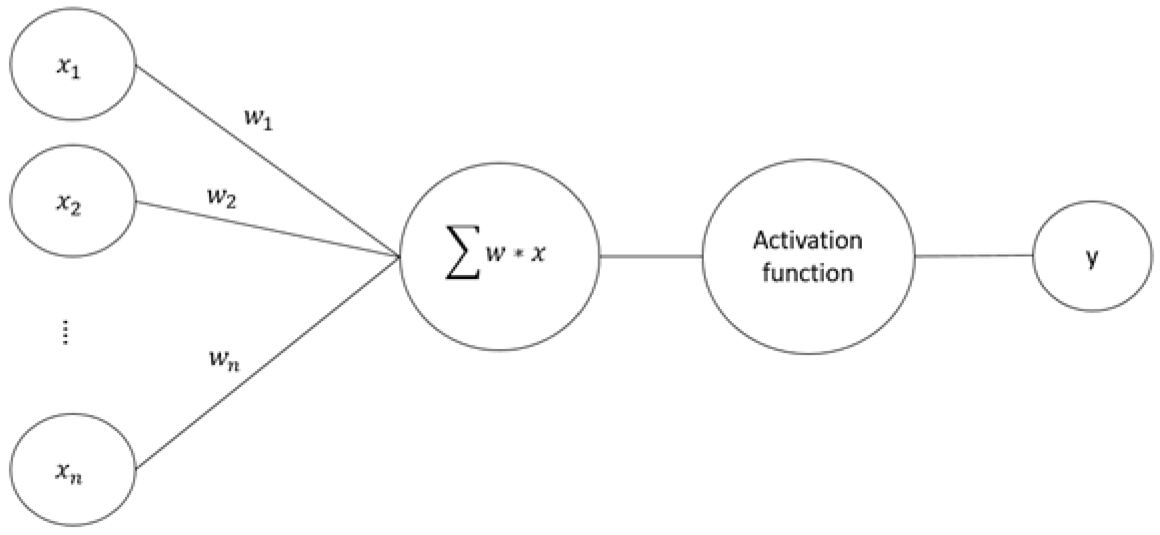
Valentina Alto Follow

Jul 6, 2019 · 10 min read ★ · Listen

Neural Networks: parameters, hyperparameters and optimization strategies

Neural Networks (NNs) are the typical algorithms used in Deep Learning analysis. NNs can take different shapes and structures, nevertheless, the core skeleton is the following:



So we have our inputs (x), we take the weighted sum of them (with weights equal to w), Needless to say, this is a tremendously poor definition, but if you keep it in mind while reading this article, you will better understand its core topic.

Indeed, what I want to focus on is how to approach some characteristic elements of NNs, whose initialization, optimization and tuning can make your algorithm much more powerful. Before starting, let's see which elements I'm talking about:

· **Parameters**: these are the coefficients of the model, and they are chosen by the model itself. It means that the algorithm, while learning, optimizes these coefficients (according to a given optimization strategy) and returns an array of parameters which minimize the error. To give an example, in a linear regression task, you have your model that will look like y=b + ax, where b and a will be your parameter. The only thing you have to do with those parameters is initializing them (we will see later on what it means).

· **Hyperparameters**: these are elements that, differently from the previous ones, you need to set. Furthermore, the model will not update them according to the optimization strategy: your manual intervention will always be needed.

· **Strategies**: these are some tips and approaches you should have towards your model. Namely, before managing your data, you might want to normalize them, especially if you have values on different scales and this might affect your algorithm's performance.

So, let's examine all of them.

Parameters

As anticipated, the only thing you have to do with respect to parameters is initializing them (note that parameters initialization is a strategy). So, which is the best way to initialize them? For sure, what you should NOT do is setting them equal to zero: indeed, by doing so you are risking to penalize the whole algorithm. To give an example of the variety of problems you might face, there is that of remaining stuck to weights equal to zero even after several re-weighting procedures.

