Neural Network from scratch in Python

Make your own machine learning library.

Photo by Mathew Schwartz on Unsplash

In this post we will go through the mathematics of machine learning and code from scratch, in Python, a small library to build neural networks with a variety of layers (Fully Connected, Convolutional, etc.). Eventually, we will be able to create networks in a modular fashion:

```
net = Network()
net.add(FCLayer(2, 3))
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FCLayer(3, 1))
net.add(ActivationLayer(tanh, tanh_prime))
```

3-layer neural network

I'm assuming you already have *some* knowledge about neural networks. The purpose here is not to explain why we make these models, but to show **how to make a proper implementation**.

Layer by Layer

We need to keep in mind the big picture here :

1. We feed **input** data into the neural network.
2. The data flows **from layer to layer** until we have the **output**.
3. Once we have the output, we can calculate the **error** which is a **scalar**.
4. Finally we can adjust a given parameter (weight or bias) by subtracting the **derivative** of the error with respect to the parameter itself.
5. We iterate through that process.

The most important step is the **4th**. We want to be able to have as many layers as we want, and of any type. But if we modify/add/remove one layer from the network, the output of the network is going to change, which is going to change the error, which is going to change the derivative of the error with respect to the parameters. We need to be able to compute the derivatives regardless of the network architecture, regardless of the activation functions, regardless of the loss we use.

In order to achieve that, we must implement **each layer separately**.

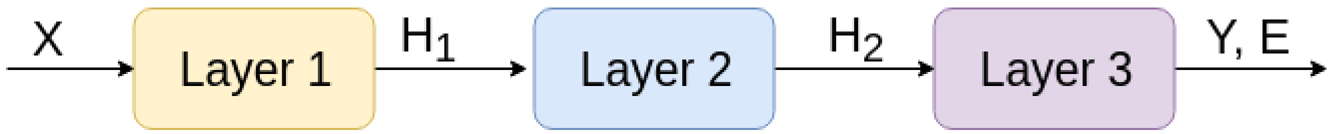
What every layer should implement

Every layer that we might create (fully connected, convolutional, maxpooling, dropout, etc.) have at least 2 things in common: **input** and **output** data.

$$X \rightarrow \boxed{\text{layer}} \rightarrow Y$$

Forward propagation

We can already emphasize one important point which is: **the output of one layer is the input of the next one**.



This is called **forward propagation.** Essentially, we give the input data to the first layer, then the output of every layer becomes the input of the next layer until we reach the end of the network. By comparing the result of the network (Y) with the desired output (let's say Y*), we can calculate en error **E**. The goal is to **minimize** that error by changing the parameters in the network. That is backward propagation (backpropagation).

Gradient Descent

This is a quick **reminder**, if you need to learn more about gradient descent there are tons of resources on the internet.

Basically, we want to change some parameter in the network (call it **w**) so that the total error **E decreases**. There is a clever way to do it (not randomly) which is the following :

$$w \leftarrow w - \alpha \frac{\partial E}{\partial w}$$

Where **α** is a parameter in the range [0,1] that we set and that is called the **learning rate**. Anyway, the important thing here is **∂E/∂w** (the derivative of E with respect to w). **We need to be able to find the value of that expression for any parameter of the network regardless of its architecture.**

Backward propagation

Suppose that we give a layer the **derivative of the error with respect to its output** (∂E/∂Y), then it must be able to provide the **derivative of the error with respect to its input** (∂E/∂X).

$$\frac{\partial E}{\partial X} \leftarrow \boxed{\text{layer}} \leftarrow \frac{\partial E}{\partial Y}$$

Remember that `E` is a **scalar** (a number) and `X` and `Y` are **matrices**.

$$\frac{\partial E}{\partial X} = \begin{bmatrix} \frac{\partial E}{\partial x_1} & \frac{\partial E}{\partial x_2} & \dots & \frac{\partial E}{\partial x_i} \end{bmatrix}$$
$$\frac{\partial E}{\partial Y} = \begin{bmatrix} \frac{\partial E}{\partial y_1} & \frac{\partial E}{\partial y_2} & \dots & \frac{\partial E}{\partial y_j} \end{bmatrix}$$

Let's forget about ∂E/∂X for now. The trick here, is that if we have access to ∂E/∂Y we can very easily calculate ∂E/∂W (if the layer has any trainable parameters) **without knowing anything about the network architecture !** We simply use the chain rule :

$$\frac{\partial E}{\partial w} = \sum_j \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial w}$$

The unknown is **∂y_j/∂w** which totally depends on how the layer is computing its output. So if every layer have access to ∂E/∂Y, where Y is its own output, then we can update our parameters !

