Neural Networks and Back-Propagation

Harvey Alférez, Ph.D.

Inputs

- learn (inputs, outputs)
updates
Internal state

Outputs

Inputs

using
Internal state
- predict from (inputs)

Outputs

A Simple Numerical Example

Input	Desired output
0	0
1	2
2	4
3	6
4	8

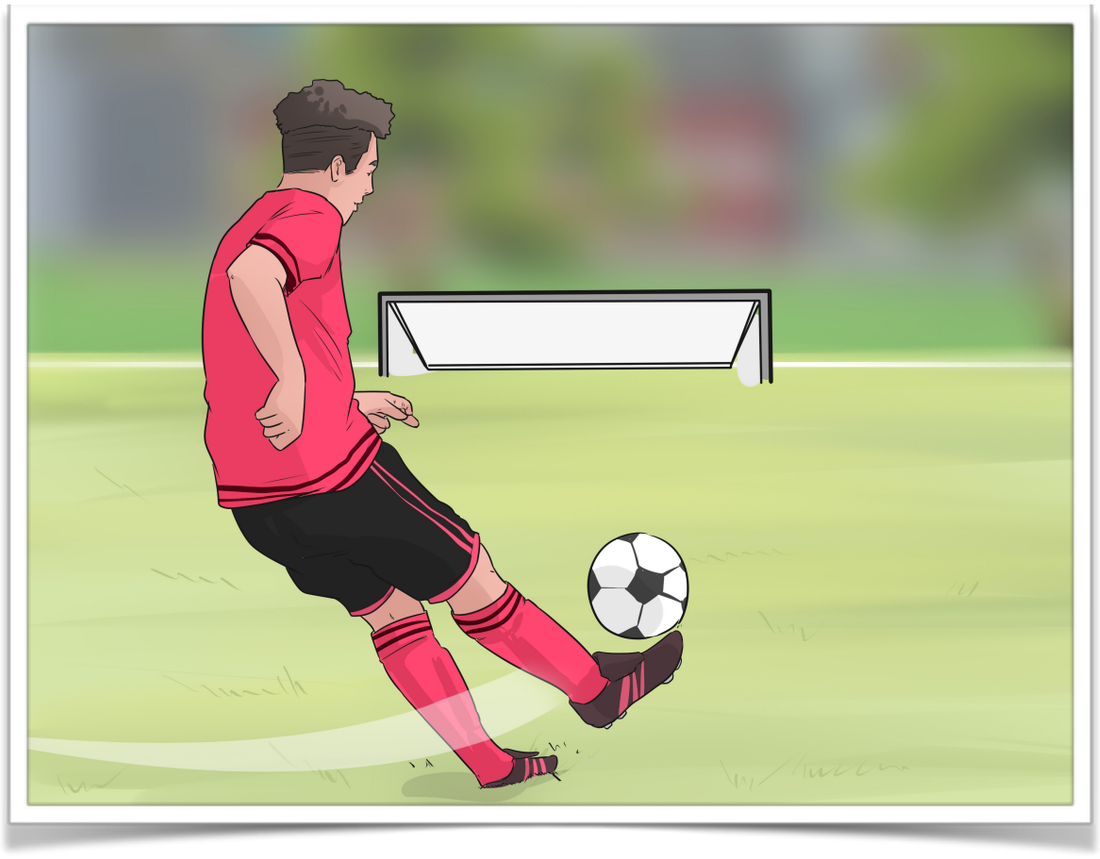
Output = 2 x input

Step 1 - Model Initialization

- A **random initialization** of the model is a common practice.

- From **wherever we start**, if we are **perseverant enough** and through an iterative learning process, we can reach the **ideal model**!

Step 1 - Model Initialization



- Random initialization (3, 5, 0.5 are random values):
  - Model 1: y=3.x
  - Model 2: y=5.x
  - Model 3: y=0,5.x

Through the learning process, all of these models can **converge** to the ideal solution **(y=2.x)**

Step 1 - Model Initialization

- We are exploring **which model** of the generic form ***y=W.x*** can fit the best the current dataset.

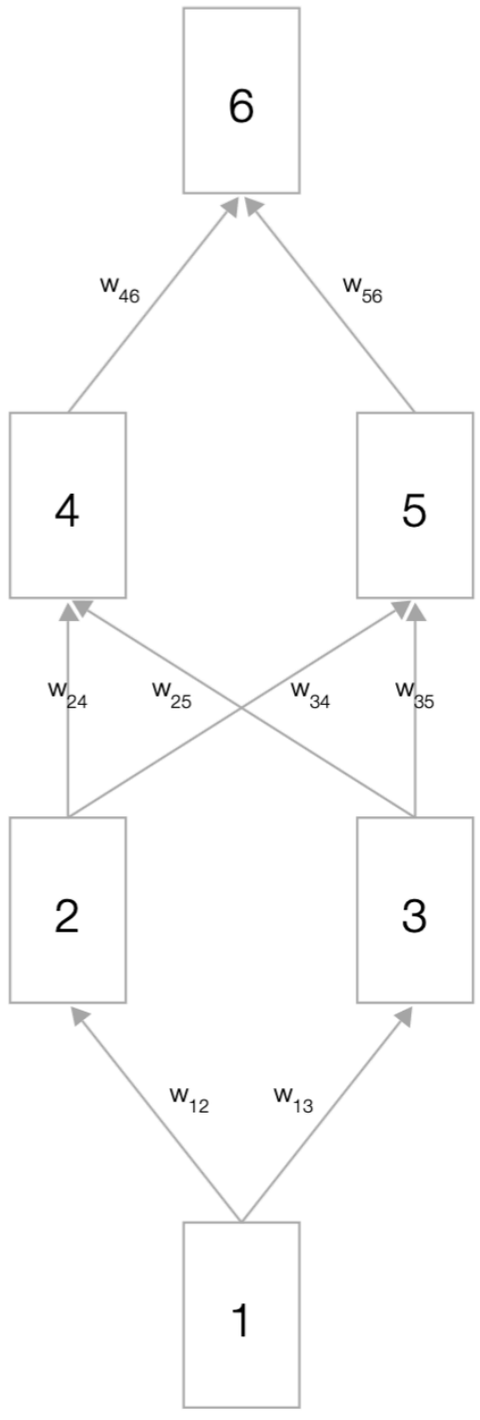
- Where ***W*** is called the ***weights*** of the network and **can be initialized randomly**.