· If you are using Sigmoid or Tanh activation function, you might use a Xavier initialization, with uniform or normal distribution:

$$Uniform\ distribution\ \left(-\sqrt{\frac{6}{n_i + n_0}}, \sqrt{\frac{6}{n_i + n_0}}\right)$$

$$Normal\ Distribution\ with\ \mu = 0\ and\ \sigma = \sqrt{\frac{2}{n_i + n_0}}$$

· If you are using a ReLU instead, you could use the He initialization, with a normal distribution:

$$Normal\ Distribution\ with\ \mu = 0\ and\ \sigma = \sqrt{\frac{2}{n_i}}$$

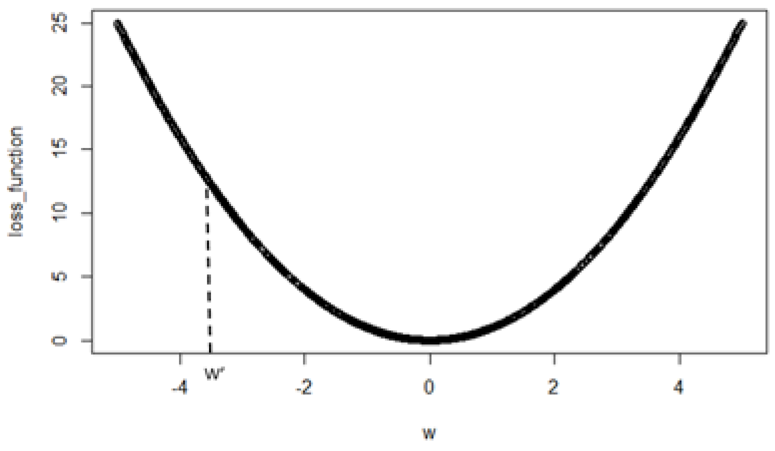
Where *ni* and *n0* are, respectively, the number of inputs and the number of outputs.

Hyperparameters

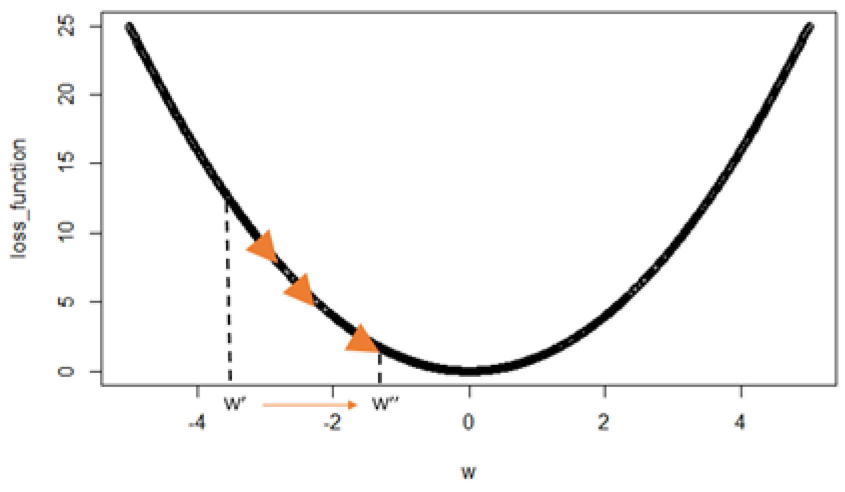
This is far more interesting. Hyperparameters requires your attention and knowledge much more than parameters. So, to have an idea on how to handle them, let's examine some of them:

· **Number of hidden layers**: this is probably the most questionable point. The idea is that you want to keep your NN as simple as possible (you want it to be fast and well those latter need to be built, before learning. So, to provide the intuition of this, I suggest you to run some experiments on this Tensorflow platform: you will see how, after a number of layers/neurons, the accuracy does not improve anymore, hence it would be inefficient to keep the algorithm so heavy.

· **Learning rate**: this hyperparameter refers to the step of backpropagation, when parameters are updated according to an optimization function. Basically, it represents how important is the change it the weight after a re-calibration. But what does it mean 're-calibration'? Well, if you think about a generic loss function with only one weight, the graphic representation will be something like that:



You want to minimize the loss, so ideally you want your current w to slide towards the minimum. The procedure should be the following:



The correspondent function is:

$$w_i' = w_i - \frac{\delta L}{\delta w_i}$$

Where the first term is your current weight, the second term is the gradient of your function (in this one-dimension case, it will be the first derivate of your loss function with respect, obviously, to your only weight). Remember that the first derivate has a negative value if the steep of the tangent segment is negative, that's why we put a minus in the middle of the two terms (intuition: if the steep is negative, the weight should move towards right, as in the example). This optimization procedure is called Gradient Descent.