But why ∂E/∂X ?

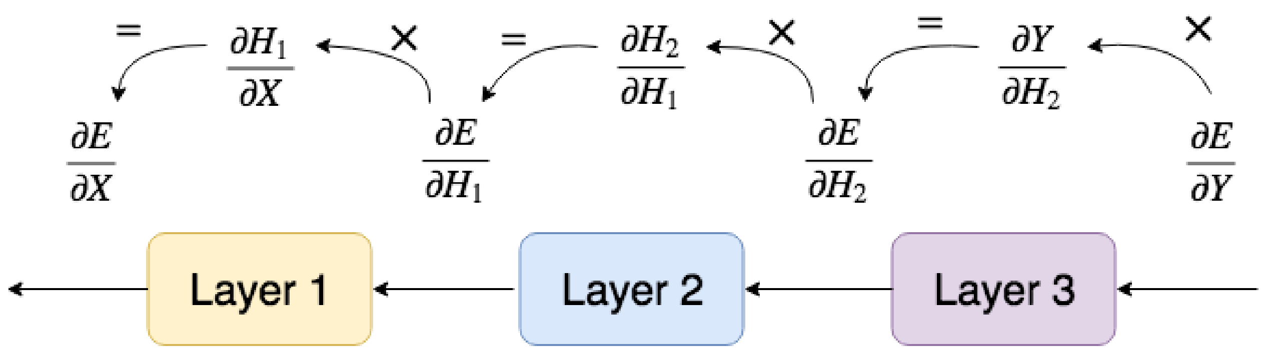
Don't forget, the output of one layer is the input of the next layer. Which means ∂E/∂X for one layer is ∂E/∂Y for the previous layer ! That's it ! It's just a clever way to propagate the error ! Again, we can use the chain rule :

$$\frac{\partial E}{\partial x_i} = \sum_j \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

This is *very* important, it's the *key* to understand backpropagation ! After that, we'll be able to code a Deep Convolutional Neural Network from scratch in no time !

Diagram to understand backpropagation

This is what I described earlier. Layer 3 is going to update its parameters using ∂E/∂Y, and is then going to pass ∂E/∂H2 to the previous layer, which is its own "∂E/∂Y". Layer 2 is then going to do the same, and so on and so forth.



This may seem abstract here, but it will get very clear when we will apply this to a specific type of layer. Speaking of *abstract*, now is a good time to write our first python class.

Abstract Base Class : Layer

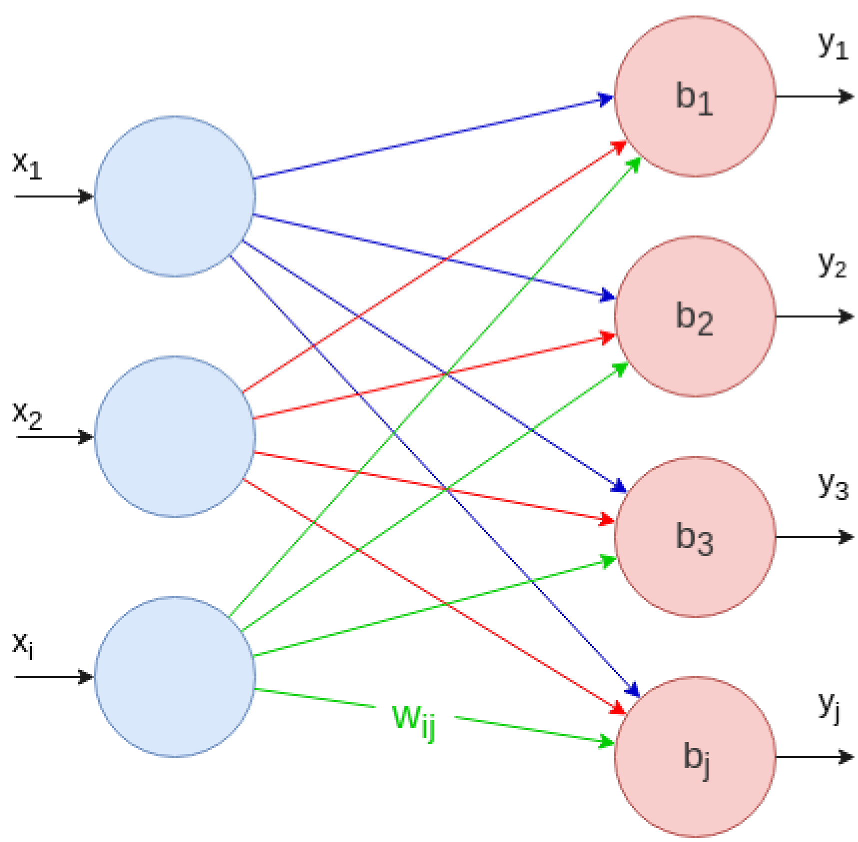
The abstract class *Layer*, which all other layers will inherit from, handles simple properties which are an **input**, an **output**, and both a **forward** and **backward** methods.

