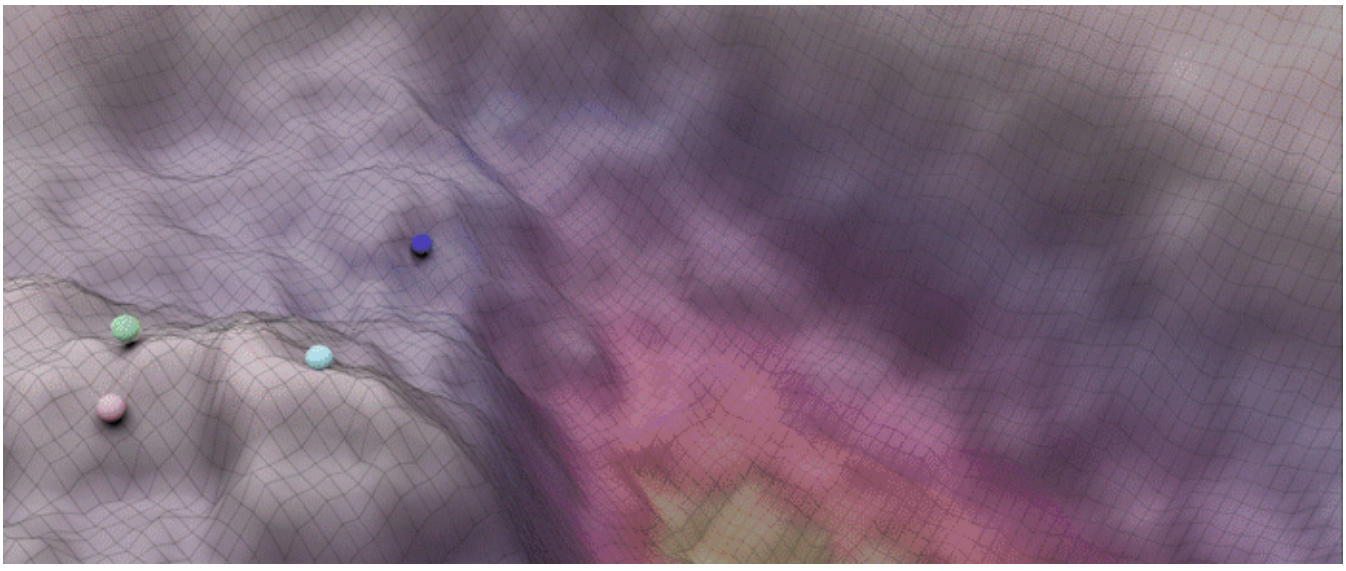
Javier Ideami Follow
Feb 8, 2019 · 21 min read · ✨ · Listen

Coding a 2 layer neural network from scratch in Python

In the second part of this series: code from scratch a neural network. Use it to predict malignant breast cancer tumors



In part 1 of this article, we understood the architecture of our 2 layer neural network. **Now it's time to build it!** In parallel, we will explore and understand in depth the foundations of deep learning, back-propagation and the gradient descent optimization algorithm.

And so, we begin! **First we import some standard Python libraries. Numpy** will help us with linear algebra and array functionality. **Pandas** dataframes will be really handy when we import and prepare our data. And **Matplotlib** will help us do some cool charts. Finally, **sklearn** helps us normalize our data and display useful graphs, such as confusion matrices.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import preprocessing
from sklearn.preprocessing import MinMaxScaler
from sklearn import metrics
from sklearn.metrics import confusion_matrix
import itertools
```

Next, we create **a Python class that setups and initializes our network.**

```
class dlnet:
    def __init__(self, x, y):
        self.X=x
        self.Y=y
        self.Yh=np.zeros((1,self.Y.shape[1]))

        self.L=2
        self.dims = [9, 15, 1]

        self.param = {}
        self.ch = {}
        self.grad = {}
```

```
        self.loss = []
        self.lr=0.003
        self.sam = self.Y.shape[1]
```

We first **name our class** (dlnet), and define its init method. The **init method** is **executed the first time** we instantiate the class. It is responsible for creating the structure of the network and the methods that will control it.

We then **create a series of class variables that will hold key data of the network.**

A lot of these variables will hold matrices. Remember that we will accelerate our computations by combining multiple calculations through the use of matrices. As I pointed out before, if you wish to refresh your linear algebra, check the amazing YouTube videos of **3Blue1Brown** and specifically his Essence of Linear Algebra Series.

- **X**: Holds our input layer, the data we give to the network. We initialize it with the value of the variable **x,** which is passed to the network when we create it. **X** is a matrix that has as many rows as features has our data (more about this later), and as many columns as samples we have available to train the network.
- **Y**: Holds our desired output, which we will use to train the network. We initialize it with the value of the variable **y**, which is passed to the network when we initialize it.
- **Yh**: Holds the output that our network produces. It should have the same dimensions than **Y**, our desired target values. We initialize it to zero.
- **L**: It holds the number of layers of our network, 2.
- **dims**: Next, we define the number of neurons or units in each of our layers. We do this with a numpy array. The first component of the array is our input (which is not counted as a layer of the network). Our input will have 9 units because, as we will see in a bit, our data-set will have 9 useful features. Next, the first layer of the neural network will have 15 neurons, and our second and final layer will have 1 (the output of the network).
- **param**: A Python dictionary that will hold the W and b parameters of each of the layers of the network.

- **ch**: a cache variable, a python dictionary that will hold some intermediate calculations that we will need during the backward pass of the gradient descent algorithm.

Finally, we declare three more parameters.

- **lr**: Our learning rate. This sets the speed at which the network will learn.
- **sam:** The number of training samples we have.
- **loss**: An array where we will store **the loss value of the network** every x iterations. The loss value expresses the difference between the predicted output of our network and the target one.

Notice that last element, **the loss,** because **it's crucial.** What is that loss value? It all has to do with the process of training our network to learn that mystery function.

We will go onto that very soon, but first, let's define the function **nInit**, which will initialize with random values the parameters of our network.

```
def nInit(self):
        np.random.seed(1)
        self.param['W1'] = np.random.randn(self.dims[1],
self.dims[0]) / np.sqrt(self.dims[0])
        self.param['b1'] = np.zeros((self.dims[1], 1))
        self.param['W2'] = np.random.randn(self.dims[2],
self.dims[1]) / np.sqrt(self.dims[1])
        self.param['b2'] = np.zeros((self.dims[2], 1))
        return
```

When we multiply matrices, as in the product **W1 X** and **W2 A1** , the dimensions of those matrices have to be correct in order for the product to be possible. That's why it's essential to set the dimensions of our weights and biases matrices right.

- **W1**: The number of rows is the number of hidden units of that layer, dims[1], and the number of columns is the number of features/rows of the previous layer (in this case X, our input data), dims[0].
  b1: Same number of rows as W1 and a single column.
  W2: The number of rows is the number of hidden units of that layer, dims[2], and the number of columns is again the number of rows of the input to that

layer, dims[1].

b2: Same number of rows as W2 and a single column.

Now, let's define within the class **a function that will perform the computation** at each unit of each layer in our network. We will call it **forward** because it will take the input of the network and pass it forwards through its different layers until it produces an output.

We also need to declare the **Relu** and **Sigmoid functions** that will compute the non-linear activation functions at the output of each layer.

