Chapter 8 : Part 3

**Deep Learning**

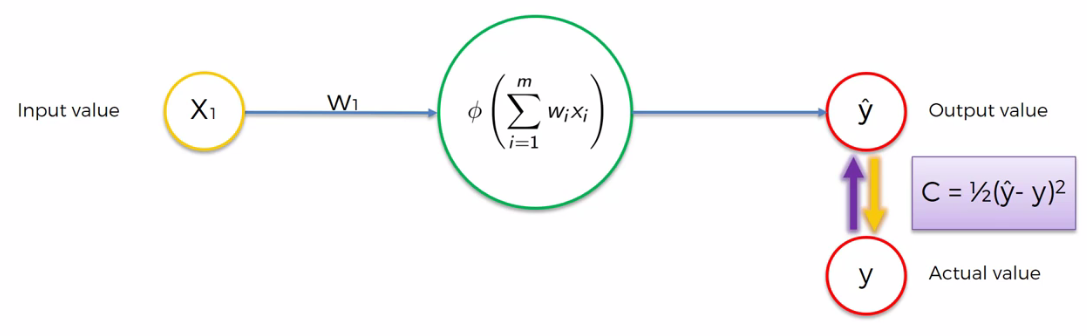
**ANN: Gradient Decent**

Gradient Decent, Stochastic Gradient Decent and Backpropagation details

**8.3.1 Why Gradient Decent?**

|  |  |
| --- | --- |
| In this section we're talking about gradient descent. We saw in previous section, that an *NN* learns using *back-propagation*. That is, when the error/difference or the sum of squared differences between and is ***back*** ***propagated*** through the ***neural*** ***network*** and the ***weights*** are ***adjusted*** accordingly. Now we're going to learn exactly how these weights are adjusted. |  |

* Gradient Descent is kind of similar to some Numerical techniques like Bisection or Regula falsi methods.
* Now consider the very *simple version of a neural work* or a PERCEPTRON, a single layer feed-forward NN. We can see here is the whole process in action: we've got some input value, we've got a weight then a activation function is applied. Then we got predicted value and we compare it to the actual value, then we calculate the cost function.



* Now the question is: How can we minimize the cost function?

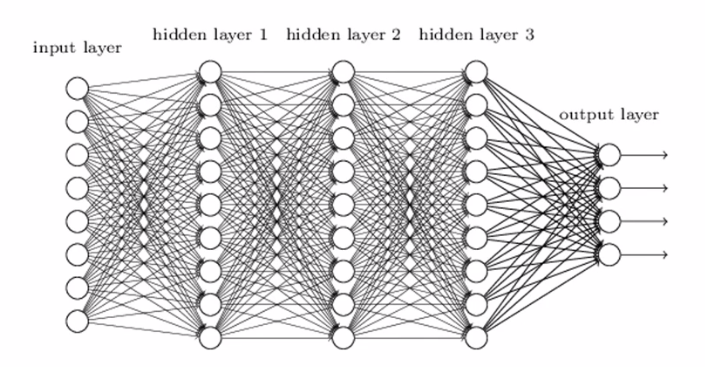
|  |  |
| --- | --- |
| * Well one approach to do it is a Brute Force Approach where we just take all lots of different possible weights and look at them and see which one looks best. * In this approach we would try out weights , for example: 1000 weights. Then we get something like this: * For the cost function and this is a chart. On the Y axis we have cost function and on X-axis we have output value **.** You'd find the ***best*** ***value*** at the bottom of that curve. |  |

* Why this brute force method is not efficient?: Well if you have just one weight to optimize this might work but as you increase the *number* of *weights* (i.e. *increase* the *number* of *Synapses* in your network) you have to face the Curse of Dimensionality.
* Curse of Dimensionality: Recall the example, when we were talking about how neural networks actually work where we were building a NN for a Property Evaluation (Recall Section 8.2.1).

|  |  |
| --- | --- |
| * So this is what it looked like when it was Trained Up Already. Here we know which one what are the best weights. | * But Before Training we didn't know about the optimal weights. The actual NN before training looks like this. Where we have all these different possible Synapses and we still have to train up the weights. Here we have a total of 25 weights: ***input*** () and ***output*** () so total ***25*** weights. |
|  |  |

|  |  |
| --- | --- |
| * And let's see how we could possibly brute force 25 ways. Now it is a very simple neural network we have right now. We have just one Hidden Layer. * But to do this we have to try possible combination. Which is impossible even we use a super computer (using 93 peta FLOPS. 1 FLOP = 1 floating point operation per second. Normal computer can do several giga FLOPs). |  |

* Now for an NN like following it is just impossible to try out the Bruit-Force technique.



