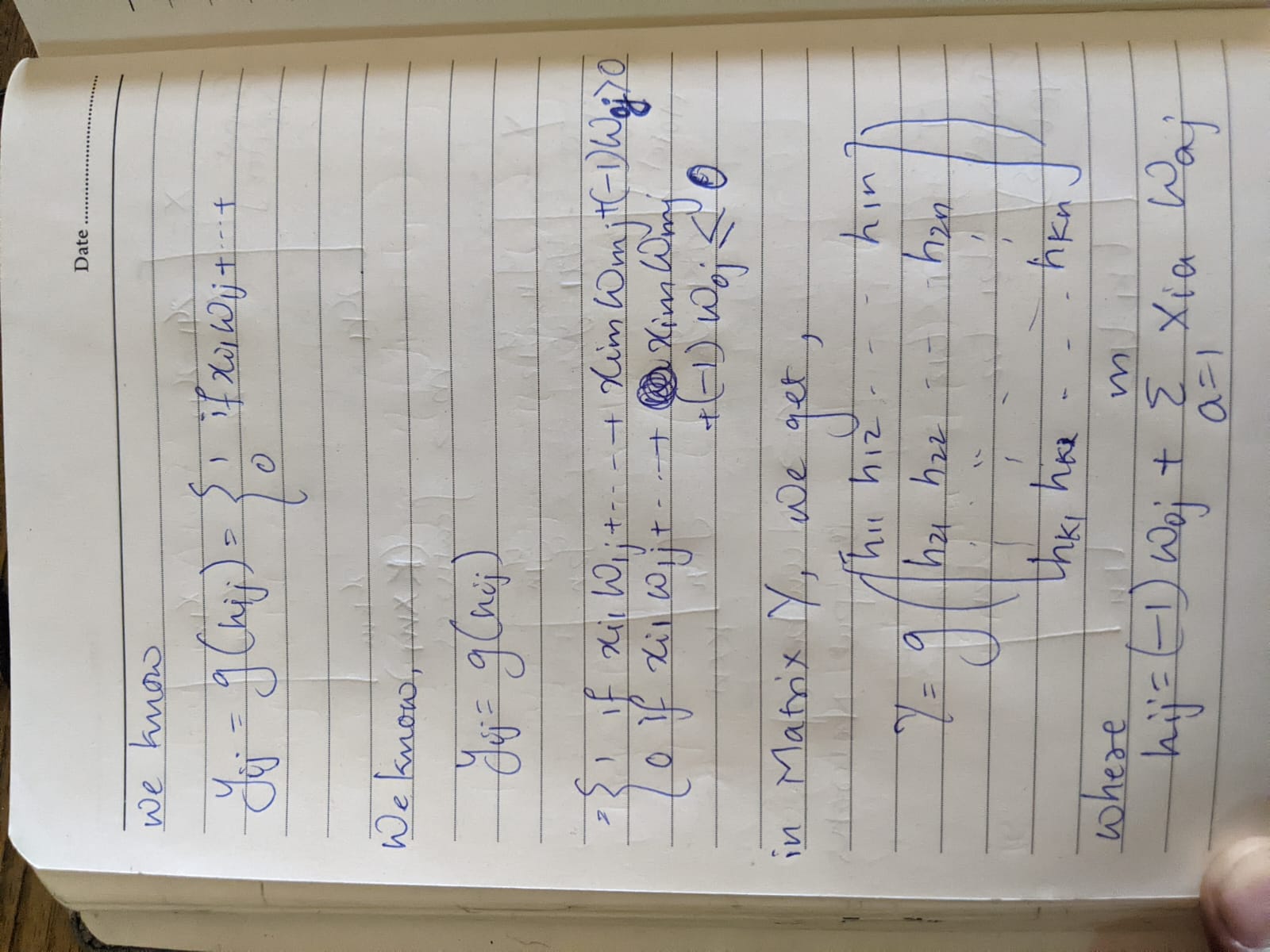


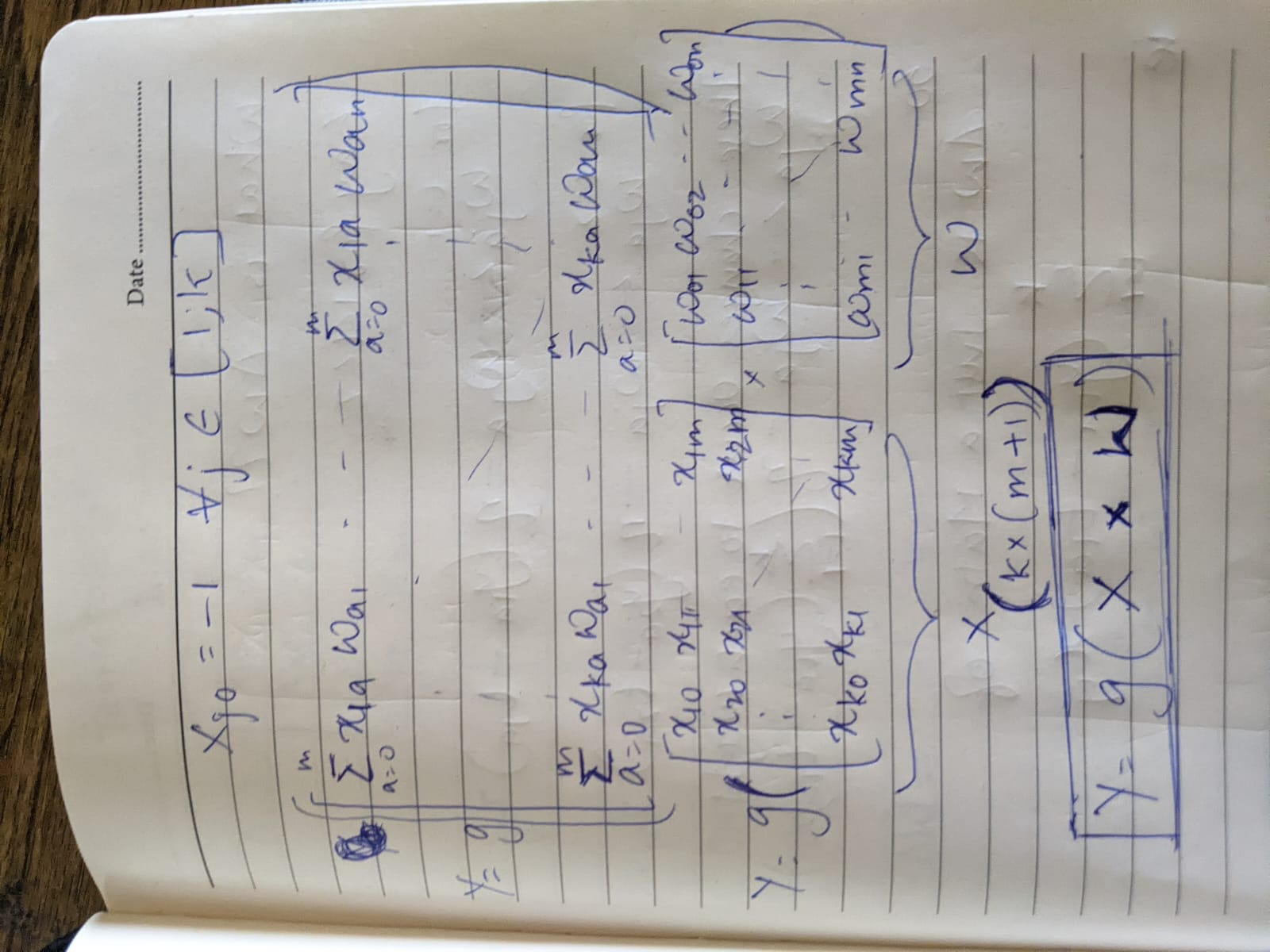
1. Testing with the trained model, to see if it is able to predict correct results or not.

This process of training each data/sample point is computationally expensive, hence we use matrices to represent our data in smaller formats. We use matrices to store intermediate calculations, the resulting matrix stores which neuron fired and which didn’t.

The input vector matrix **X** will be (K x M) order.