```
def Sigmoid(Z):
    return 1/(1+np.exp(-Z))

def Relu(Z):
    return np.maximum(0,Z)

def forward(self):
        Z1 = self.param['W1'].dot(self.X) + self.param['b1']
        A1 = Relu(Z1)
        self.ch['Z1'],self.ch['A1']=Z1,A1

        Z2 = self.param['W2'].dot(A1) + self.param['b2']
        A2 = Sigmoid(Z2)
        self.ch['Z2'],self.ch['A2']=Z2,A2

        self.Yh=A2
        loss=self.nloss(A2)
        return self.Yh, loss
```

The **Relu** and **Sigmoid** functions declare the activation computations. And the forward function performs the computations we have described earlier.

We multiply the weights of the first layer by the input data and add the first bias matrix , **b1**, to produce **Z1**. We then apply the **Relu** function to **Z1** to produce **A1**.

Next, we multiply the weight matrix of the second layer by its input, **A1** (the soutput of the first layer, which is the input of the second layer), and we add the second bias matrix, **b2**, in order to produce **Z2**. We then apply the Sigmoid function to **Z2** to produce **A2**, which is in fact **Yh**, the output of the network.

That was it! **We just ran our input data through the network and produced Yh,** an output.

The logical next step is to find out how good our result was.

For that, we can compare the output we have produced to the output we should have obtained: **Yh and Y**. To compute that, we will add a final function to the network, **the loss function.**

Nothing to lose, hopefully

There are many kinds of loss functions. The **objective of the loss function** is to **express how far from the intended target our result was**, and to average that difference across all the samples we have used to train the network.

One of the simplest loss functions used in deep learning is **MSE,** or **mean square error**.

```
squared_errors = (self.Yh - self.Y) ** 2
self.Loss= np.sum(squared_errors)
```

The **MSE** loss function calculates the difference, the distance between our predicted and target outputs across all the samples we have used, and then squares that difference.

Finally, it adds up all those operations. Squaring the distances ensures that we produce an absolute distance value that is always positive.

MSE is a simple way to find out how far we are from our objective, how precise is so far the function computed by our network in terms of connecting our input data with our target outputs.

MSE is often used in regression challenges, when the output of the network is a continuous value, for example: a temperature value or the cost of a house.

However, in this article we will work on a different kind of challenge, a **binary classification challenge**, where our output will be either 0 or 1 (0 meaning benign, 1 meaning malignant).

When working with classification challenges, there is a different loss function that works better at expressing the difference between our predicted output and our correct one. It is called **the Cross-Entropy Loss Function** and we will use it in our network.

We pick loss functions based on **how well they express the quality of our network's performance in relation to the specific kind of challenge** we are working on. Cross-entropy is a **great loss function for classification problems** (like the one we will work on) because **it strongly penalizes predictions that are confident and yet wrong** (like predicting with high confidence that a tumor is malign when in fact it is benign).