3-layer neural network

As you can see there is an extra parameter in `backward_propagation` that I didn't mention, it is the `learning_rate`. This parameter should be something like an update policy, or an optimizer as they call it in Keras, but for the sake of simplicity we're simply going to pass a learning rate and update our parameters using gradient descent.

Fully Connected Layer

Now let's define and implement the first type of layer: fully connected layer or FC layer. FC layers are the most basic layers as every input neurons are connected to every output neurons.



Forward Propagation

The value of each output neuron can be calculated as the following :

$$y_j = b_j + \sum_i x_i w_{ij}$$

With matrices, we can compute this formula for every output neuron in one shot using a **dot product** :

$$X = \begin{bmatrix} x_1 & \dots & x_i \end{bmatrix} \quad W = \begin{bmatrix} w_{11} \dots w_{1j} \\ \vdots \quad \ddots \quad \vdots \\ w_{i1} \dots w_{ij} \end{bmatrix} \quad B = \begin{bmatrix} b_1 & \dots & b_j \end{bmatrix}$$

$$Y = XW + B$$

We're done with the forward pass. Now let's do the backward pass of the FC layer.

Note that I'm not using any activation function yet, that's because we will implement it in a separate layer!

Backward Propagation

As we said, suppose we have a matrix containing the derivative of the error with respect to **that layer's output** (∂E/∂Y). We need :

1. The derivative of the error with respect to the parameters (∂E/∂W, ∂E/∂B)
2. The derivative of the error with respect to the input (∂E/∂X)

Let's calculate ∂E/∂W. This matrix should be the same size as W itself : `ixj` where `i` is the number of input neurons and `j` the number of output neurons. We need **one gradient for every weight** :

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial w_{11}} & \cdots & \frac{\partial E}{\partial w_{1j}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial w_{i1}} & \cdots & \frac{\partial E}{\partial w_{ij}} \end{bmatrix}$$

Using the chain rule stated earlier, we can write :

$$\begin{aligned} \frac{\partial E}{\partial w_{ij}} &= \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial w_{ij}} + \ldots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial w_{ij}} \\ &= \frac{\partial E}{\partial y_j} x_i \end{aligned}$$

Therefore,

$$
\begin{aligned}
\frac{\partial E}{\partial W} &= \begin{bmatrix} \frac{\partial E}{\partial y_1} x_1 & \dots & \frac{\partial E}{\partial y_j} x_1 \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial y_1} x_i & \dots & \frac{\partial E}{\partial y_j} x_i \end{bmatrix} \\
&= \begin{bmatrix} x_1 \\ \vdots \\ x_i \end{bmatrix} \begin{bmatrix} \frac{\partial E}{\partial y_1} & \dots & \frac{\partial E}{\partial y_j} \end{bmatrix} \\
&= X^t \frac{\partial E}{\partial Y}
\end{aligned}
$$

That’s it we have the first formula to update the weights! Now let's calculate ∂E/∂B.

$$
\frac{\partial E}{\partial B} = \begin{bmatrix} \frac{\partial E}{\partial b_1} & \frac{\partial E}{\partial b_2} & \dots & \frac{\partial E}{\partial b_j} \end{bmatrix}
$$

Again ∂E/∂B needs to be of the same size as B itself, one gradient per bias. We can use the chain rule again :

$$
\begin{aligned}
\frac{\partial E}{\partial b_j} &= \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial b_j} + \ldots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial b_j} \\
&= \frac{\partial E}{\partial y_j}
\end{aligned}
$$

And conclude that,

$$
\begin{aligned}
\frac{\partial E}{\partial B} &= \begin{bmatrix} \frac{\partial E}{\partial y_1} & \frac{\partial E}{\partial y_2} & \ldots & \frac{\partial E}{\partial y_j} \end{bmatrix} \\
&= \frac{\partial E}{\partial Y}
\end{aligned}
$$