Notes:

* Brute-Force Search: In computer science, *Brute-Force* search or *Exhaustive* search, also known as *Generate And Test*, is a very general problem-solving technique and *algorithmic* *paradigm* that consists of *systematically* *enumerating* all *possible* *candidates* for the *solution* and checking whether *each* *candidate* *satisfies* the problem's statement.
* *Brute Force* Algorithms are exactly what they sound like – *straightforward* methods of *solving* a *problem* that rely on *sheer computing* *power* and trying *every* *possibility* rather than *advanced* *techniques* to improve *efficiency*.
* For example, imagine you have a small padlock with 4 digits, each from 0-9. You forgot your combination, but you don't want to buy another padlock. Since you can't remember any of the digits, you have to use a brute force method to open the lock.
* So you set all the numbers back to 0 and try them one by one: 0001, 0002, 0003, and so on until it opens. In the worst case scenario, it would take , or 10,000 tries to find your combination.
* A classic example in computer science is the Traveling Salesman Problem (TSP). Suppose a salesman needs to visit *10 cities* across the country. How does one determine the order in which those cities should be visited such that the total *distance* *traveled* is *minimized*?
* The brute force solution is *simply to calculate* the *total* *distance* for *every possible route* and then select the *shortest* *one*. This is not particularly *efficient* because it is possible to eliminate many possible routes through *clever* *algorithms*.

Gradient Decent: Similar to Numerical methods: Bisection/Regula-falsi

**8.3.2 Gradient Decent**

|  |  |
| --- | --- |
| It is called gradient descent because we calculate the Gradient every-time and check if it is descending.   * So here is our cost function, and now we going see how we can find our *best* *value* faster. * Lets say we start somewhere at point in the top left, we're going to look at the angle of the tangent drawn at that point of our cost function then we calculate the gradient (you have to differentiate). |  |

* You just need to *differentiate* to find out what the *slope* is in that *specific* *point* and find out if the *slope* is *positive* or *negative*.
* If the *slope* is *negative* (like in this case), means that you're *going* *downhill* (so to the *right* is *downhill* to the *left* is *uphill*), so you need to *go right* i.e. you need to *go downhill* (find the best/optimal point. Something like *"finding global minima"*).
* Lets say we go to *right*, and again we calculate the *gradient* and find the *slope* is *positive*. That means we are *too* *right* we need to go *left*.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  | |

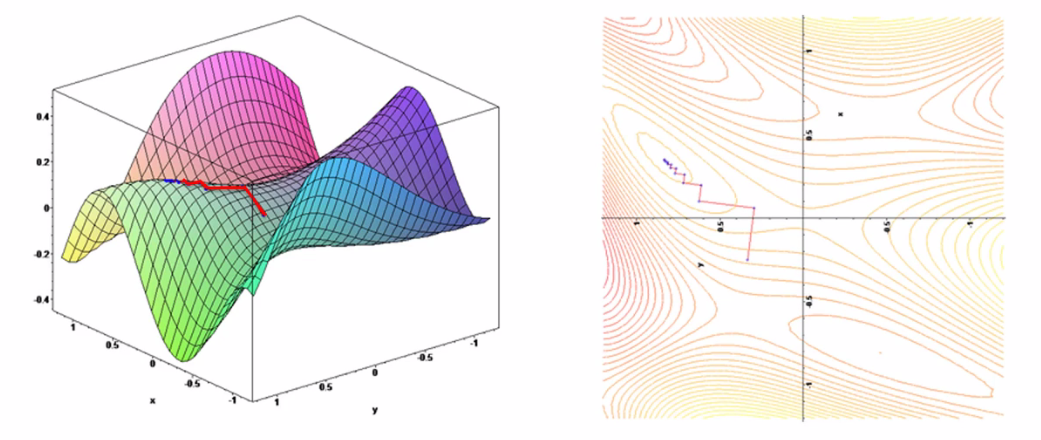
|  |  |
| --- | --- |
| * Gradient: The Gradient (also called Slope) of a straight line shows how steep a straight line is. * To calculate the Gradient, divide the *change in height* by the *change in horizontal distance.* |  |
| * To remember it as a fun way to think it as a ball rolling. But in reality it's going to be like a step by step approach is going to be a zigzag type of method. |  |