Step 2- Forward propagate

Input	Actual output of model 1 (y= 3.x)
0	0
1	3
2	6
3	9
4	12

Forward-propagation: the calculation flow is going in the natural **forward** direction from the input -> through the neural network -> to the output.

Step 2- Forward propagate



Step 3- Loss function

Input	Actual output	Desired output
0	0	0
1	3	2
2	6	4
3	9	6
4	12	8

What he can **learn** from this, is that he needs to shoot a bit more to the left next time he trains.

Step 3- Loss function

- ***Loss = Absolute value of (desired—actual)***
    - However, several situations can lead to the same total sum of errors:
        - lot of small errors or
        - few big errors can sum up exactly to the same total amount of error.
- **It is more preferable to have a distribution of lot of small errors, rather than a few big ones.**

Step 3- Loss function

- **Loss function:** sum of **squares** of the **absolute errors.**
  - **Small errors** are counted **much less than large errors!**
    - Absolute Error 1 = 2, $2^2 = 4$
    - Absolute Error 2 = 10, $10^2 = 100$ (more penalization)
      - An error of 10, is penalized 25 times more than an error of 2 (100 / 4 = 25)—not only 5 times (10 / 2 = 5)!

Step 3- Loss function

- **Summary:** the **loss function** is an error metric, that gives an indicator on how much **precision** we lose, if we replace the real desired output by the actual output generated by our trained neural network model.

- That's why it's called **loss**!

    - The machine learning goal becomes then to **minimize** the **loss function** (**to reach as close as possible to 0**).

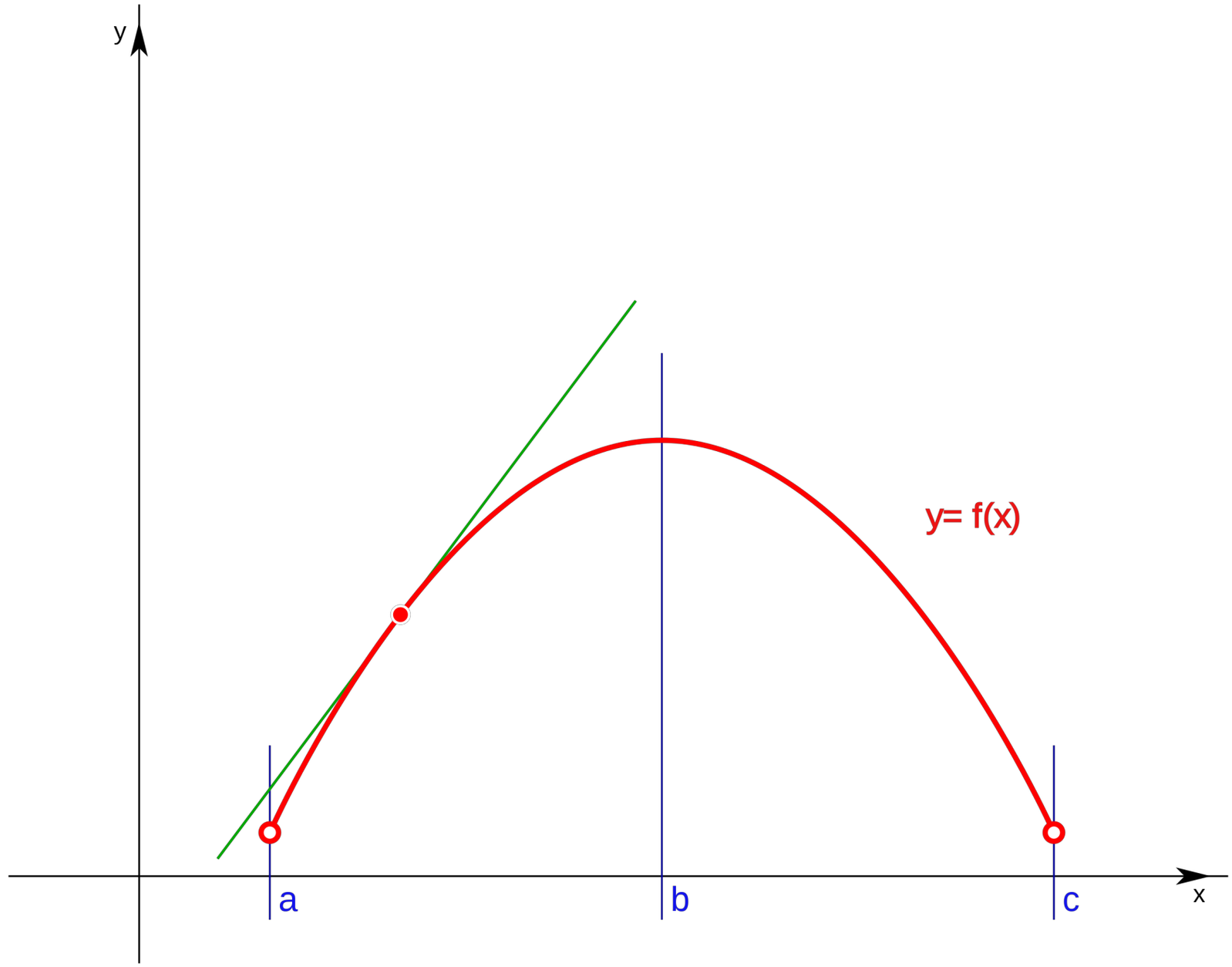
Step 4- Differentiation

- **Differentiation:** A powerful concept in mathematics that can guide us how to optimize the **weights**.