```
def nloss(self,Yh):
        loss = (1./self.sam) * (-np.dot(self.Y,np.log(Yh).T) -
np.dot(1-self.Y, np.log(1-Yh).T))
        return loss
```

As you can see above, at the end of the **forward** function we call this **nloss** method (which computes the loss), and then store the resulting loss value in the **loss** array. This will later allow us to plot and visually understand how the loss value changes during the training of the network.

Believe or not, **we have already created almost half of all the code we will need.** But **now we arrive to the crucial point.**

We have computed an output **Yh**, and **calculated the loss:** how far we are from the intended output, **Y**. The question now is: **how can we improve that result Yh, and what does it mean to improve it?**

To improve the result, **we need to get that loss value to decrease.** The lower our loss value, the lower the distance between our target and predicted outputs (**Y** and **Yh**), and the better our network will perform.

Let's recap. We are producing two outputs at the network:

- **Yh**, the result of the computations of the network.
- **The Loss**, the distance between **Yh and Y**.

And it is from that objective, **from the objective of minimizing the loss,** of minimizing the distance between our predicted and correct outputs, that **the training process of the network is born.**

I need a Gym

At this stage, we have performed a forward pass, obtained our output **Yh**, and then calculated our loss, our error, the distance between our predicted and correct output (**Yh and Y**).

The next logical step is to **change slightly the values of the parameters of our network,** of our weights and biases, **and perform the forward pass again to see if our loss hopefully decreases.** Therefore, the training process would look like this:

- We initially **set weights and biases** to random values.
- We run the input data forwards through the network and **produce a result: Yh.**
- We **calculate the loss, the distance between Yh and Y**, between our predicted and target outputs.
- **If it's not good enough**, we **change slightly our weights and biases and run again** the input data through the network to see if hopefully our loss has improved, if maybe now is low enough. If not, **we keep repeating the same process** (probably for thousands of years).

You got it, **that's not efficient** at all.

Now, think about this, to modify our weights and biases by a small amount, we can do one of two things with each one of them:

- We can increase them a bit
- We can decrease them a bit.

There must be a way, in which, **taking our loss as a starting point,** we can **calculate if we should increase them or decrease them in order to minimize** such loss.

Enter Calculus, and enter the mighty derivative, the gradient. Let's explore in very simple ways how the derivative works. If you want to go deeper I have you covered again with 3Blue1Brown Essence of Calculus series.

Hello gradient

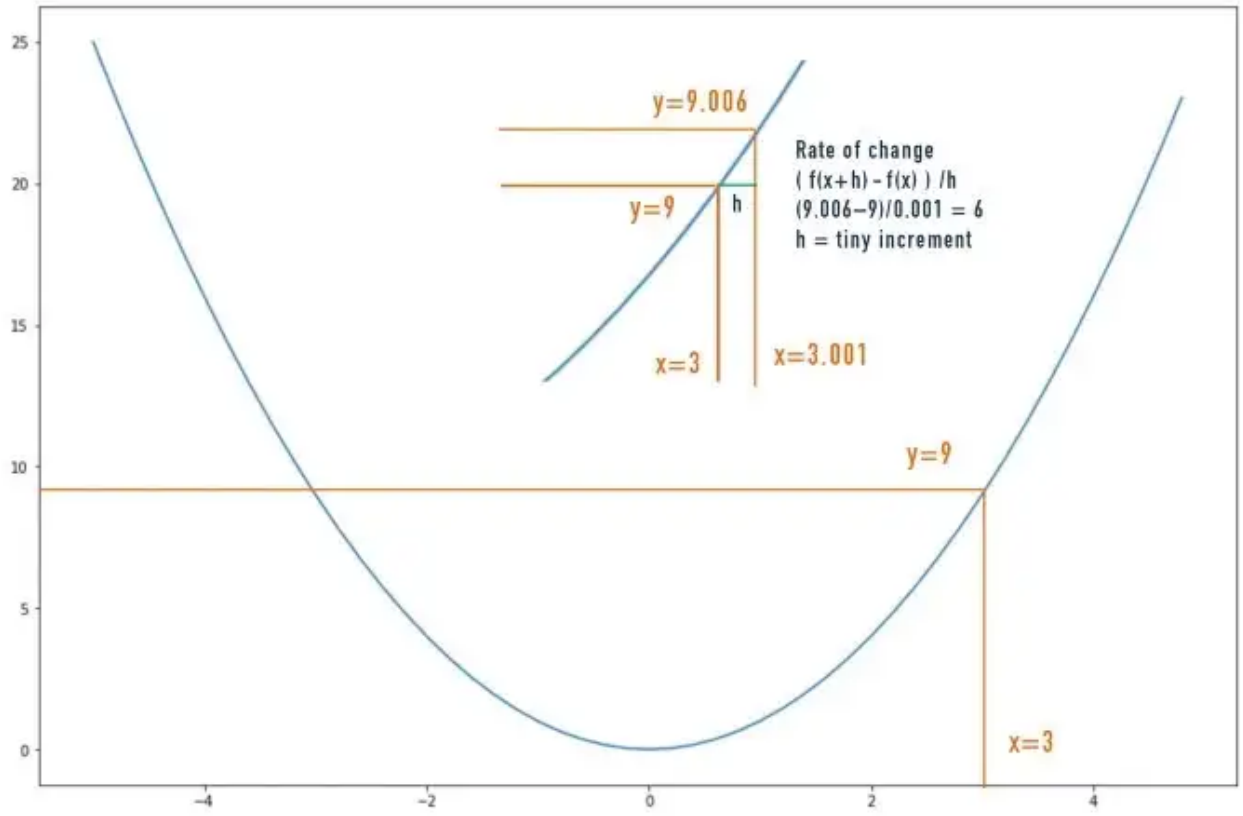
In order to understand in what direction we should change our weights and biases, it would be great to **understand what impact a small change** in each of those weights and biases **has in our final loss.**

And **we can use derivatives for this**, partial derivatives to be precise. Because a partial derivative **is going to tell us what impact a small change** on a specific parameter, say, **W1**, has on our final loss.

And with that info, **we will be able to decide in what direction we want to modify W1** in order to decrease the loss.

Let's first **refresh the intuition** of the derivative.

Think of **the function x to the power of 2:** x**2



At x=3, y=9. Let's focus on that point and find the derivative, the rate of change at x=3.

To do that, we will study **what happens to y** when we **increase x by a tiny amount,** which we call **h.** That tiny amount eventually converges to 0 (the limit), but for our purposes we will consider it to be a really small value, say 0.001.

So, when **x=3+0.001, what's the value of y?** y=(3.001)**2= **9.006**

Therefore **x=3.001 becomes y=9.006 when increasing x by 0.001.**

And the rate of change (the derivative) is the difference between the new f(x+h) and the previous f(x), divided by that tiny increment h: (9.006–9)/0.001 = **6.**

What is 6 telling us?

6 is telling us that in this function x**2, at x=3, the rate of change is positive and has a strength of 6. It is telling us that at that point, if we increase x a bit, y will change in a positive way and with a strength of "6 times more". Basically**, that the 0.001 increment at the input will become a 0.006 increment at the output.**

So we see that we can calculate a derivative by hand easily following this method:

Derivative = (f(x+h) — f(x)) /h

So **what about at x=-2**?
dx = f (-2 + 0.001) — f(-2)) / 0.001
dx =f (-1.999) — f(-2)) / 0.001
dx =3.996–4 / 0.001 = -4

At x=-2, the rate of change is negative, the function is moving down and with a strength of 4.

Calculating derivatives in this way **takes a long time** but thanks to the geniuses of math, it's easy to calculate them really fast by using differential equations, special equations that express the derivative of the original function. So, instead of having to calculate the derivative at each point, **a single equation can calculate it for us everywhere in that function automatically!**

Most, but not all equations, have a derivative that can be expressed with another equation.

The derivative of **x**2 i**s the function **2x.** And we will express the derivative with the letter **d** followed by the variable whose rate of change we are studying.

If dx =2x:

- when x is 3, dx equals 2*3 = 6
- when x is -2, dx equals 2*-2 = -4
- both match our calculations by hand. Using a differential equation is so much faster!

All right, so let's recap. **Thanks to the derivative we can understand in what direction the output of a function is changing at a certain point** when we modify a certain input variable, x in this case.

It would be great then if we could **use the derivative to understand how small changes to our weights and biases** impact the loss of the network.

- If we see that the **derivative of the loss in relation to a weight is positive**, it means that increasing the weight will make the loss increase. That means: do the opposite, **decrease the weight** to make the loss decrease.
- And if we find that the **derivative is negative**, it means that increasing the weight makes the loss decrease. That's what we want!. So then we proceed to **increase the value of the weight.**
- So, by looking at the derivative of the loss in relation to a parameter in our network, we can understand the impact that changing that parameter has on the loss of the network.
- And based on that, we can modify that parameter to move its influence in the direction that lowers the loss.

However, there is a little problem, **a final obstacle.**

Our **network is made of layers.** It could have 2 or 200 layers. And **we need to understand how changes** to all of our weights and biases **impact the loss at the end**

of the network.

Remember that our network is a series of functions chained together. If we want to calculate in a multi layer network how, for example, a change in W1 impacts the loss at the final output, **we need to somehow find a way to connect, to relate to each other, the different derivatives** that exist between W1 and the loss of the network at its end.

Could we maybe, somehow, chain them?

The chain rule

Yes, indeed. Calculus gives us something called **the chain rule of derivatives**, which really it's a pretty simple concept when we look at it in detail.

First of all, **a partial derivative** is a derivative that studies the change that occurs in a variable when we modify another variable.

To connect it all with the code that is coming:

- I will name, for example, the partial derivative of the **loss** in relation to the output **Yh** as **dLoss_Yh**
- That is: the derivative of the Loss function in relation to the variable Yh.
- Which means: **when we modify slightly Yh, what is the impact on the Loss?**

The **chain rule** tells us that to understand **the impact of the change of a variable on another**, when they are distant from each other, we can **chain the partial derivatives in between by multiplying them**.

It's time to talk about **the Back-Propagation algorithm** within a neural network, and in this case, specifically, in our 2 layer network.

Back-propagation makes use of the chain rule to find out to what degree changes to the different parameters of our network influence its final loss value.

Let's pick one of our parameters and understand the chain rule in action.

Say that we want to understand how small changes to **W1** will impact the Loss. All right, let's begin with the equation of the Loss:

Loss = -(Y Log Yh + (1-Y) Log (1-Yh))

Well, **W1** is not present in this equation, but **Yh** is. Let's proceed to calculate how **a change in Yh, our result, influences the loss**. And let's see if, after we do that, we can continue chaining derivatives until we arrive to **W1.**

To calculate this derivative, we look for **the derivative equation of the Loss function.** You can learn to quickly find the derivatives of all kinds of equations by refreshing your calculus a bit or looking them up online. In this case we find that:

dLoss_Yh = — (Y/Yh — (1-Y)/(1-Yh))

All right, one down. Now, remember, we want to **continue chaining derivatives until we arrive to W1.**

So let's see, **what's the next step backwards in our network, how did we produce Yh?**

Yh = sigmoid (Z2)

All right, great. W1 is still not there, but we got Z2. So let's find out what impact a change in Z2 has on Yh. For that we need to know the derivative of the sigmoid function, which happens to be:

dSigmoid = sigmoid(x) * (1.0 — sigmoid(x)).

To simplify the writing, we will represent that differential equation as **dSigmoid**. Therefore::

dYh_Z2 = dSigmoid(Z2)

At this stage, we can already chain (multiply) these 2 derivatives to find the derivative of the Loss in relation to Z2.

dLoss_Z2 = dLoss_Yh * dSigmoid(Z2)

Excellent, let's proceed. **How did we calculate z2?**

Z2 = W2 A1 + b2

Again, **W1** is still not there, but we got **A1.** Let's find out what impact a change on A1 has on Z2. Therefore:

dZ2_A1 = W2

And **we can chain that derivative to the previous 2** in order to get the total derivative between A1 and the loss of the network:

dLoss_A1 = **W2** * **dLoss_Z2**

As you can see, **we are chaining derivatives**, one after the other, until we arrive to W1, our target. So far we have moved from the Loss to Yh, from Yh to Z2 and from Z2 to A1.

Excellent. Let's continue. How did we produce A1?

A1 = Relu (Z1).

And W1 is still not there, but we got Z1. We need the derivative of Relu. The derivative of the Relu function is 0 when the input is 0 or less than 0, and 1 otherwise.

Again, to simplify the writing, we will express it as **dRelu**.

dA1_Z1=dRelu (Z1)

Great, let's chain again this latest derivative with all the previous ones to get the full derivative of the Loss in relation to Z1:

dLoss_Z1 = **dLoss_A1 * dRelu (Z1)**

Superb, we are approaching! How did we calculate Z1?

Z1 = W1 X+ b1

Yeah! W1 is there! We had missed you W1! It's so great to see you! :)

So exciting! **This will therefore be the final derivative:**

dZ1_W1 = X

And let's chain this latest derivative to all the previous ones:

dLoss_W1= X * dLoss_Z1

And that's it. We have calculated the derivative of the Loss in relation to our parameter W1. **That is, how much and in what direction the Loss changes when we modify slightly W1.**

Let's recap:

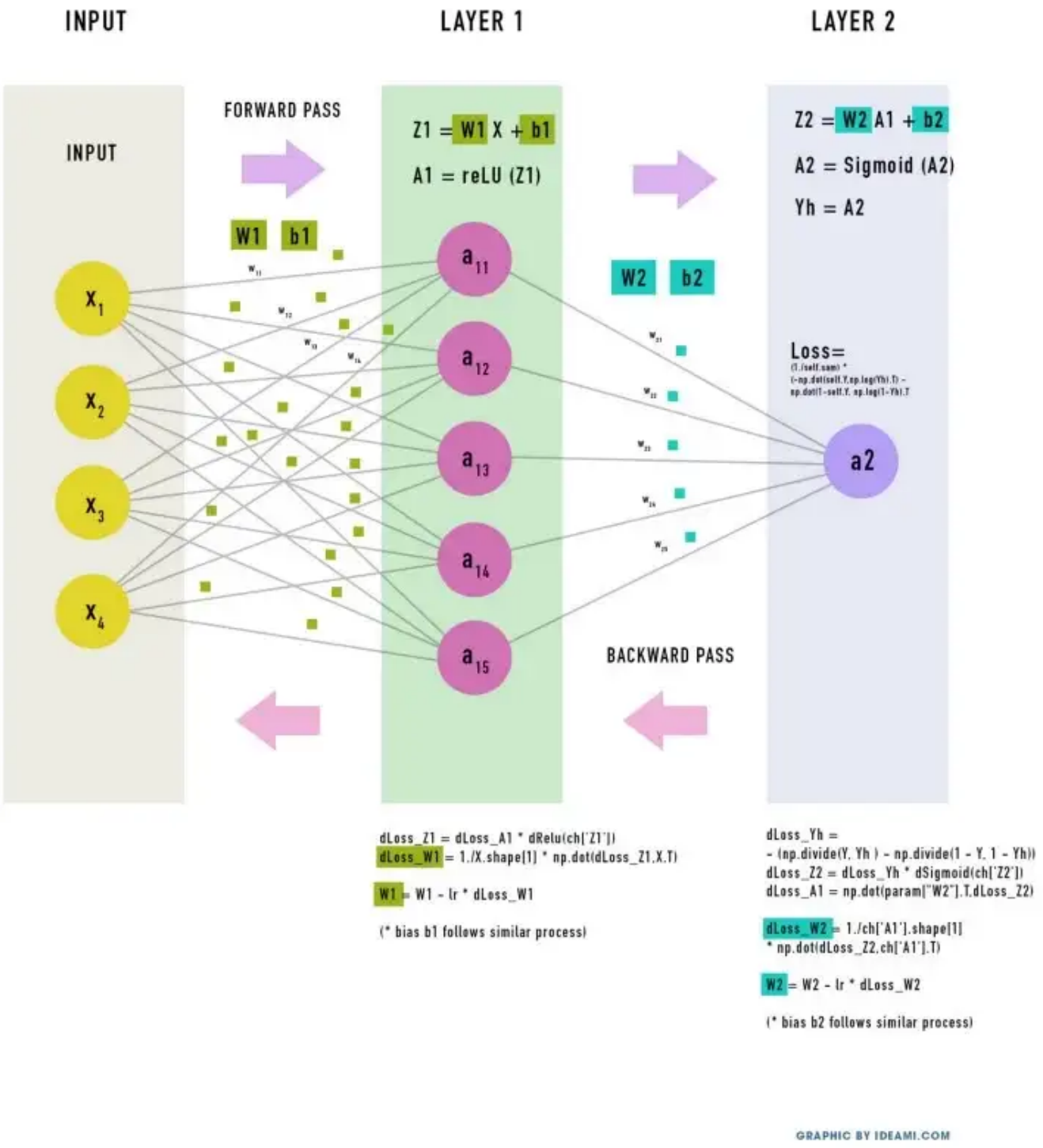
- We have started at the end of the network, at the loss value, and gradually chained derivatives until we arrived to W1.
- By the **chain rule**, we proceeded to one by one **multiply all those derivatives** together to find the final rate of change, **the impact that changes in W1 have on the Loss at the output of the network.**

In Python code, ordering things correctly to account for the way we multiply matrices, **the code of this chaining process is**:

dLoss_Yh = — (np.divide(self.Y, self.Yh) — np.divide(1 — self.Y, 1 — self.Yh))
dLoss_Z2 = **dLoss_Yh** * dSigmoid(self.ch['Z2'])
dLoss_A1 = np.dot(self.param["W2"].T,**dLoss_Z2**)
dLoss_Z1 = **dLoss_A1** * dRelu(self.ch['Z1'])
dLoss_W1 = 1./self.X.shape[1] * np.dot(**dLoss_Z1**,self.X.T)

Notice that at the last step, **we divide the result by the number of units of the layer,** so that the derivative in relation to each weight **W** is **scaled correctly at each unit.**

And that, what you have just seen, is **back-propagation**, or **the key ingredient of pretty much all deep learning processes.**



Let's breathe! That was the hardest bit of the entire article, from now on things get easier.

- By calculating the way changes to W1 impact the loss at the output, we can now decide how to modify W1 in order to decrease that loss.
- If the derivative is positive, it means that changes to W1 are increasing the loss, therefore: we will decrease W1 instead.

- If the derivative is negative, it means that changes to W1 decrease the loss, which is what we want, so: we will increase the value of W1.
- What we have done with W1, we will do in exactly the same way with W2 , b1 and b2.

In this way we produce the backwards pass, which becomes the back-propagation function of our python class. We also declare dRelu and dSigmoid, the derivatives of the Relu and Sigmoid functions, which are needed when we compute the back-propagation algorithm.