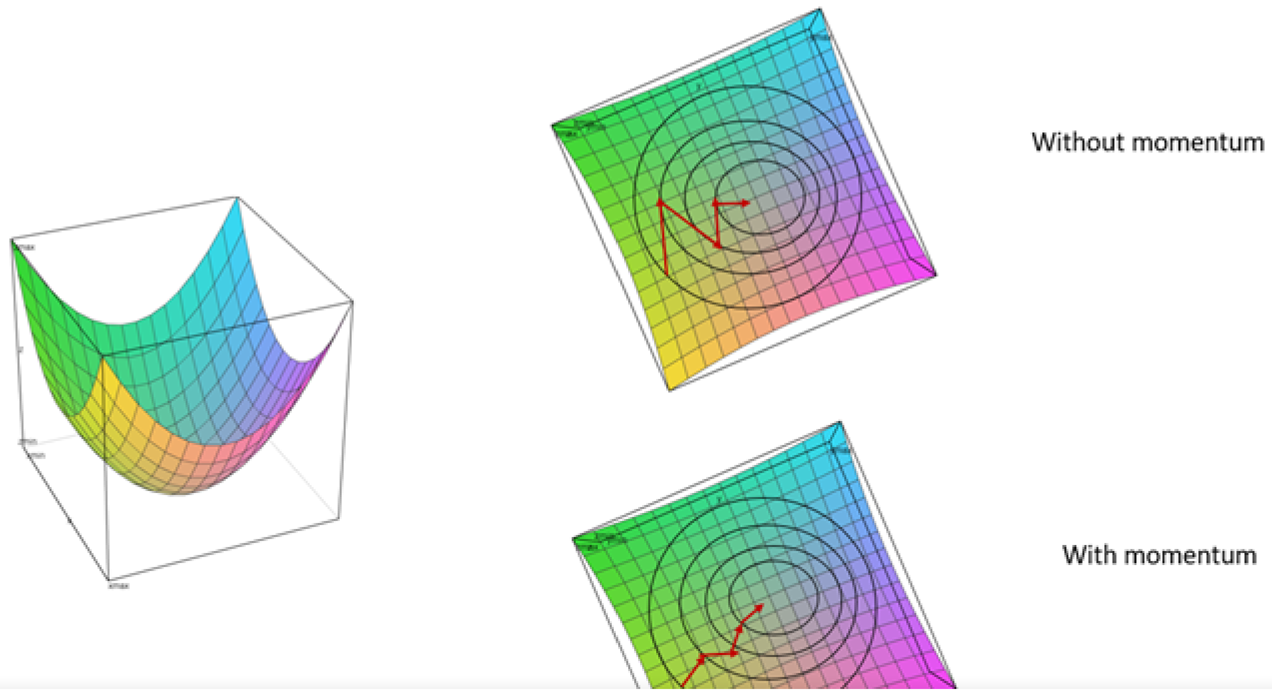
Now let's add a new term to the formula:

$$w_i' = w_i - \gamma \frac{\delta L}{\delta w_i}$$

'Vanishing Gradient'. On the other side, if gamma is very big, the risk is missing the minimum and incur in the scenario of 'Exploding Gradient'.

A good strategy might be starting with a value around 0.1, and then exponentially reduce it: at some point, the value of the loss function starts decreasing in the first few iterations and that's the signal the weight took the right direction.

· **Momentum**: it is a technique used during the backpropagation phase. As said regarding the learning rate, parameters are updated so that they can converge towards the minimum of the loss function. This process might be too long and affecting the efficiency of the algorithm. Hence, one possible solution is taking track of the previous directions (that are the gradients of the loss function with respect to weights) and keeping them as embedded information: this is what momentum is thought for. It basically increases the speed of convergence not in terms of learning rate (how much a weight is updated each time) but in terms of embedded memory of past re-calibration (the algorithm knows the previous direction of that weight was, let's say, right, and it will directly proceed towards this direction during the next propagation). We can visualize it if we consider the projection of a two-weights loss function (specifically, a paraboloid):

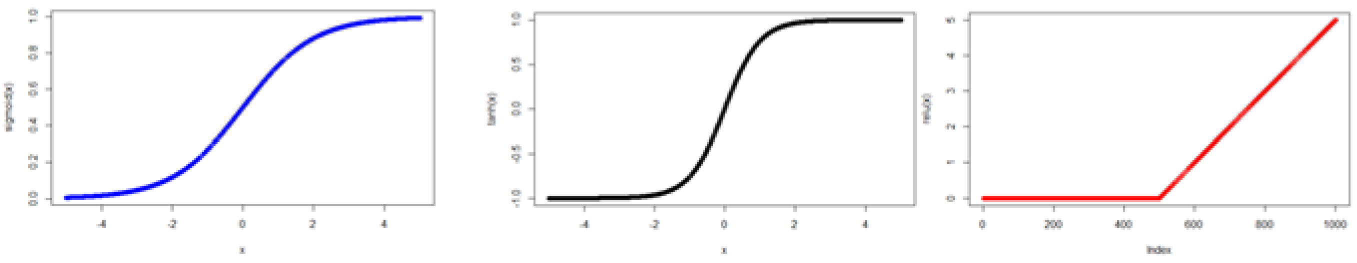


You can find the source for making these 3D graphs here.