Now that we have **∂E/∂W** and **∂E/∂B**, we are left with **∂E/∂X** which is **very important** as it will "act" as ∂E/∂Y for the layer before that one.

$$
\frac{\partial E}{\partial X} = \begin{bmatrix} \frac{\partial E}{\partial x_1} & \frac{\partial E}{\partial x_2} & \ldots & \frac{\partial E}{\partial x_i} \end{bmatrix}
$$

Again, using the chain rule,

$$
\begin{aligned}
\frac{\partial E}{\partial x_i} &= \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial x_i} + \ldots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial x_i} \\
&= \frac{\partial E}{\partial y_1} w_{i1} + \ldots + \frac{\partial E}{\partial y_j} w_{ij}
\end{aligned}
$$

Finally, we can write the whole matrix :

$$
\begin{aligned}
\frac{\partial E}{\partial X} &= \begin{bmatrix} (\frac{\partial E}{\partial y_1} w_{11} + \ldots + \frac{\partial E}{\partial y_j} w_{1j}) & \ldots & (\frac{\partial E}{\partial y_1} w_{i1} + \ldots + \frac{\partial E}{\partial y_j} w_{ij}) \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial E}{\partial y_1} & \ldots & \frac{\partial E}{\partial y_j} \end{bmatrix} \begin{bmatrix} w_{11} & \ldots & w_{i1} \\ \vdots & \ddots & \vdots \\ w_{1j} & \ldots & w_{ij} \end{bmatrix} \\
&= \frac{\partial E}{\partial Y} W^t
\end{aligned}
$$

That's it! We have the three formulas we needed for the FC layer!

$$
\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y} W^t
$$
$$
\frac{\partial E}{\partial W} = X^t \frac{\partial E}{\partial Y}
$$
$$
\frac{\partial E}{\partial B} = \frac{\partial E}{\partial Y}
$$

Coding the Fully Connected Layer

We can now write some python code to bring this math to life!

Activation Layer

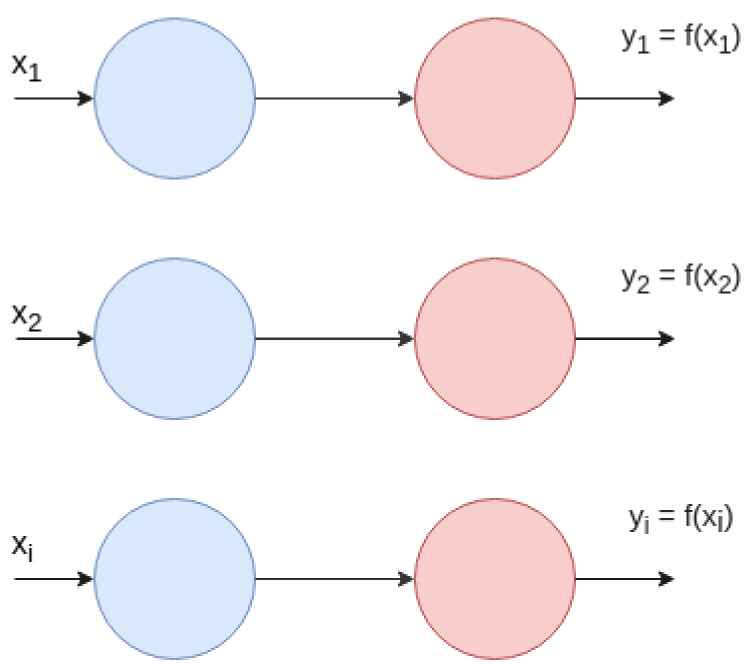
All the calculation we did until now were completely linear. It's hopeless to learn anything with that kind of model. We need to add **non-linearity** to the model by applying non-linear functions to the output of some layers.

Now we need to redo the whole process for this new type of layer!

https://gph.is/21pKLjE

No worries, it's going to be way faster as there are no *learnable* parameters. We just need to calculate **∂E/∂X**.

We will call `f` and `f'` the activation function and its derivative respectively.