- It deals with the **derivative of the loss function**.

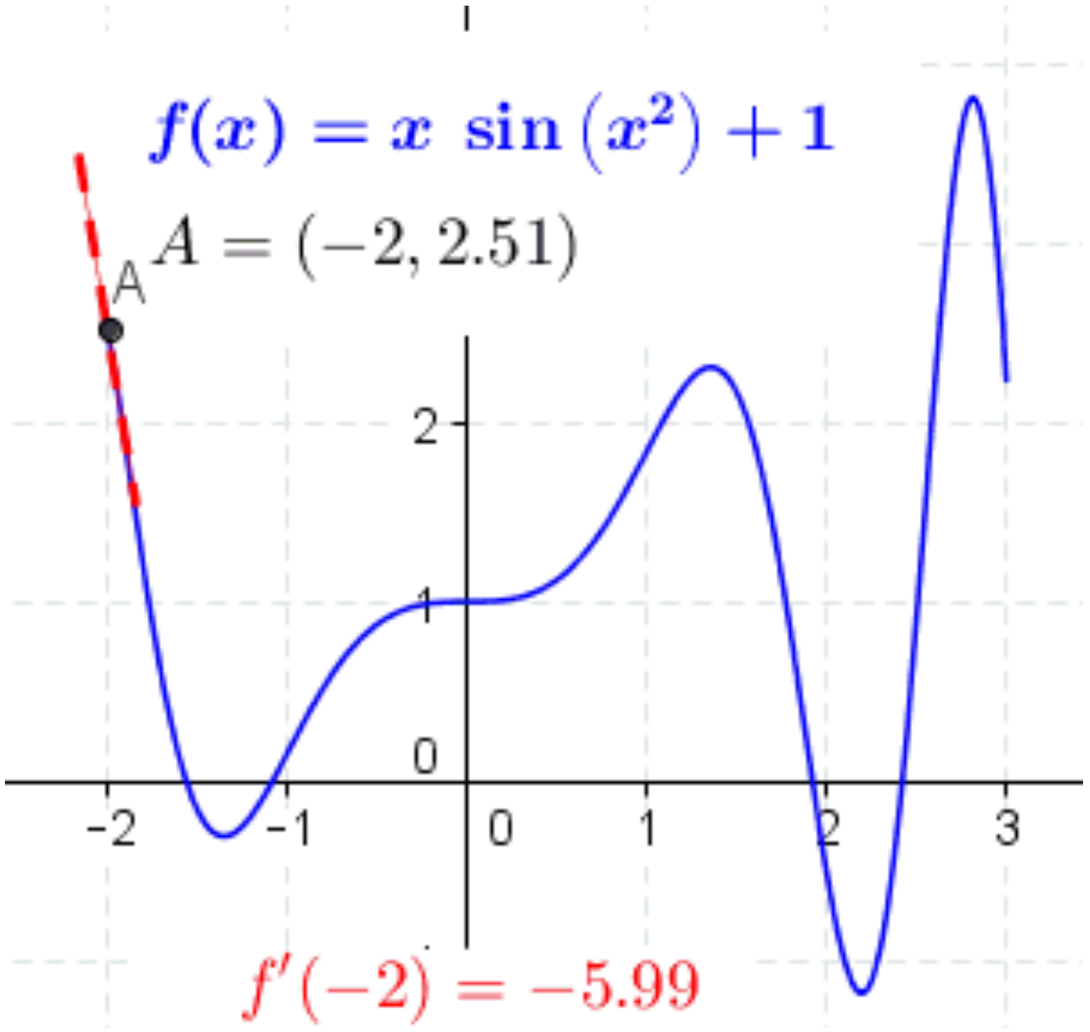
  - The **derivative** of a function at a certain point, gives the **rate** or the **speed** of which this function is changing its values at this point.

Step 4- Differentiation



The graph of a function and a tangent line to that function, drawn in red.
The slope (pendiente) of the tangent line is equal to the derivative of the function at the marked point.

Step 4- Differentiation



The **derivative** of a function at a certain point, gives the **rate** or the **speed** of which this function is changing its values at this point.

Step 4- Differentiation

- In order to see the **effect of the derivative**:
  - **How much the total error will change if we change the internal weight of the neural network with a certain small value δW?**
    - E.g. δW=0.0001