As you can see, if we add momentum hyperparameter the descending phase is faster, since the model keeps traces of the past gradient directions.

If you decide for high values of momentum, it means it will massively take into account the past directions: it might result in an incredibly fast learning algorithm, but the risk of missing some correct 'deviations' is high. The suggestion is always starting with low values and then increasing them little by little.

· **Activation function**: it is the function through which we pass our weighed sum, in order to have a significant output, namely as a vector of probability or a 0–1 output. The major activation functions are Sigmoid (for multiclass classification, a variant of this function is used, called SoftMax function: it returns as output a vector of probability whose sum is equal to one), Tanh and RELU.



$$S(x) = \frac{1}{1 + e^{-x}} \qquad T(x) = \frac{e^{2x}+1}{e^{2x}-1} \qquad R(x) = \begin{cases} 0 & if\ x < 0 \\ x & if\ x \geq 0 \end{cases}$$

Note that activation function can be located at any point in the NN, as many times as you want. However, you always have to think about efficiency and velocity. Namely, the ReLU function is very quick in terms of training, while the Sigmoid is more complex and it takes more time. Hence, a good practice might be using ReLU for hidden layers and then, in the last layer, inserting your Sigmoid smaller samples of your data, called batches: by doing so, every time the algorithm train itself, it will train on a sample of the same size of the batch. The typical size is 32 or higher, however you need to keep in mind that, if the size is too big, the risk is an over generalized model which won't fit new data well.

· **Epochs**: it represents how many time you want your algorithm to train on your whole dataset (note that epochs are different from iterations: those latter are the number of batches needed to complete one epoch). Again, the number of epochs depend on the kind of data and task you are facing. An idea could be imposing a condition such that epochs stop when the error is close to zero. Or, more easily, you can start with a relatively low number of epochs and then increase it progressively, tracking some evaluation metrics (like accuracy).

· **Dropout**: this technique consists of removing some nodes so that the NN is not too heavy. This can be implemented during the training phase. The idea is that we do not want our NN to be overwhelmed by information, especially if we consider that some nodes might be redundant and useless. So, while building our algorithm, we can decide to keep, for each training stage, each node with probability p (called 'keep probability') or drop it with probability 1-p (called 'drop probability').

Strategies

Strategies are approaches and best practices we might want to have towards our algorithm to make it more performing. Among these there are the following:

· **Parameter initialization**: we have been talking about that in the first paragraph.

· **Data normalization**: while inspecting your data, you might notice that some features are represented on different scales. This might affect the performance of your NN, since the convergence is slower. Normalizing data means converting all of them to the same scale, within the range [0–1]. You can also decide to Standardize your data, that means making them normally distributed with mean equal to 0 and standard deviation equal to 1. While data normalization happens before training your NN, another way you can

· **Optimization algorithm**: in the previous paragraph, I mentioned the gradient descent as the optimization algorithm. However, we have many variants of this latter: Stochastic Gradient Descent (it minimizes the loss according to the gradient descent optimization, and for each iteration it randomly selects a training sample — that's why it's called stochastic), the RMSProp (that differs from the previous since each parameter has an adapted learning rate) and the Adam Optimizer (it is a RMSProp + momentum). Of course, this is not the full list, yet it is sufficient to understand that Adam optimizer is often the best choice, since it allows you to set different hyperparameters and customize your NN.

· **Regularization**: this strategy is pivotal if you want to keep your model simple and avoid overfitting. The idea is that regularization adds a penalty to the model if weights are great/too many. Indeed, it adds to our loss function a new term which tends to increase (hence, the loss increases too) if the re-calibration procedure increases weights. There are two kinds of regularization: the Lasso regularization (L1) and Bridge regularization (L2):

$$Loss\ Function = L(\mathbf{w}) + \lambda \sum w_i^2$$

L2 Regularization

$$Loss\ Function = L(\mathbf{w}) + \lambda \sum |w_i|$$

L1 Regularization

The L1 regularization tends to shrink weights to zero, with the risk of getting rid of some inputs (since they will be multiplied with a null value), whereas the L2 might shrink weights to very low values, but not to zero (hence inputs are preserved).

It is interesting to note that this concept is strongly related to the Information Criteria in time series analysis. Indeed, while optimizing the Maximum Likelihood function of our Autoregressive model, we might incur in the same problem of overfitting, since this procedure tends to increase the number of parameters: that's why it is a good practice to add a penalty if this latter increases.

Conclusion