Forward Propagation

As you will see, it is quite straightforward. For a given input `X`, the output is simply the activation function applied to every element of `X`. Which means **input** and **output** have the **same dimensions**.

$$
\begin{aligned}
Y &= \begin{bmatrix} f(x_1) & \dots & f(x_i) \end{bmatrix} \\
&= f(X)
\end{aligned}
$$

Backward Propagation

Given **∂E/∂Y**, we want to calculate **∂E/∂X**.

$$
\begin{aligned}
\frac{\partial E}{\partial X} &= \begin{bmatrix} \frac{\partial E}{\partial x_1} & \dots & \frac{\partial E}{\partial x_i} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial x_i} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial E}{\partial y_1} f'(x_1) & \dots & \frac{\partial E}{\partial y_i} f'(x_i) \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial E}{\partial y_1} & \dots & \frac{\partial E}{\partial y_i} \end{bmatrix} \odot \begin{bmatrix} f'(x_1) & \dots & f'(x_i) \end{bmatrix} \\
&= \frac{\partial E}{\partial Y} \odot f'(X)
\end{aligned}
$$

Be careful, here we are using an **element-wise** multiplication between the two matrices (whereas in the formulas above, it was a dot product).

Coding the Activation Layer

The code for the activation layer is as straightforward.

You can also write some activation functions and their derivatives in a separate file. These will be used later to create an `ActivationLayer`.

Loss Function

Until now, for a given layer, we supposed that **∂E/∂Y** was given (by the next layer). But what happens to the last layer? How does it get **∂E/∂Y**? We simply give it manually, and it depends on how we define the error.

The error of the network, which measures how good or bad the network did for a given input data, is defined by **you**. There are many ways to define the error, and one of the most known is called **MSE — Mean Squared Error**.

$$E = \frac{1}{n} \sum_{i}^{n} (y_i^* - y_i)^2$$

Mean Squared Error

Where `y*` and `y` denotes **desired output** and **actual output** respectively. You can think of the loss as a last layer which takes all the output neurons and squashes them into one single neuron. What we need now, as for every other layer, is to define **∂E/∂Y**. Except now, we finally reached `E` !

$$\begin{aligned} \frac{\partial E}{\partial Y} &= \begin{bmatrix} \frac{\partial E}{\partial y_1} & \dots & \frac{\partial E}{\partial y_i} \end{bmatrix} \\ &= \frac{2}{n} \begin{bmatrix} y_1 - y_1^* & \dots & y_i - y_i^* \end{bmatrix} \\ &= \frac{2}{n}(Y - Y^*) \end{aligned}$$

These are simply two python functions that you can put in a separate file. They will be used when creating the network.

Network Class

Almost done ! We are going to make a `Network` class to create neural networks very easily akin the first picture !

I commented almost every part of the code, it shouldn't be too complicated to understand if you grasped the previous steps. Nevertheless, leave a comment if you have any question, I will gladly answer !

Building Neural Networks

Finally ! We can use our class to create a neural network with as many layers as we want ! We are going to build two neural networks : a simple **XOR** and a **MNIST** solver.

Solve XOR

Starting with XOR is always important as it's a simple way to tell if the network is learning anything at all.

I don't think I need to emphasize many things. Just be careful with the training data, you should always have the **sample** dimension **first**. For example here, the input shape is **(4,1,2)**.

Result

```
$ python xor.py
epoch 1/1000 error=0.322980
epoch 2/1000 error=0.311174
epoch 3/1000 error=0.307195
...
epoch 998/1000 error=0.000243
epoch 999/1000 error=0.000242
epoch 1000/1000 error=0.000242

[
    array([[ 0.00077435]]),
    array([[ 0.97760742]]),
    array([[ 0.97847793]]),
    array([[-0.00131305]])
]
```

Clearly this is working, great ! We can now solve something more interesting, let's solve MNIST !

Solve MNIST

We didn't implemented the Convolutional Layer but this is not a problem. All we need to do is to reshape our data so that it can fit into a Fully Connected Layer.

MNIST Dataset consists of images of digits from 0 to 9, of shape 28x28x1. The goal is to predict what digit is drawn on a picture.

Result

```
$ python example_mnist_fc.py
epoch 1/30   error=0.238658
epoch 2/30   error=0.093187
epoch 3/30   error=0.073039
...
epoch 28/30   error=0.011636
epoch 29/30   error=0.011306
epoch 30/30   error=0.010901

predicted values :
[
    array([[ 0.119,  0.084 , -0.081,  0.084, -0.068, 0.011,  0.057,
0.976, -0.042, -0.0462]]),
    array([[ 0.071,  0.211,  0.501 ,  0.058, -0.020, 0.175,  0.057 ,
0.037,  0.020,  0.107]]),
    array([[ 1.197e-01,  8.794e-01, -4.410e-04, 4.407e-02, -4.213e-02,
5.300e-02, 5.581e-02,  8.255e-02, -1.182e-01, 9.888e-02]])
]
true values :
[[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

This is working perfectly ! Amazing :)