* There's also lots of other elements to it. For instance: Why does it go down why does it *not go way over* the line or it could have gone *upwards* instead of *downwards*. So there are *parameters* that you can tweak.
* Here, we are getting to the bottom to understanding which way we need to go. Instead of *Brute-force* through *billions and quadrillions* of *combinations*. We can simply look at which way is it *sloping*: *right/left*. Then we try to get to the *bottom*. (like you're standing on a hill. Which way does it feel that it's going downwards).

|  |  |
| --- | --- |
| **8.3.3 Gradient Descent in 2D**  Here's an example of gradient descent applied in a two dimensional space. You can see it's getting closer to the minimum and hence it is called *gradient* *descent* because you're *descending* into the *minimum* of the *cost function*. |  |

**8.3.4 Gradient Descent in 3D**

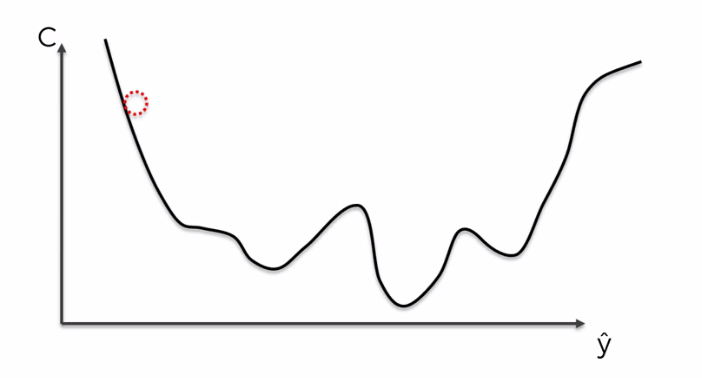
Here is a gradient descent applied in 3D. And if you projected onto two dimensions you can see *zigzagging* its way into the *minimum*.



**8.3.5 Stochastic Gradient Decent**

|  |  |
| --- | --- |
| Above we see that gradient descent is a very efficient method to solve our *optimization* *problem* where we're trying to *minimize* the *cost* *function*.  It basically takes us from years to solving a problem within *minutes* or *hours* or within a *day* or so. And it really *helps* *speed* *things* *up* because we can see which way is *downhill* and we can just go in that direction and get to the *minimum* faster. |  |

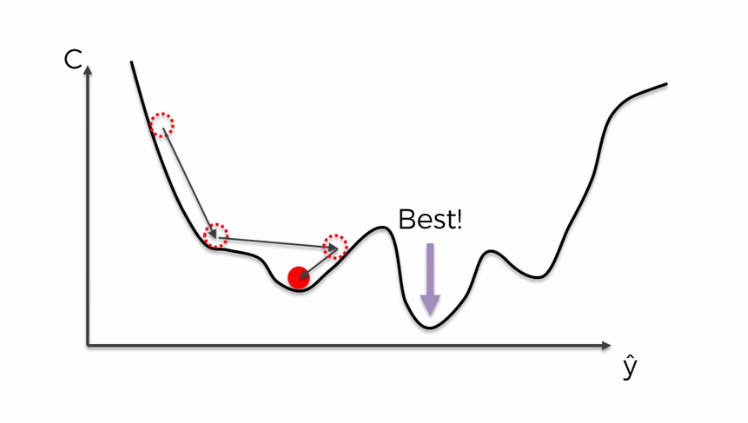
* Non- CONVEX cost-function: But the problem is, this method *requires* for the *cost* *function* to be *convex*. Which has only one global minima. What if our cost-function is not CONVEX. What if it looks something like this:



* This could happen:

1. If we choose a cost function which is not the square difference between and or
2. If we do choose the cost function as square difference between and but in a *multi* *dimensional* *space* it can actually turn into something that is *not convex*.

* The local-minima trap: In these cases if we apply our normal gradient decent method we will end up something like following. Which is not the global minima, we'll end up with the local-minima. But the *best* *value* is the *Global-minima*.