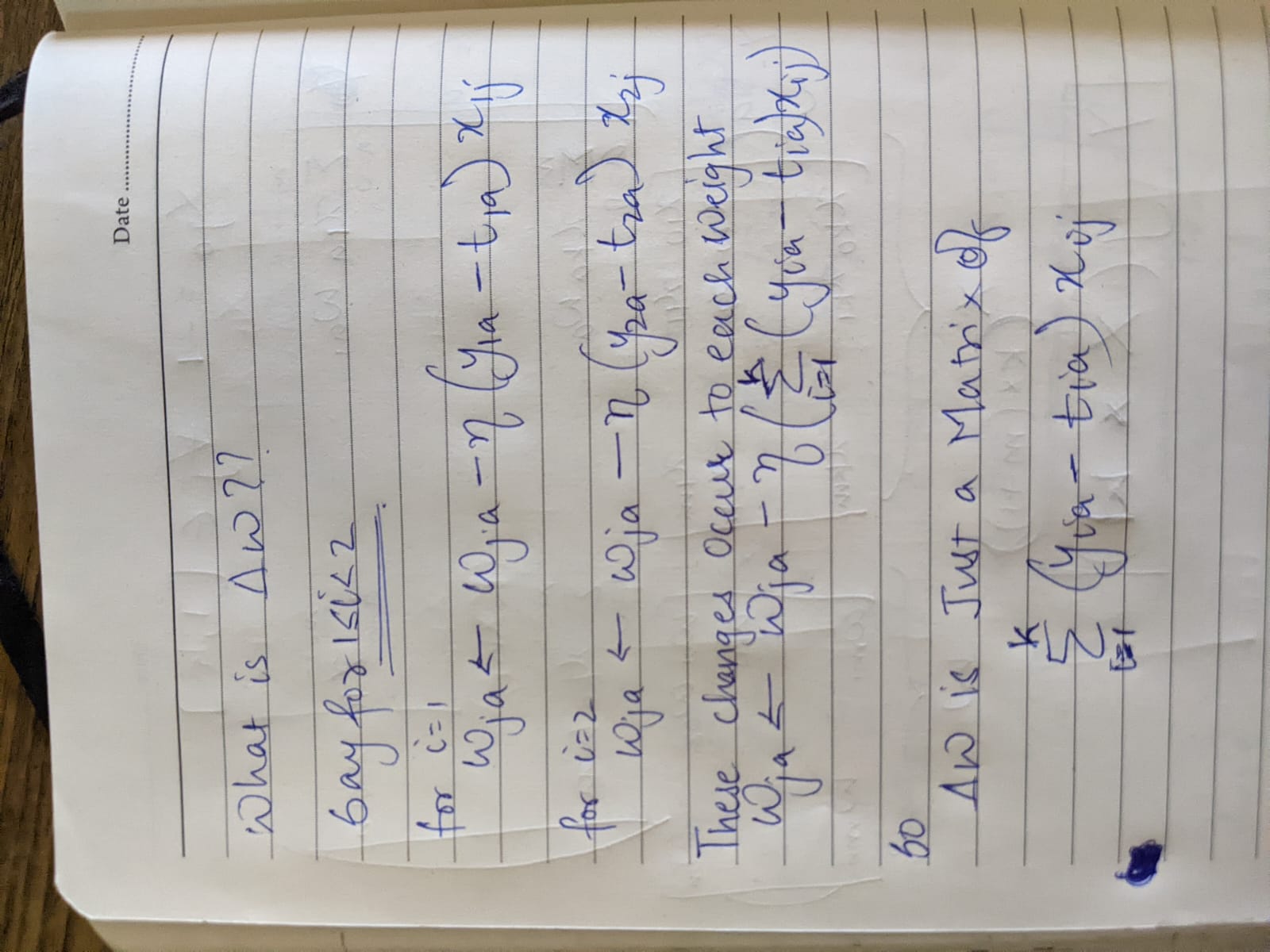
From the left matrix we know the input matrix (X) with the extra column on far left, which is -1, so we redefine the matrix to include extra column.