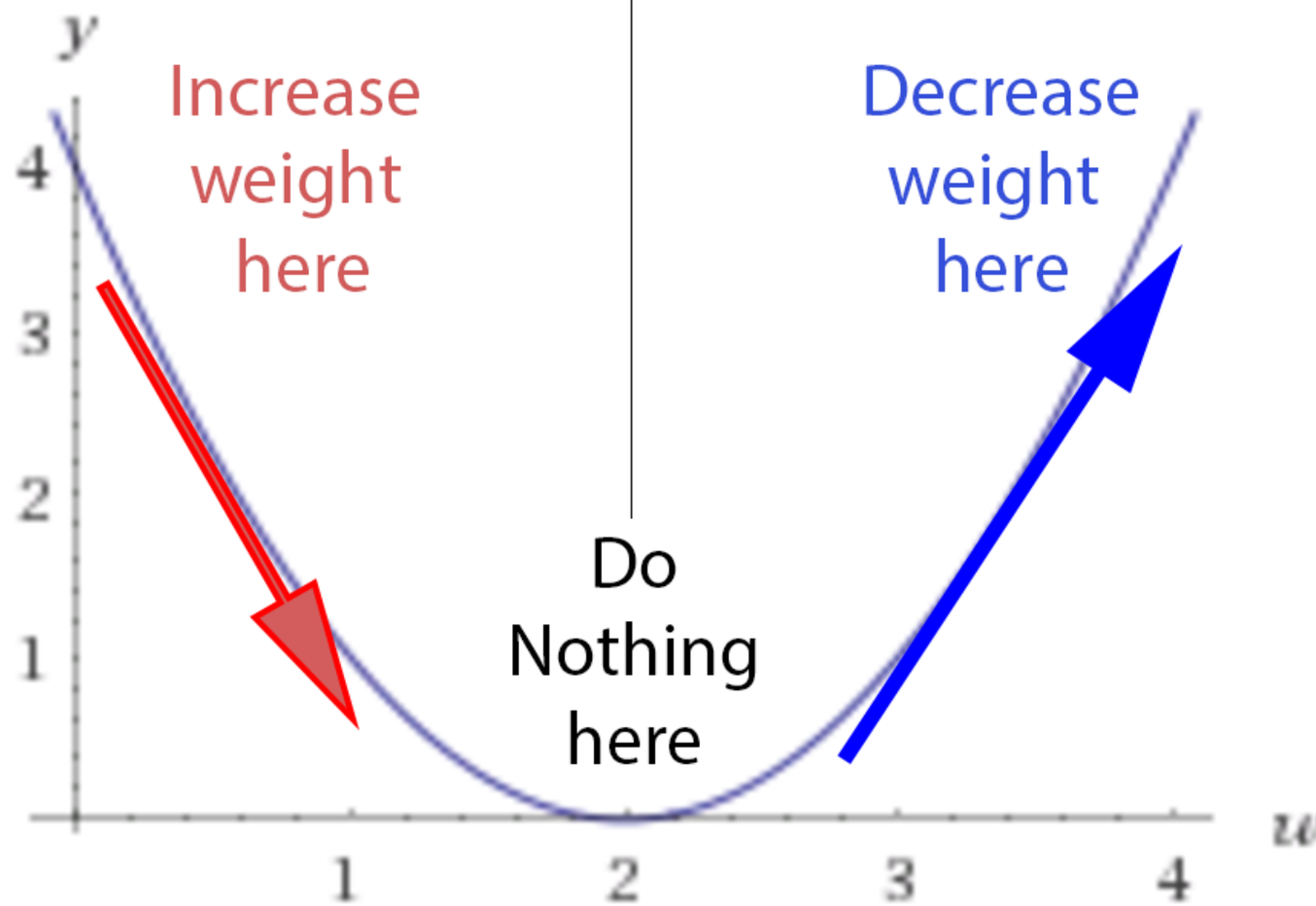
Step 4- Differentiation

Input	Output	W=3	rmse(3)	W=3.0001	rmse
0	0	0	0	0	0
1	2	3	1	3.0001	1.0002
2	4	6	4	6.0002	4.0008
3	6	9	9	9.0003	9.0018
4	8	12	16	12.0004	16.0032
Total:	-	-	30	-	30.006