```
def dRelu(x):
    x[x<=0] = 0
    x[x>0] = 1
    return x

def dSigmoid(Z):
    s = 1/(1+np.exp(-Z))
    dZ = s * (1-s)
    return dZ

def backward(self):
        dLoss_Yh = - (np.divide(self.Y, self.Yh ) - np.divide(1 -
self.Y, 1 - self.Yh))

        dLoss_Z2 = dLoss_Yh * dSigmoid(self.ch['Z2'])
        dLoss_A1 = np.dot(self.param["W2"].T,dLoss_Z2)
        dLoss_W2 = 1./self.ch['A1'].shape[1] *
np.dot(dLoss_Z2,self.ch['A1'].T)
        dLoss_b2 = 1./self.ch['A1'].shape[1] * np.dot(dLoss_Z2,
np.ones([dLoss_Z2.shape[1],1]))

        dLoss_Z1 = dLoss_A1 * dRelu(self.ch['Z1'])
        dLoss_A0 = np.dot(self.param["W1"].T,dLoss_Z1)
        dLoss_W1 = 1./self.X.shape[1] * np.dot(dLoss_Z1,self.X.T)
        dLoss_b1 = 1./self.X.shape[1] * np.dot(dLoss_Z1,
np.ones([dLoss_Z1.shape[1],1]))

