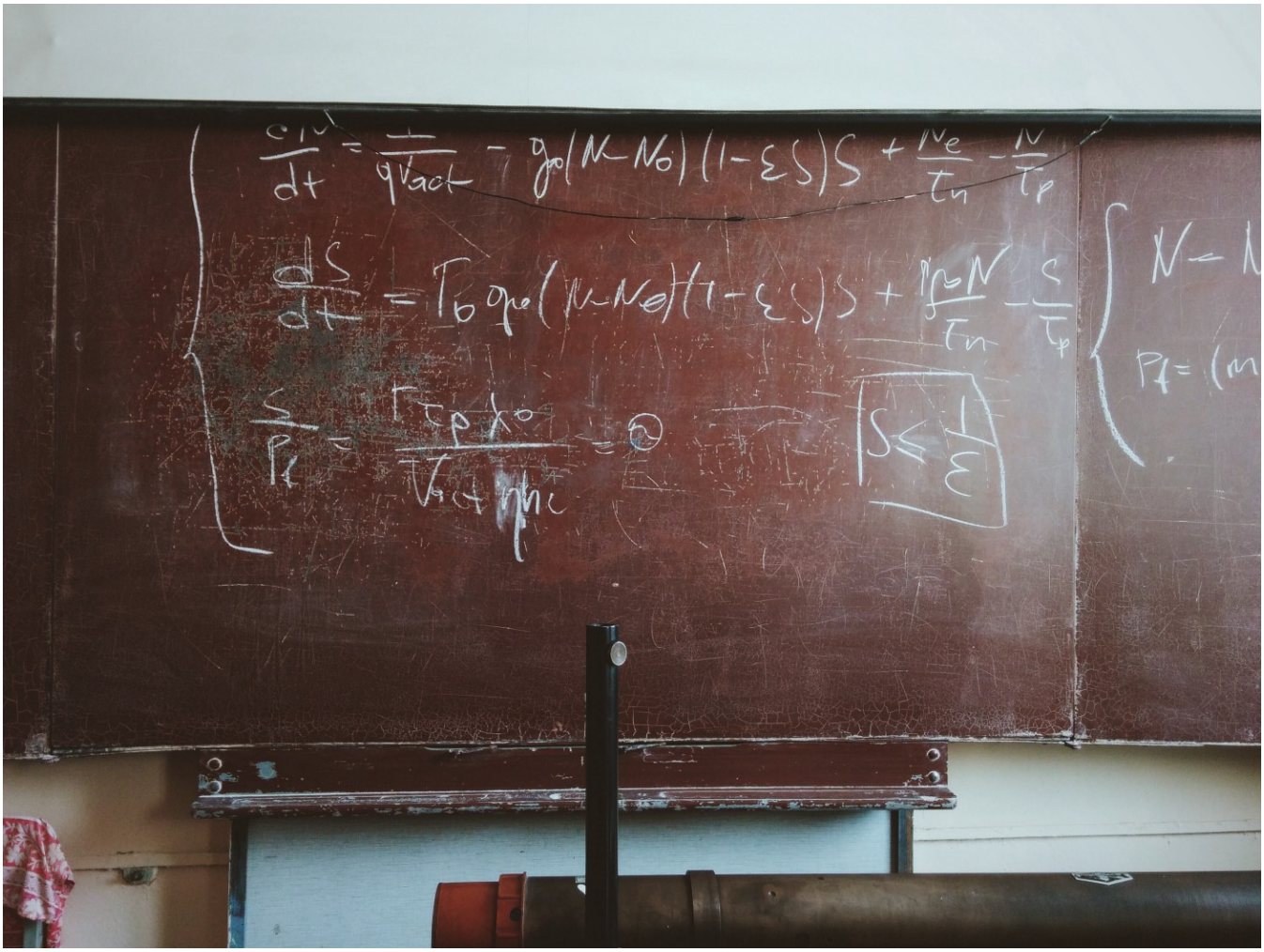


Photo by Roman Mager on Unsplash

Deep Learning's mathematics

Discover the mathematics and the equations behind the success of deep learning

Ismail Mebsout

Jan 31 · 14 min read ★

Deep learning is a subfield of Machine Learning Science which is based on artificial neural networks. It has several derivatives such as Multi-Layer Perceptron-MLP-, Convolutional Neural Networks -CNN- and Recurrent Neural Networks -RNN- which can be applied to many fields including Computer Vision, Natural Language Processing, Machine Translation…