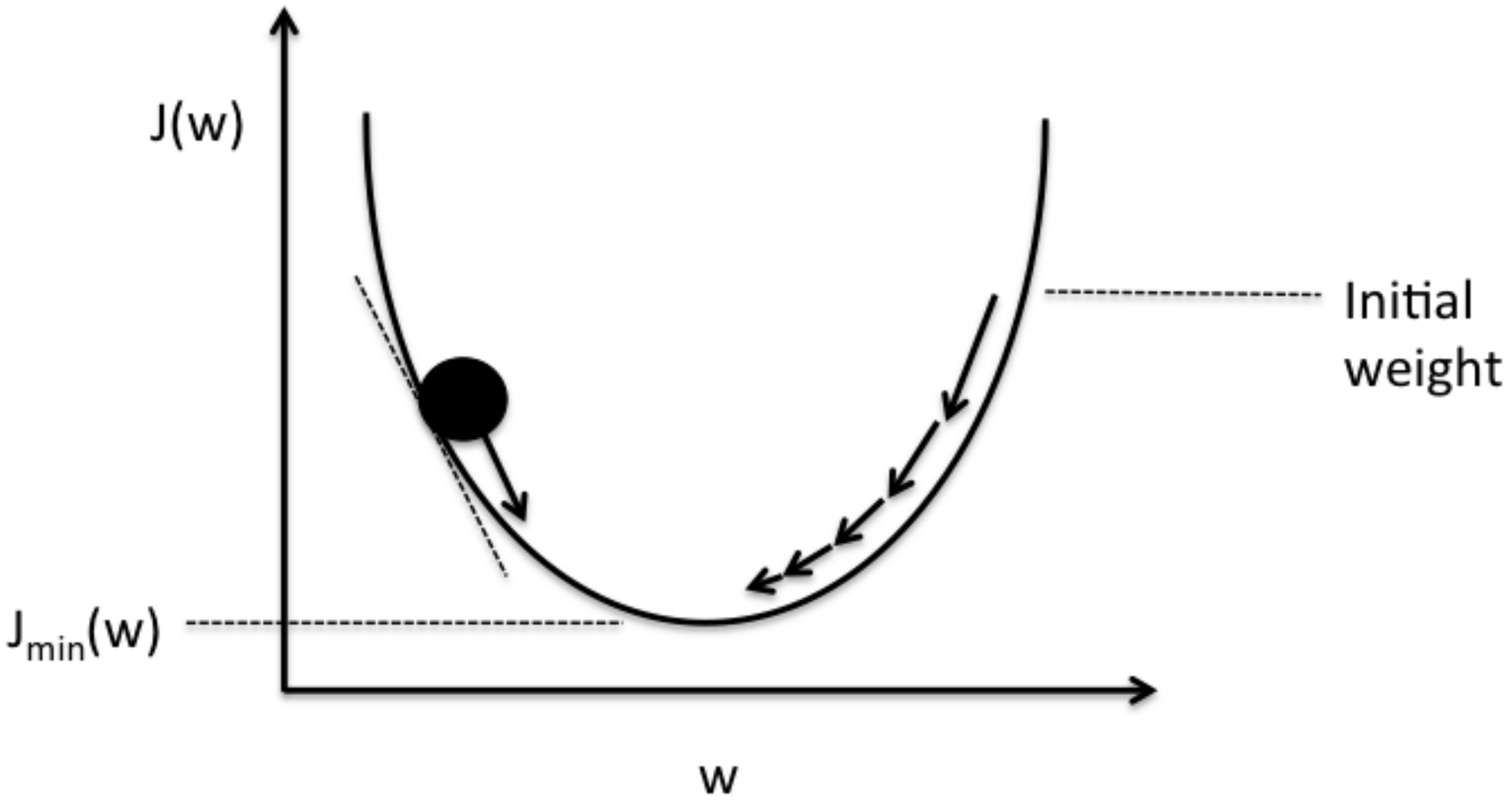
Step 4- Differentiation

Here is what our **loss function** looks like:

- If w=2, we have a loss of 0, since the neural network actual output will fit perfectly the training set.
- If w<2, the derivative is negative, meaning that an increase of weight will decrease the loss function.
- At w=2, the loss is 0 and the derivative is 0, we reached a perfect model, nothing is needed.
- If w>2, the derivative is as well positive, meaning that any more increase in the weight, will increase the losses even more!!



Step 4- Differentiation



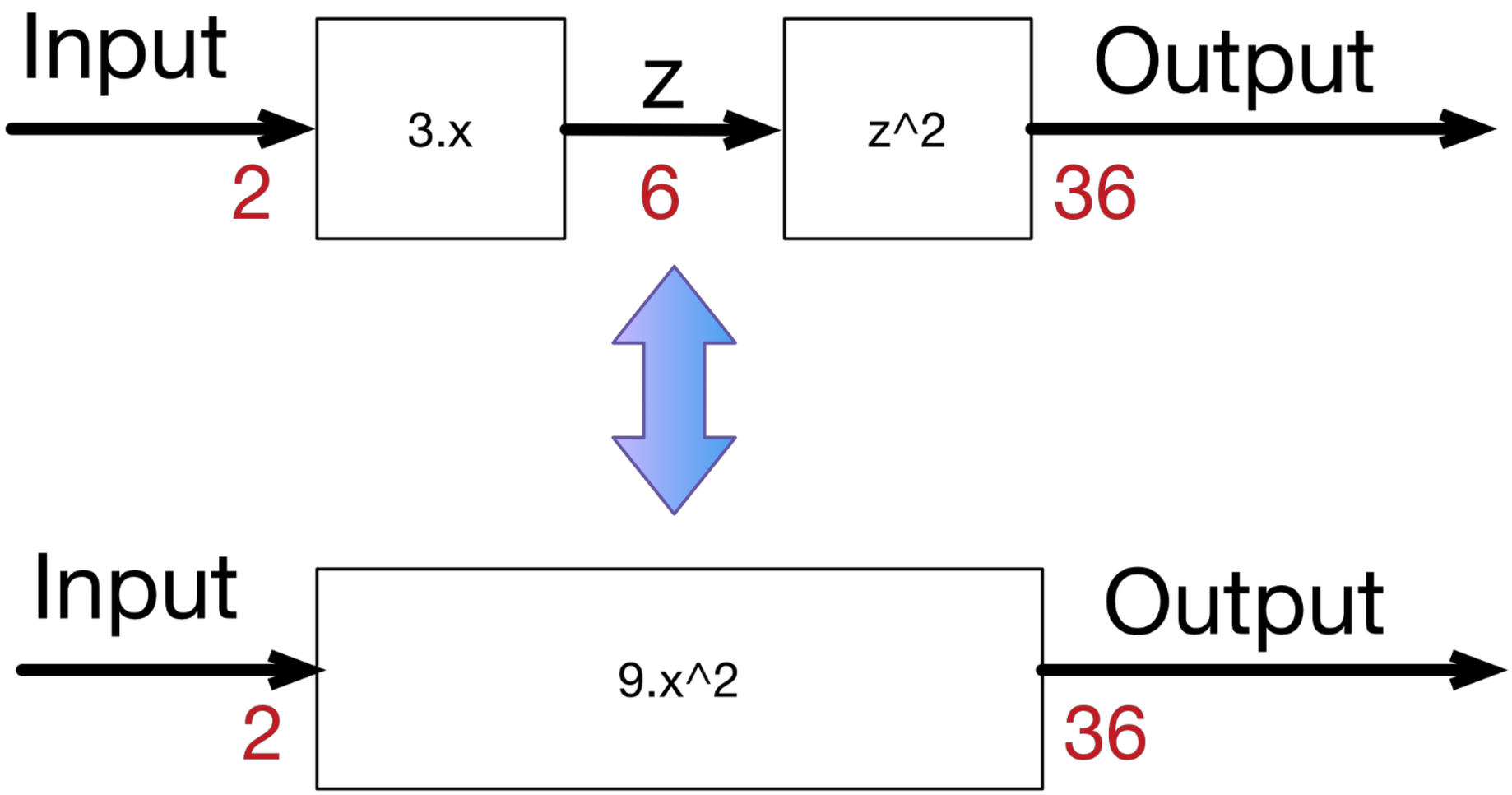
Schematic of gradient descent.

Step 5- Back-propagation

- We **used only one layer** inside the neural network between the inputs and the outputs.

  - In many cases, more layers are needed.