        self.param["W1"] = self.param["W1"] - self.lr * dLoss_W1
        self.param["b1"] = self.param["b1"] - self.lr * dLoss_b1
        self.param["W2"] = self.param["W2"] - self.lr * dLoss_W2
        self.param["b2"] = self.param["b2"] - self.lr * dLoss_b2
```

Within the backward function, after calculating all the derivatives we need for W1, b1, W2 and b2, we proceed, in the final lines, to update our weights and biases by subtracting the derivatives, multiplied by our learning rate.

Remember that the **learning rate** is a parameter that allows us to **set how fast the network learns**. We, therefore, modify our weights and biases by a quantity proportional to that learning rate.

All right, so far we have:

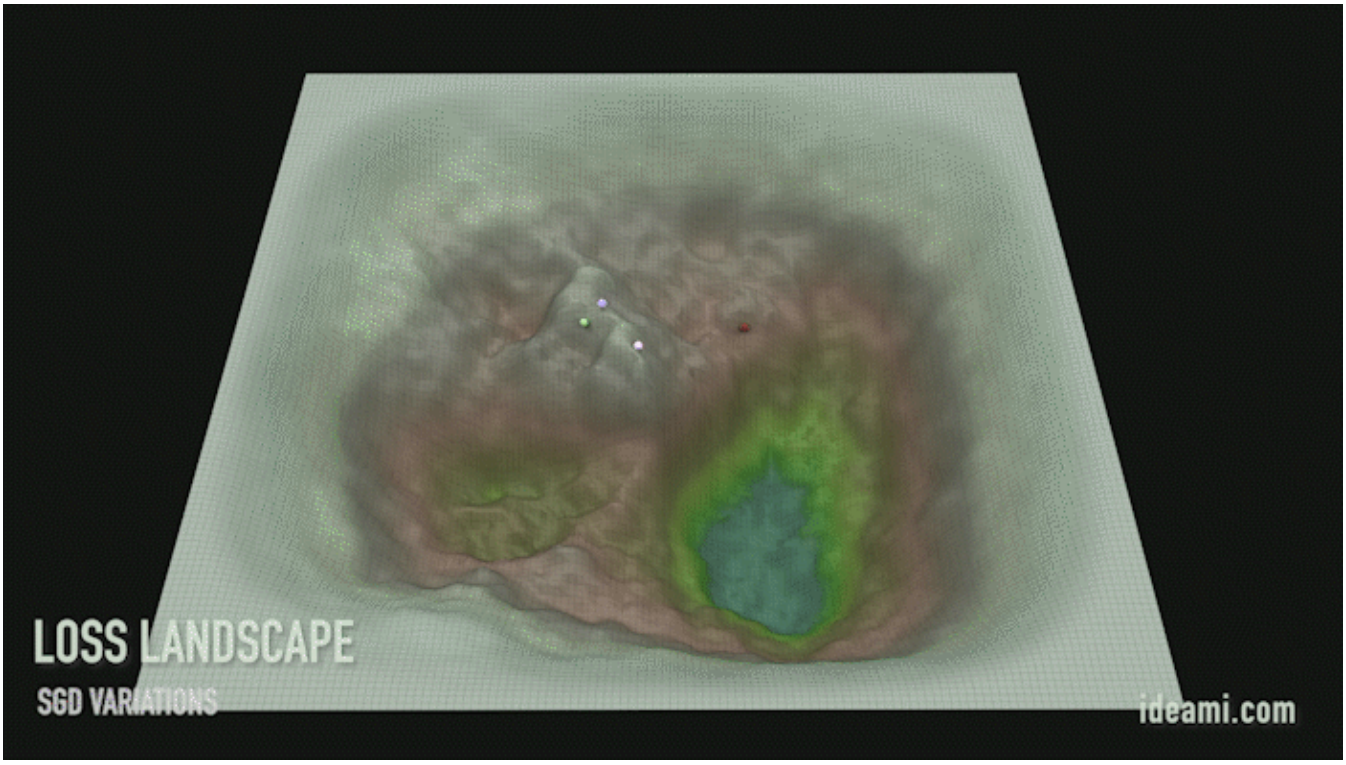
- Performed a forward pass
- Calculated the loss of the network
- Performed a backward pass and updated the parameters of our network (so that in the next forward pass the loss will decrease).

That is, in fact, **the Gradient Descent optimization algorithm**, the other piece of this fascinating puzzle that is training our neural network.

Time to return to the very first animation in this article.

Descending along the gradient

Let's look again at the first animation of the article.

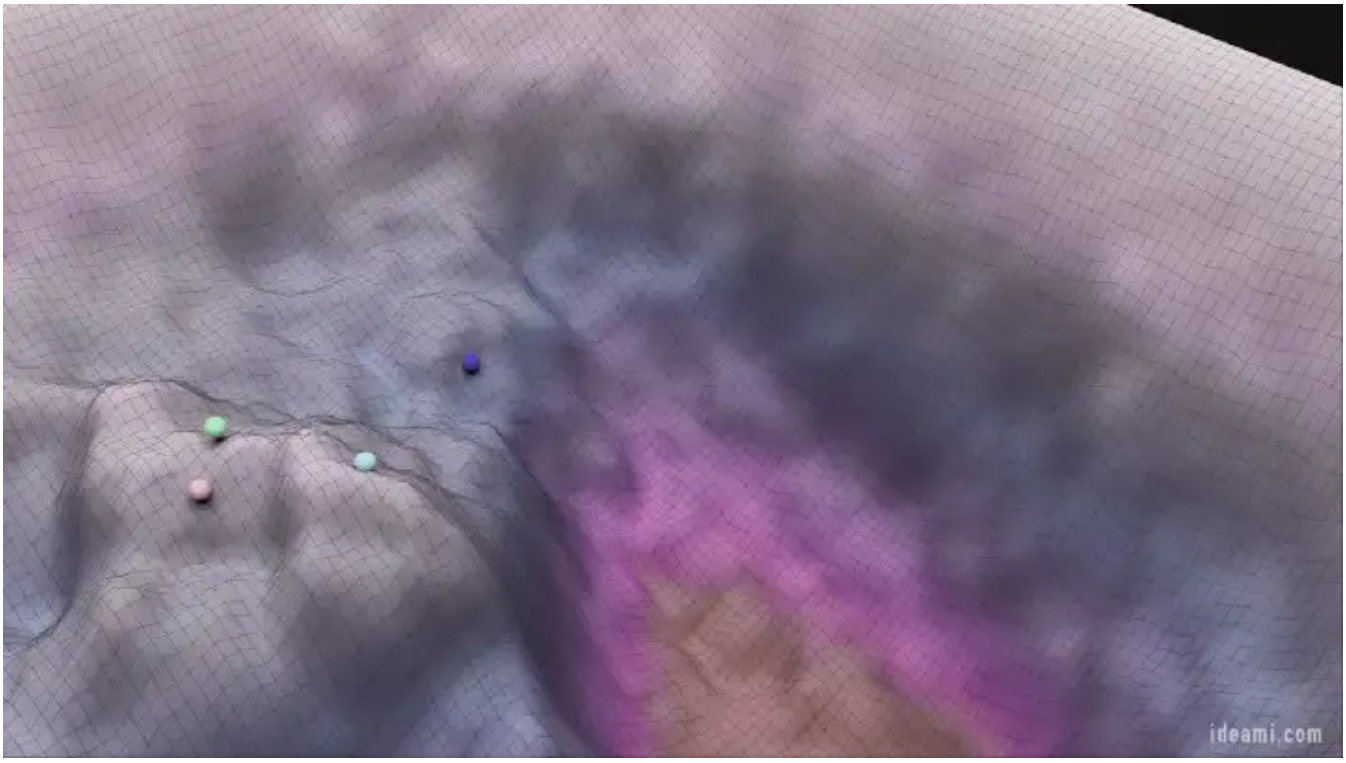


Navigating the Loss Landscape. Values have been modified and scaled up to facilitate visual contrast.

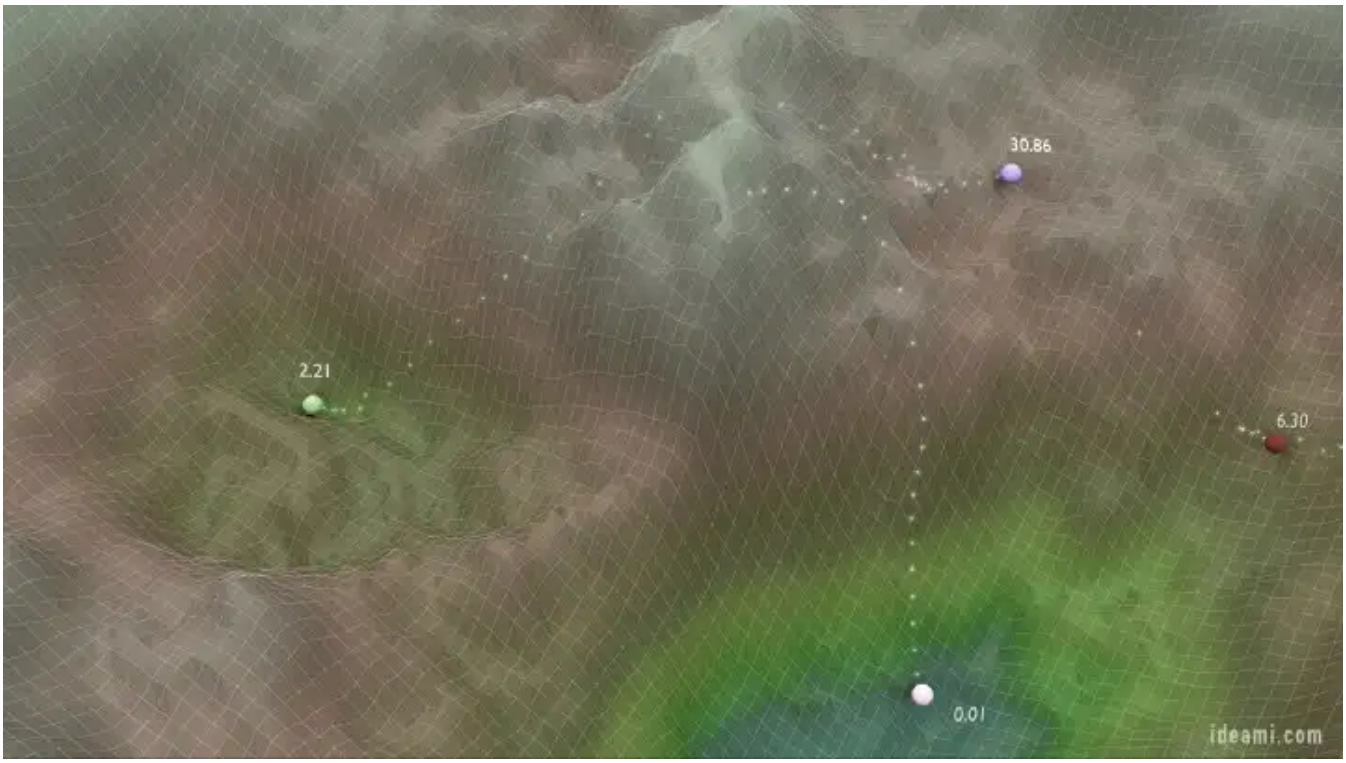