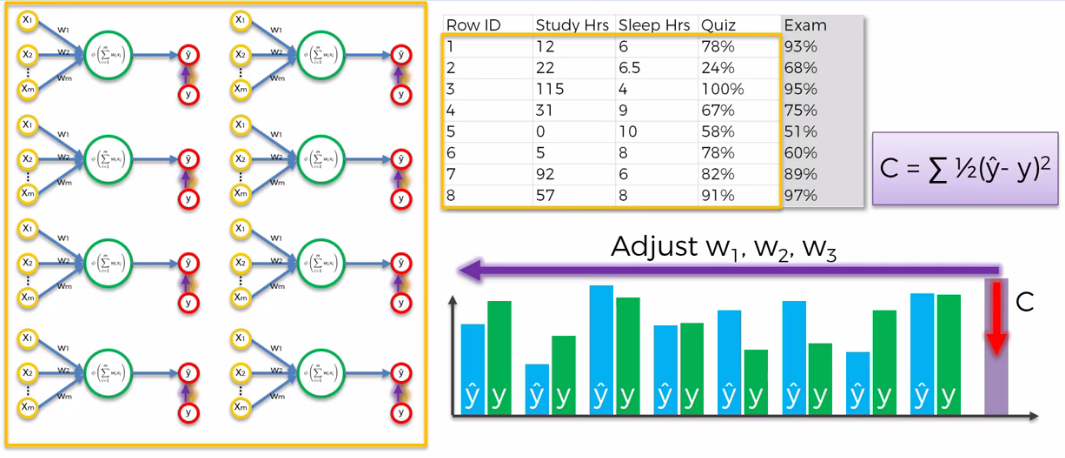


We could find a local minimum of the cost function rather than the global one.

* *Global-minima* is the best one and we found the wrong one (the *Local-minima*) and therefore we don't have the correct weight. As a result we failed to find an optimized neural network.
* So how to avoid this trap?

The solution is Stochastic Gradient descent. *Stochastic gradient descent* doesn't require for the Cost-function to be convex.

* So let's have a look at the differences between the normal Gradient Descent and the *stochastic gradient descent*.
* Batch Selection in normal Gradient descent: In normal Gradient Descent we take all of our rows we ***plug*** ***them*** into our ***neural*** ***network*** all at once.

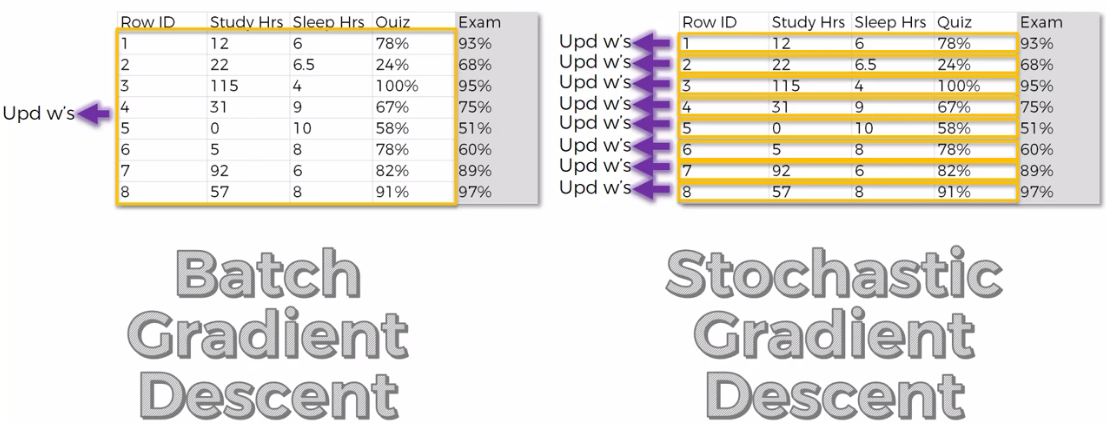


*Batch Selection in normal Gradient descent*

* Here we've got the *neural* *network* (copied over several times) the *rows* are being plugged into that *same* *neural* network every time.
* Once we plug them in, we've *calculated* our *cost* *function* and then we adjust the weights. This is called the ***gradient descent method*** or in the proper term ***"batch gradient descent method"***.
* Stochastic Gradient Descent: The stochastic gradient descent method is a bit different. Here we take the rows one by one. We take that row, we run our NN and then we adjust the weights.
* Then we move onto the second row we run our NN again, then adjusted the weights again. Then we move to 3rd row. And so on.
* So basically we're *adjusting* the *weights* after *every* *single* *row* rather than doing everything at once.

|  |
| --- |
|  |

* And now we're going to just compare the two side by side. For visually remember them.