Step 5- Back-propagation



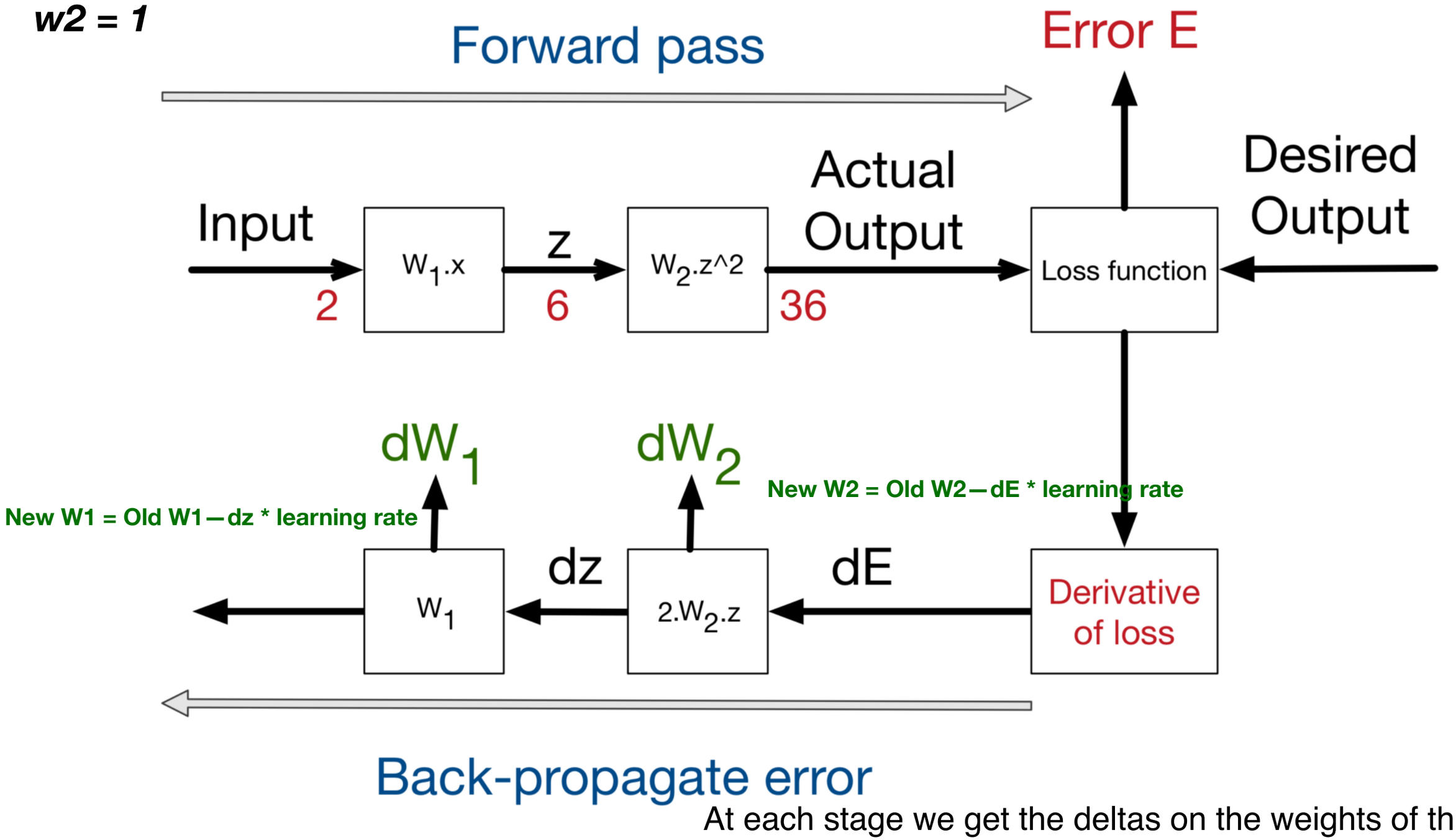
However in most cases **composing the functions** is very hard. Plus for every composition one has to calculate the dedicated derivative of the composition (which is not at all scalable and very error prone).

Step 5- Back-propagation

- In order to solve the problem, luckily for us, **derivative** is **decomposable**, thus can be **back-propagated**.
    - We have the starting point of errors, which is the loss function
    - We know how to derivate the loss function
    - If we know how to derivate each function from the composition, we can propagate back the error from the end to the start

Step 5- Back-propagation

x = 2
w1 = 3
w2 = 1



At each stage we get the deltas on the weights of this stage.

Step 6- Weight update

- **Delta Rule:**
  - **New weight = old weight—Derivative Rate * learning rate**
    - The learning rate is introduced as a constant (usually very small), in order to force the weight to get updated very smoothly and slowly (to avoid big steps and chaotic behavior).