This is the **gradient descent optimization algorithm**, the cornerstone and most often used method to **gradually optimize the weights** of our network, so that eventually they will allow us to compute a function that accurately and efficiently connects our input data with our desired output.

Let's analyze what's happening at the animation, which represents key aspects of the **gradient descent algorithm.**

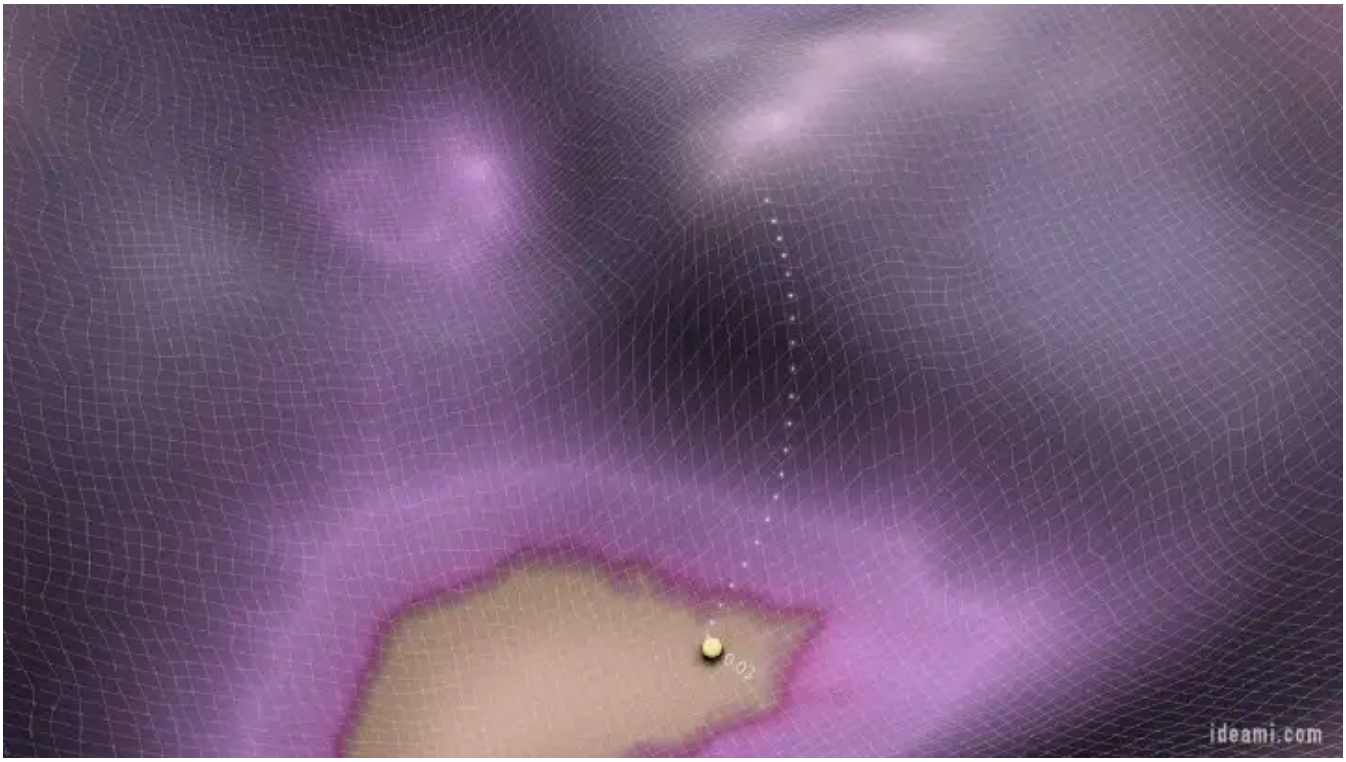
- We first initialize our weights and biases with random values. We **establish the initial state** of our training process.



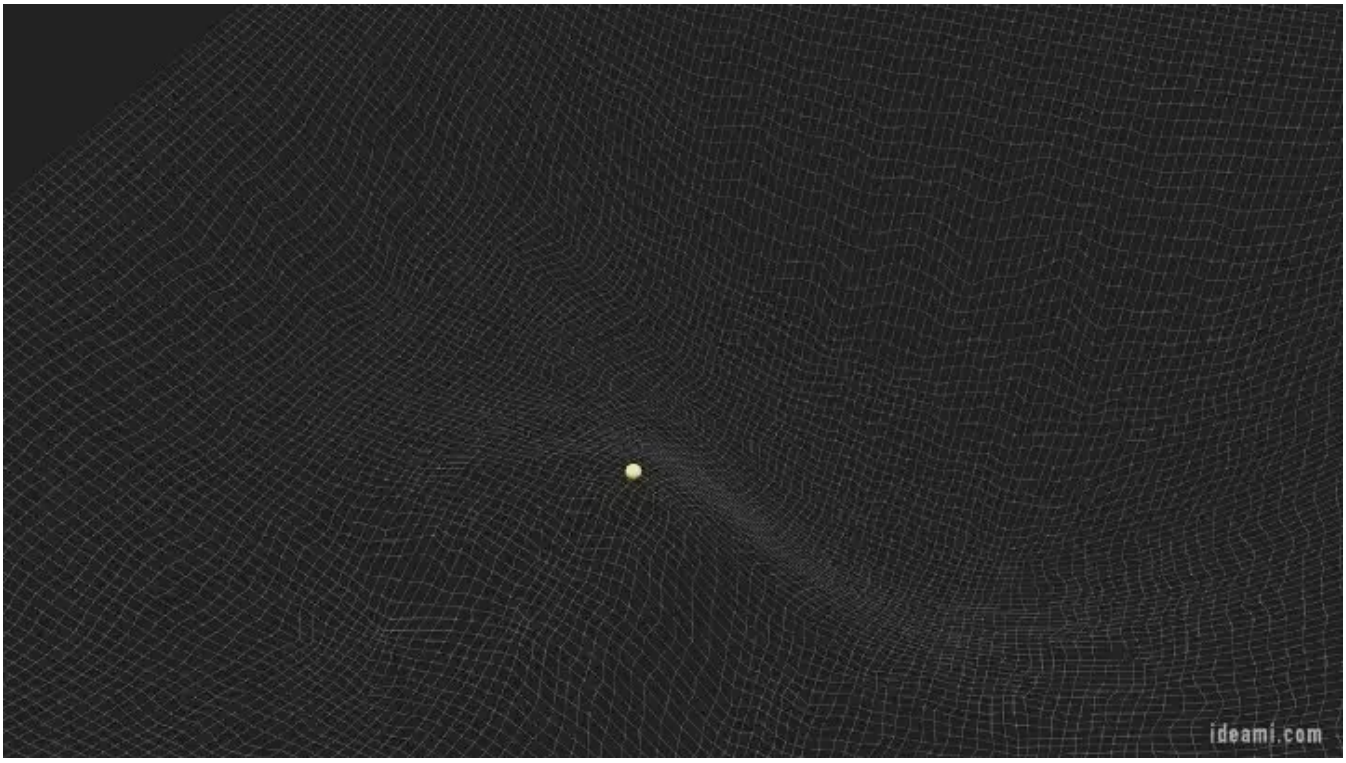
- We then feed the input data through the network to perform a forward pass and obtain **Yh**. With **Yh** we can now **calculate the loss:** how far are we from the ideal result **Y**? Our objective is to minimize the loss, to make it as small as possible.
- What you are seeing in the animation is **a landscape of the possible loss values of our network.**
- The landscape has **hills and valleys.** Hills are places where the loss is high. Valleys are what we call **minima,** places where the loss is low.



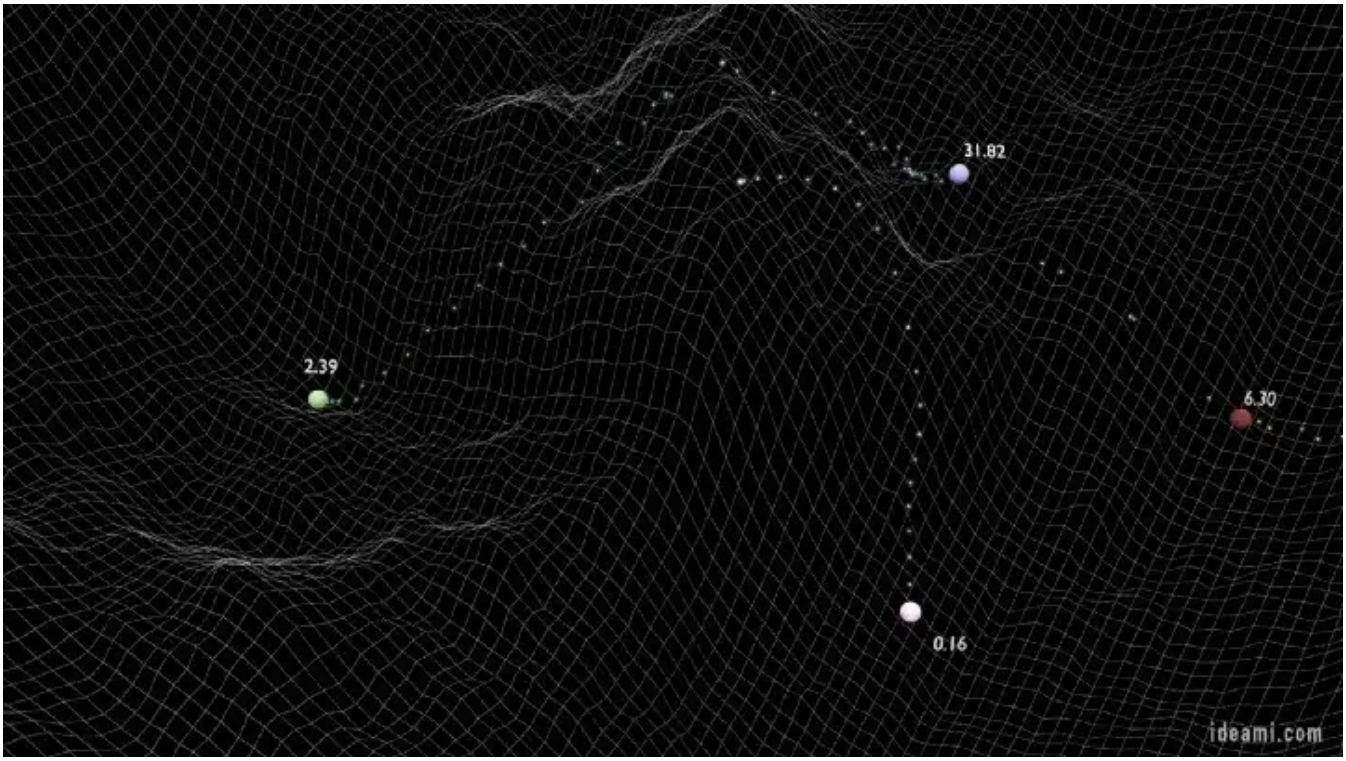
- A function can have multiple **local minima** and a **global minima.** The global minima is the very lowest part of the landscape, the lowest possible loss value. There can be other valleys that are local minima, places where the loss is low, yet not as low as it could potentially be.