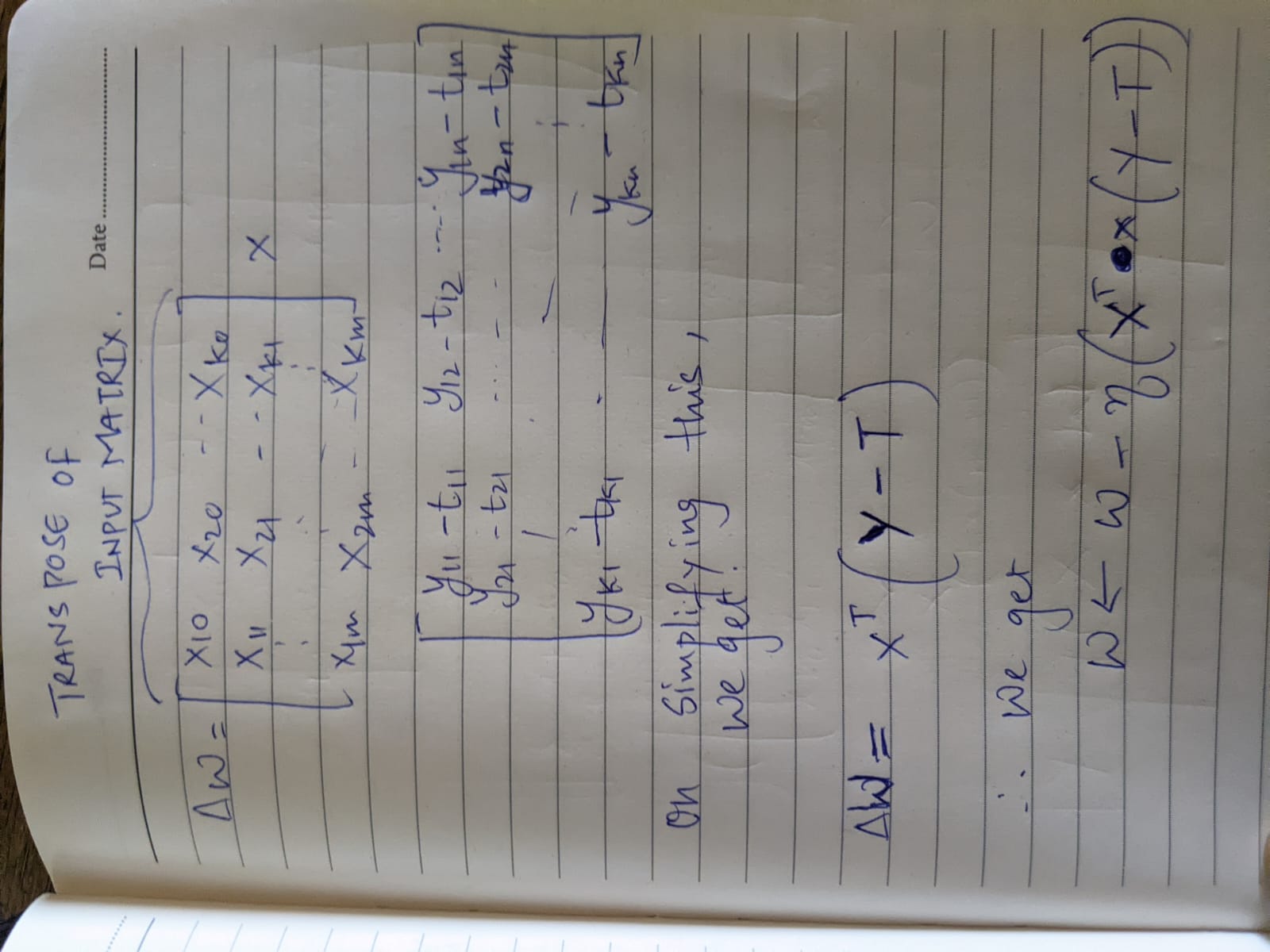
We improve the weight computation from each sample by using a matrix to update weights at once.

We will not dive into extra math of weight updation, we simply write the formula.

W←W−ηΔW(16) [In matrix formation❣️❣️]

where W is the weight matrix and ΔW is the matrix having the corresponding (yj−tj)⋅xi for each weight.





**ΔW=X^T×(Y−T)**

Now we update our weights like this:

**W ← W−η{X^T×(Y−T)}**

**When we do this from N iterations our model adjusts weights and learns to recognize patterns from our data. We learnt how to model a PERCEPTRON & change weights using the error from the matrix.**

We have made a basic **theoretical ground** for NEURONS/SYNAPSES, MODEL, WEIGHTS, BIAS.

IN OUR **NEXT BLOG** WE WILL SEE THE **IMPLEMENTATION** OF THIS **PERCEPTRON** IN **PYTHON.** See you guys in the next one.😄😄😄

DO SHARE THIS WITH YOUR FRIENDS.