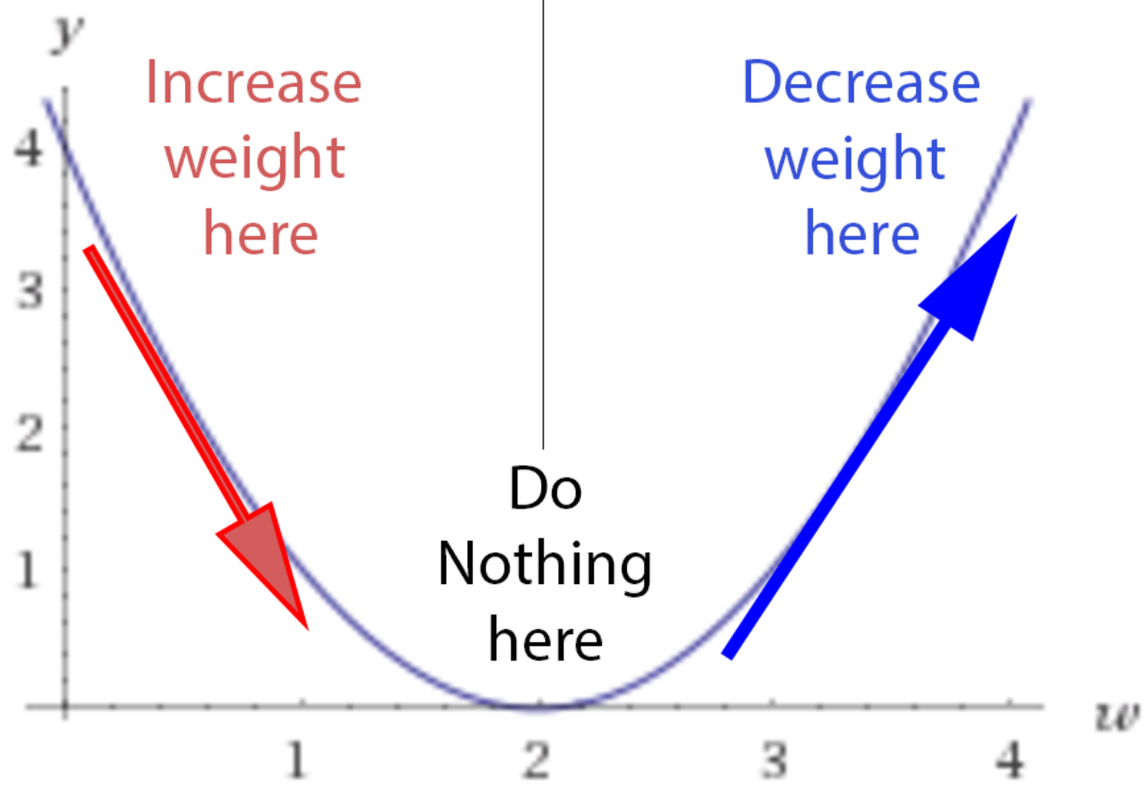
Step 6- Weight update

- **New weight = old weight— Derivative Rate * learning rate**

  - If the derivative rate is positive, it means that an increase in weight will increase the error, thus the new weight should be smaller.

  - If the derivative rate is negative, it means that an increase in weight will decrease the error, thus we need to increase the weights.

  - If the derivative is 0, it means that we are in a stable minimum. Thus, no update on the weights is needed -> we reached a stable state.



Step 7- Iterate until convergence

- Since we update the weights with a small delta step at a time, it will take several iterations in order to learn.

- This is very similar to genetic algorithms where after each generation we apply a small mutation rate and the fittest survives.

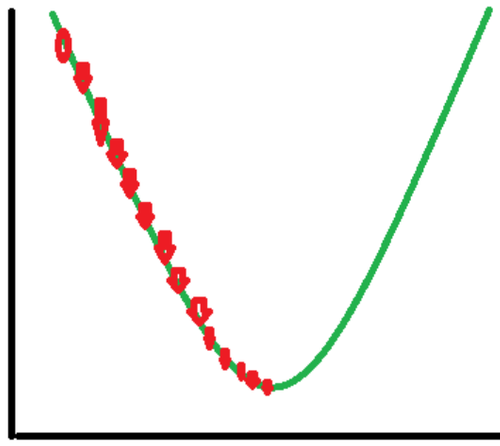
- In neural network, after each iteration, the gradient descent force updates the weights towards less and less global loss function.

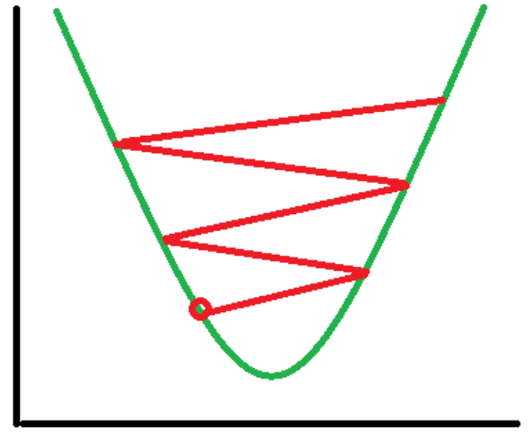
Step 7- Iterate until convergence

- The similarity is that the delta rule acts as a mutation operator, and the loss function acts a fitness function to minimize.
- The difference is that in genetic algorithms, the mutation is blind. Some mutations are bad, some are good, but the good ones have higher chance to survive.
    - The weight update in NN are however smarter since they are guided by the decreasing gradient force over the error.

Step 7- Iterate until convergence

- **How many iterations are needed to converge?**
  - This depends on how strong the learning rate we are applying. High learning rate means faster learning, but with higher chance of instability.