- **When we initially set our weights and biases with random values** and perform our first forward pass to calculate our first loss value, it is as if we were **positioning the ball randomly in an initial part of the landscape** within that animation.
- We are **dropping the ball randomly somewhere in the landscape of the possible loss values.**
- Typically we will drop it in one of the hills of that landscape, because at the beginning the weights are random, the network is not very efficient, the loss will be high and we will be positioned at one of the high points, at one of the hills (high loss) of the landscape

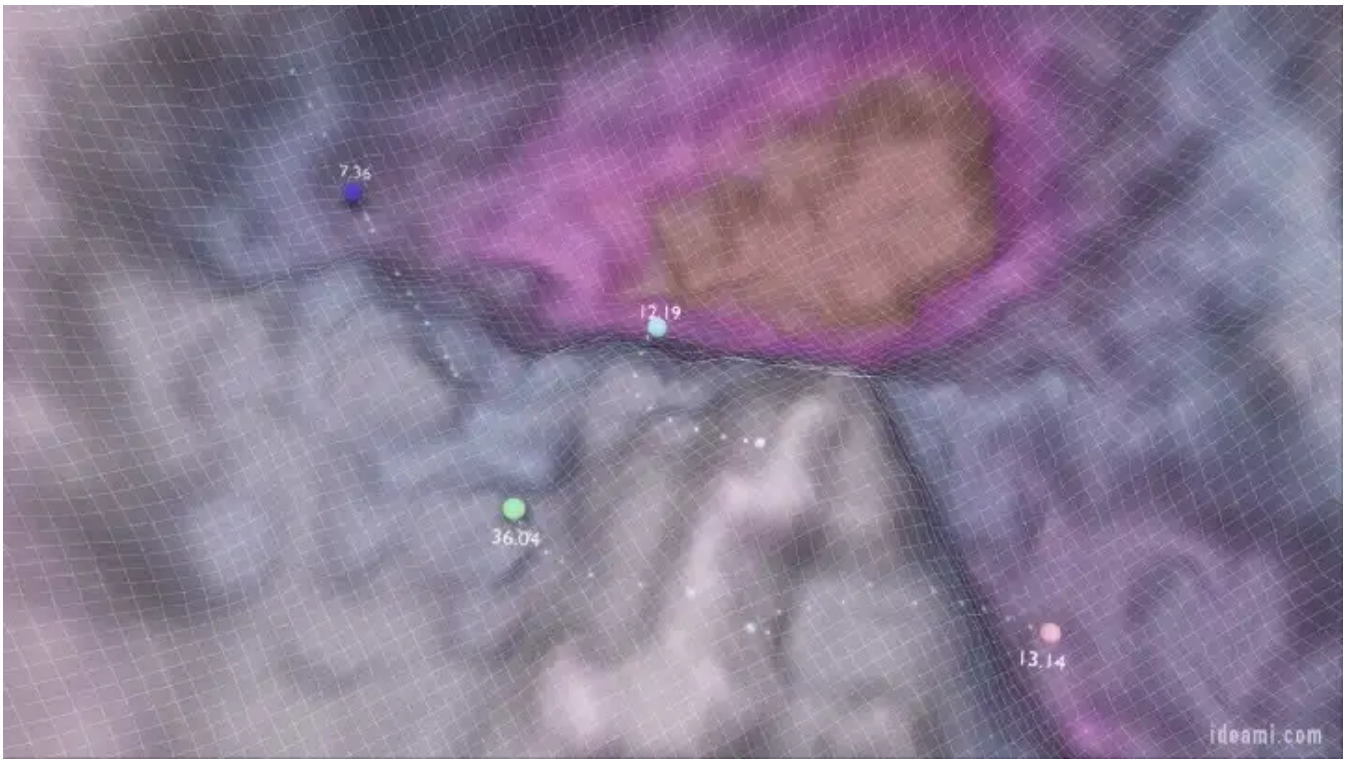


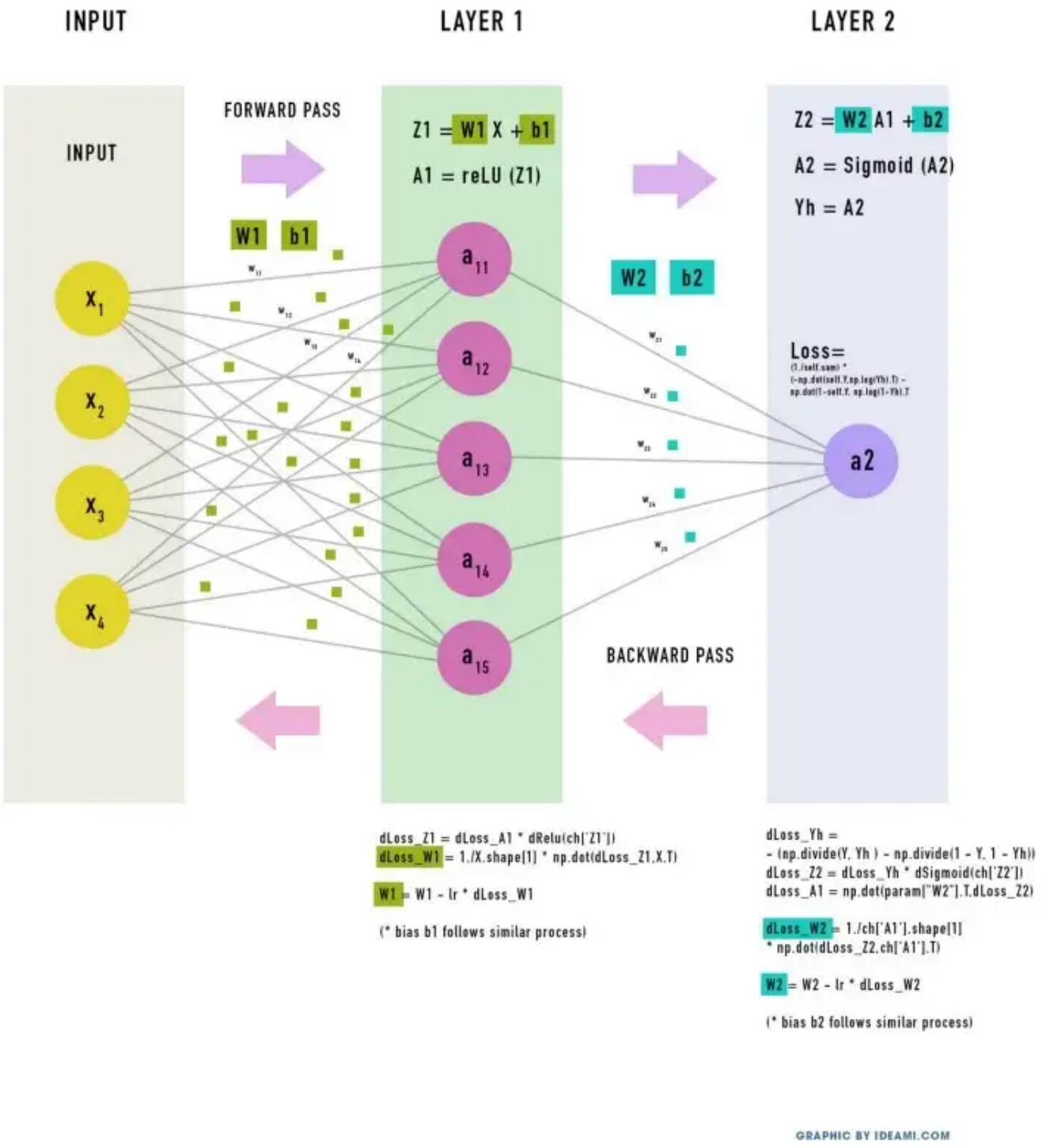
- Our objective is to gradually move from that initial point, high up, towards one of the valleys, hopefully to the global minima (the lowest valley), a part of the landscape where the loss is as small as possible.



- In order to do that, we perform the **gradient descent algorithm.** We already saw an iteration of it in the previous section. All we have to do now is to keep **repeating the same process until our loss becomes small enough.**

- We perform a forward pass, calculate the loss, and then perform a backward pass to update our weights and biases.
- We then repeat the same process for a number of iterations (set in advance), or until the loss becomes stable.





Let's take a look at the code:

```
nn = dlnet(x,y)
nn.gd(x, y, iter = 15000)

def gd(self,X, Y, iter = 3000):
        np.random.seed(1)

        self.nInit()
```

```
        for i in range(0, iter):
            Yh, loss=self.forward()
            self.backward()

            if i % 500 == 0:
                print ("Cost after iteration %i: %f" %(i, loss))
                self.loss.append(loss)

        return
```

that's it? Yes, that's it.

We first instantiate our neural network. And then run a number of iterations, performing forward and backward passes and updating our weights. Every x iterations we print the loss value.

After less than 100 lines of Python code, **we have a fully functional 2 layer neural network that performs back-propagation and gradient descent.**

This is a basic network that can now be optimized in many ways. Because as we will soon discuss, the performance of neural networks is strongly influenced by a number of key issues. Two very important ones are:

- **Feature engineering:** Understanding our input data and preparing it in a way that makes the job of the network easier.
- **Hyper-parameter optimization:** The hyper-parameters of a neural network are those variables that have a key impact on the training process, beyond the weights and biases. They include: **the learning rate, the number of layers, the number of units per layer** and others.

Regarding the optimization algorithm, in this article we are using the simplest and purest version of the gradient descent algorithm. Our purpose is to understand back-propagation and the basic optimization and training process.

There are many variations of gradient descent, and later I will name a few of them. Two very simple ones are:

- **Batch Gradient Descent:** Instead of running our forward and backward passes on the entire training set, we will run them in batches. Suppose we have 1000 rows of data. We divide our 1000 rows in 10 batches of 100 rows each. We will

then run the training loop (forward and backward pass) on the first batch, the first 100 rows, and proceed to update our weights. We then proceed with the rest of the batches. When we complete all our batches (10 in this case), that is called an **Epoch**. We then proceed with the next Epoch. When working with large data-sets, Batch Gradient Descent trains faster than the pure gradient descent algorithm because you are updating your weights more often (you don't have to wait till you process your entire data-set before updating the parameters of the network).

- **Stochastic Gradient Descent:** When your batches have a **size of 1**, that's called stochastic gradient descent. It's even faster because you update your weights after processing every single sample. However, it can produce irregular paths (noisy) through the loss landscape because the network is taking decisions based on very little data. There are different ways to deal with this and some of them are mentioned later in this article.

It is now time to test and try our network. Only by using it we will fully understand its potential and its limitations.

In the final part of this article, part 3, we will work with the Wisconsin cancer data-set, learn to prepare our data, run it through out network and analyze the results. We will also discuss some more advanced topics. Let's go to **Part 3**.

Links to the 3 parts of this article:
Part 1 | **Part 2** | **Part 3**

Machine Learning Deep Learning Artificial Intelligence Data Science

Towards Data Science