* In the Batch Gradient Descent, you are adjusting the weights *after you've run all of the rows* in your NN and then adjust the weights and that’s the first iteration.
* For the *2nd iteration* we repeat whole thing *adjusting* the *weights* and then we do everything again for *3rd iteration*.
* The Stochastic Gradient Descent method helps you to avoid the problem with *local* *extrema* or *local* *minima* . It helps to find *global* *minima*.
* And the reason is that SGD or STOCHASTIC GRADIENT DESCENT method has much higher fluctuations, because it is doing *one iteration* on *one row at a time* and therefore the fluctuations are much higher and it is *much more likely to find* the *global* *minimum* rather than just the *local* *minimum*.
* SGD is more faster than BGD: Stochastic Gradient Descent has more advantages over the Batch Gradient descent method. At the first impression you might have think that Batch Gradient Descent is more faster because it's doing all row once at a time. But in reality Stochastic Gradient Descent is faster because it doesn't have to load up all the data into memory and *run and wait* until all of those rules are on altogether.
* The main advantage is that BGD is a Deterministic algorithm but SGD being a Stochastic algorithm (meaning it's random).
* With BGD as long as you have the same starting weights for your NN. Every time you run, you will get the same iterations to update the weights.
* But for *SGD* you won't get that because it is a *stochastic* *method* you're picking your rows possibly at *random* and you are updating your NN in a *stochastic* *manner*. Therefore in SGD even if you have the same weights at the start you're going to have a different iterations to get there.

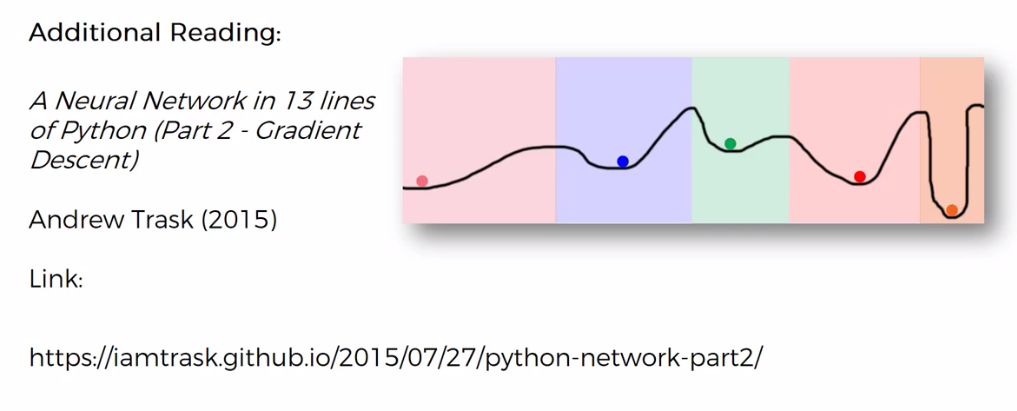
**8.3.6 Mini Batch Gradient Descent**

There's a method in-between the BGD and SGD, it is called the Mini Batch Gradient Descent (mini BGD) method. Where you are running *smaller batches of rows* rather than running *whole* *batch* once at a time.

* From your *whole* *dataset*, from *all the rows*, you set that *batch* *size* (number of rows). You *divide* all the rows into *several* *groups* and you run your *NN* through each group and update the *weight*.

And that's called the Mini Batch Gradient Descent method if you'd like to learn more about gradient

* Additional Reading: There's a great article which you can have a look at. It's called A Neural Network In 13 Lines Of Python Part 2 Gradient Descent by ***Andrew Trask*** and the links below it's an good 2015 article very well-written, very simple terms. You will got some very cool tips-tricks and hacks.
* It discussed on how to apply Gradient,
* know advantages and disadvantages and
* how to do things in certain situations so