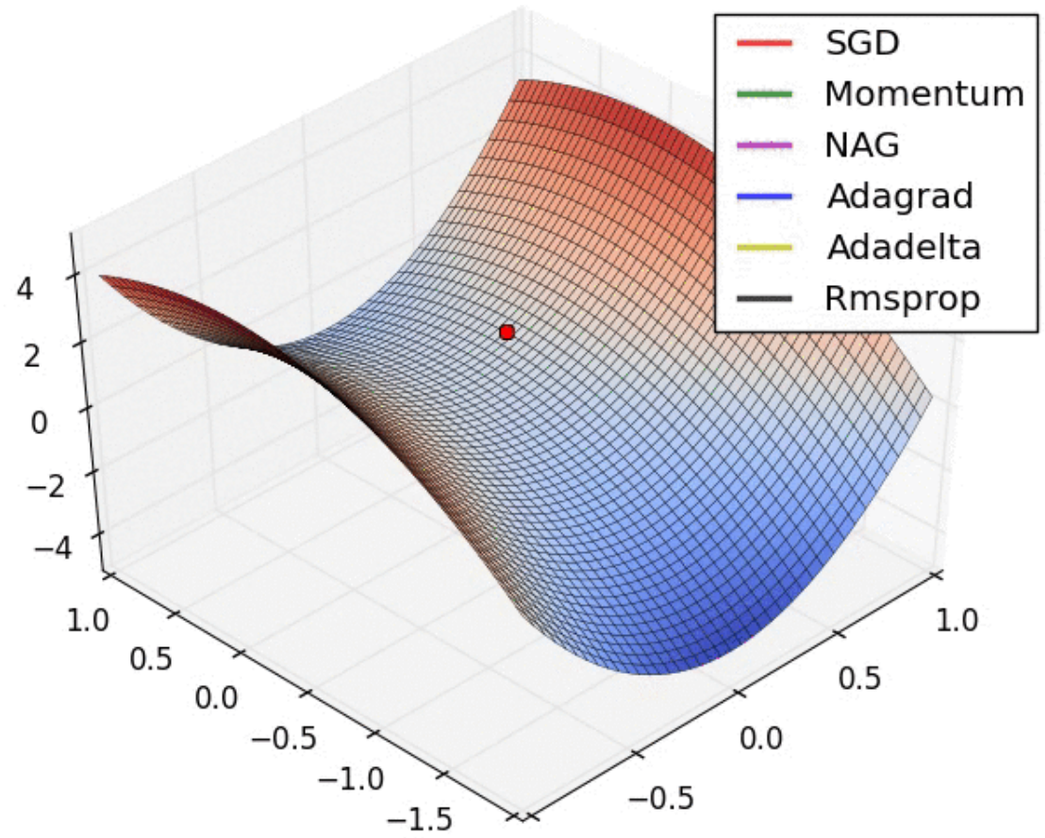


Step 7- Iterate until convergence

- **How many iterations are needed to converge?**

  - It depends as well on the meta-parameters of the network (how many layers, how complex the non-linear functions are). The more it has variables the more it takes time to converge, but the higher precision it can reach.

  - It depends on the optimization method use, some weight updates rule are proven to be faster than others.



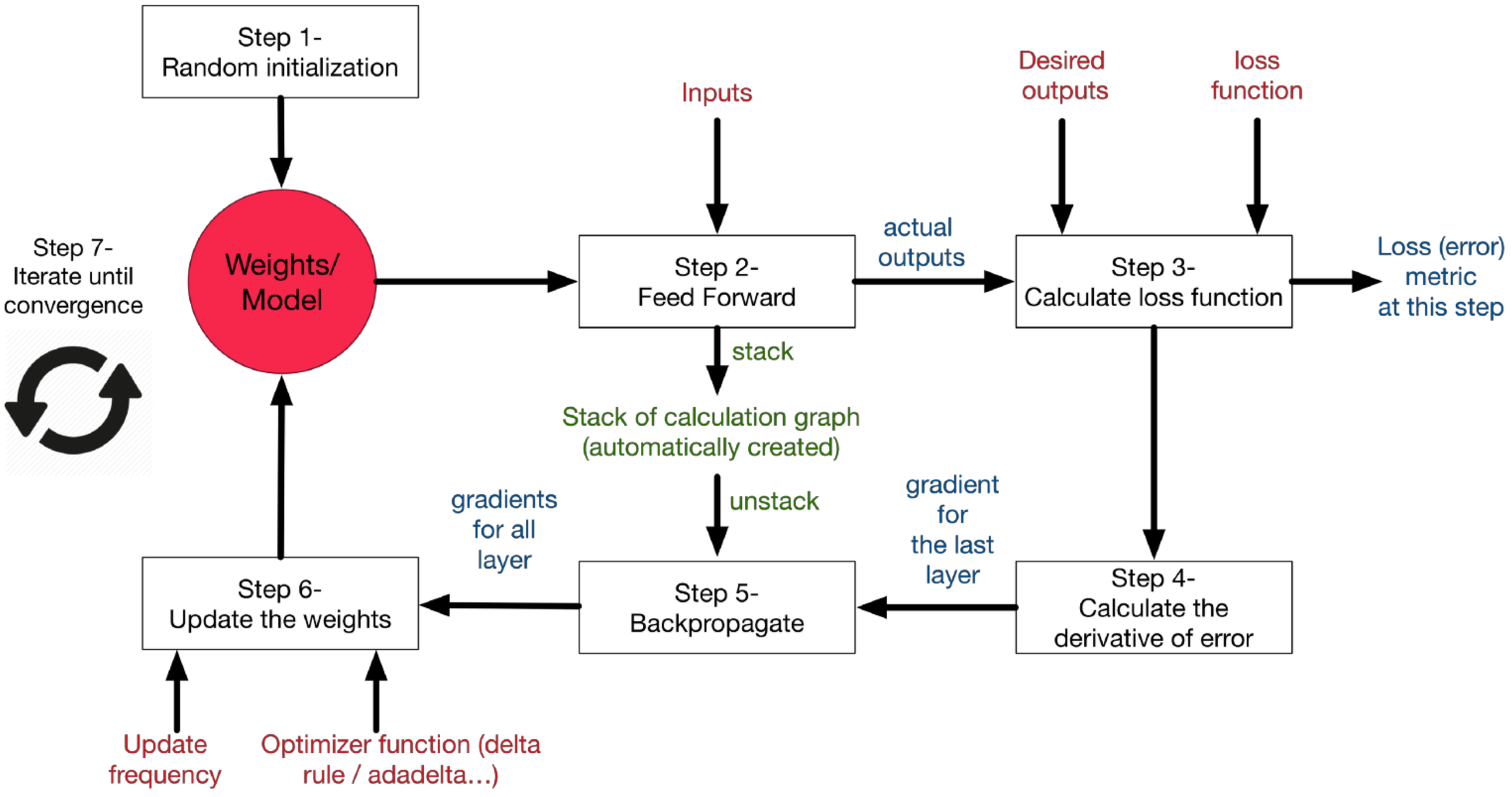
Step 7- Iterate until convergence

- **How many iterations are needed to converge?**

  - It depends on the random initialization of the network. Maybe with some luck you will initialize the network with **W=1.99** and you are only one step away from the optimal solution.

  - It depends on the quality of the training set. If the input and output has no correlation between each other, the neural network will not do magic and can’t learn a random correlation.

Overall Picture



Overall Picture

- https://google-developers.appspot.com/machine-learning/crash-course/backprop-scroll/