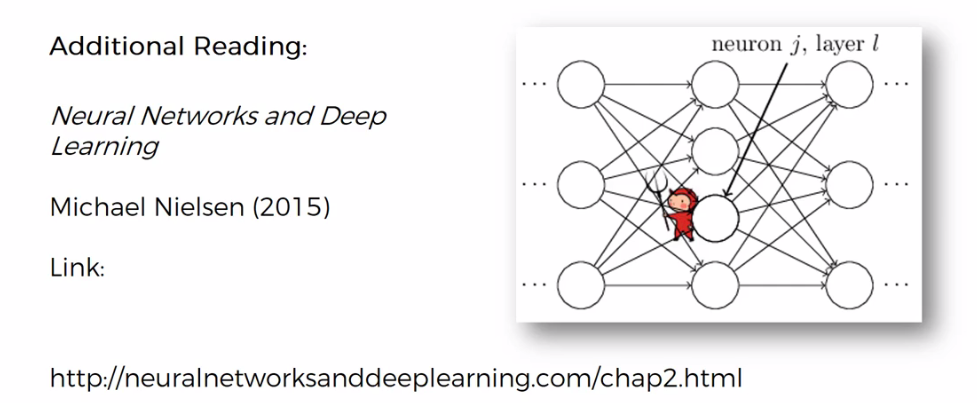
|  |  |
| --- | --- |
| * Book: There is another article, it's a bit more heavier to read. For *those* of you who are *into* *mathematics* who want to get to the *bottom of the mathematics*. What is Gradient descent is specifically. *What are the formulas* that are *driving* *Gradient* and *how is it calculate* and so on. Check out the Article or actually the Book. * It's a free online book called Neural Networks And Deep Learning by Michael Nielsen 2015 book. |  |

**8.3.7 Back-propagation**

* Forward propagation: There's a process called Forward propagation where information is entered into the input layer and then it's *propagated* *forward* to get our *output* value .
* Then we compare those to the *actual* value in our Training set.
* And then we *calculate* the *errors* then the *errors* are Back Propagated through the *network* in the *opposite* *direction* and that allows us to train the network by adjusting the weights.

|  |  |
| --- | --- |
|  |  |

* Back propagation adjusts all the Weights simultaneously: Back propagation is an advanced algorithm driven by very interesting and sophisticated mathematics which allows us to *adjust* the *weights*. All the weights at the same time are adjusted simultaneously.
* The huge advantage of Back propagation and it a key thing to remember is that during the process of back propagation, you are able to adjust all the *weights* at the *same time*, because that's the way the algorithm is structured. This is the key fundamental underlying principle of back propagation.
* So you can track which part of the *error* each of your weights in the *neural* *network* is responsible for.
* Adjust Each Of The Weights Independently/Individually: If we were doing this *manually* or if we're coming up a *very different type of algorithm* than even if we calculated the error and then we were trying to understand what effect each of the weights has on the error then we'd have *adjust* each of the *weights* *independently/individually*.
* And if you'd like to learn more about that and how exactly the mathematics works in the background then a good article which we've already mentioned is the Neural Networks And Deep Learning is actually a book by ***Michael*** ***Nielsen***. You'll find the mathematics written out and it will help you understand how exactly this is possible.



**8.3.8 Steps on Training Of A Neural Network**

Now we're going to just wrap everything up with a Step By Step Walkthrough of what happens in the Training Of A Neural Network.

* STEP 1: We randomly initialize the ***weights*** to ***small*** ***numbers*** close to ***0*** but not zero. i.e. They are initialized with random values near zero. And from there through the process of Back Propagation these *weights* are *adjusted* until the *error/* *cost function* is *minimized*.
* STEP 2: Input the *first* *observation* of your *dataset* in the *input* *layer*, each ***feature*** in one***input node***. That is the first row into input layer and each ***feature*** in one***input node.***
* STEP 3: Forward-Propagation: from *left* to *right*, the neurons are activated in a way that the *impact of each neuron's activation* is *limited* by the *weights* *(the* ***weights*** *basically* ***determine*** *how important each* ***neurons******activation*** *is)*. Propagate the activations until getting the ***predicted*** ***result*** .
* STEP 4: Compare the *predicted* result to the *actual* result . Measure the generated ERROR.
* STEP 5: Back-Propagation: from *right* to *left*, the Error is Back-Propagated. *Update* the *weights* according to *how much they are* ***responsible*** for the *error*. The learning rate decides by ***how much*** we update the weights. The *learning* *rate*, as a ***parameter*** you can control it in your neural network.
* STEP 6:
* SGD: Repeat *Steps 1 to 5* and *update* the *weights* after each *observation* (this is called *reinforcement* *learning* and in our case that was *stochastic gradient descent*).
* BGD/Mini-BGD: Repeat *Steps 1 to 5* but update the *weights* only after a *batch of observations* (Batch Learning).
* STEP 7: When the whole training set passed through the ANN. that makes an Epoch. Redo More Epochs. Just keep doing that to allow your ***NN*** to ***train*** better and better and ***constantly adjust itself*** until you ***minimize*** the ***cost function***.

Those are the steps you need to take to build your artificial neural networks (ANN) and train it.