Deep learning is taking off for three main reasons:

- **Instinctive features engineering**: while most of machine learning algorithms require human expertise for the feature engineering and extraction, deep learning handles automatically the choice of variables and their weights
- **Huge Datasets**: the continuous collection of data has led to large databases which allow deeper neural networks
- **Hardware evolution**: the new GPUs, for Graphical Process Units, allow faster algebraic calculation which is the core base of DL

In this blog, we will focus mainly on the Multi-Layer Perceptron -MLP- where we will detail the mathematical background behind the success of deep learning and explore the optimization algorithms used to improve its performances.

The summary is as follows:

1. Definition
2. Learning algorithm
3. Parameter Initialization
4. Forward — Backpropagation
5. Activation functions
6. Optimization algorithm

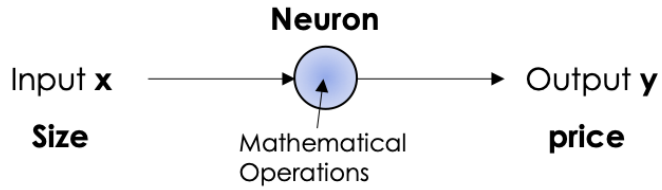
. . .

1- Definition

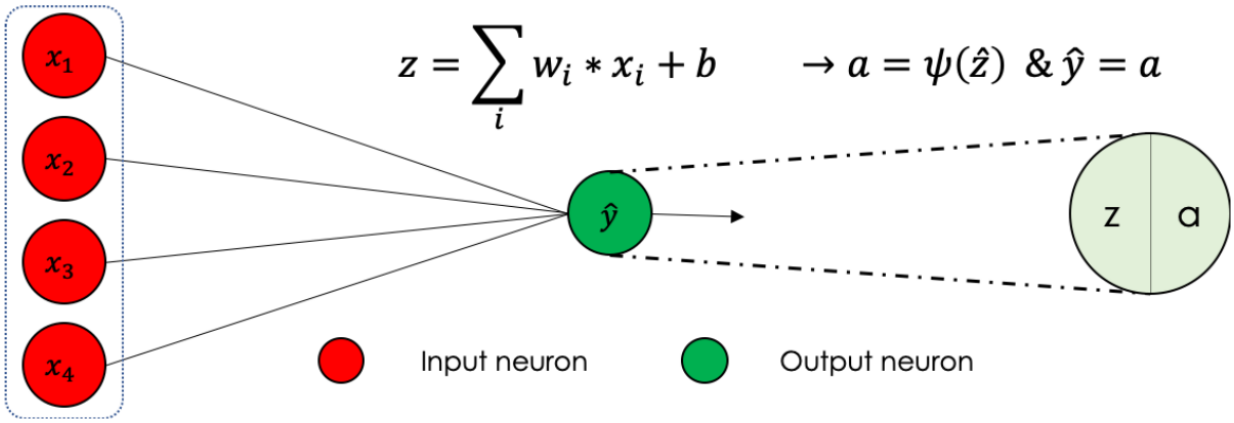
A neuron

> *It is a bloc of mathematical operations linking entities*

Let's consider the problem where we estimate the price of a house based on its size, it can be schematized as follows:



When including more description about the house by adding more variables, the graph becomes as follow:



Each neuron is divided into two main blocks:

- Computation of `z` using the inputs x_i:

$$z = \sum_i w_i \star x_i + b$$

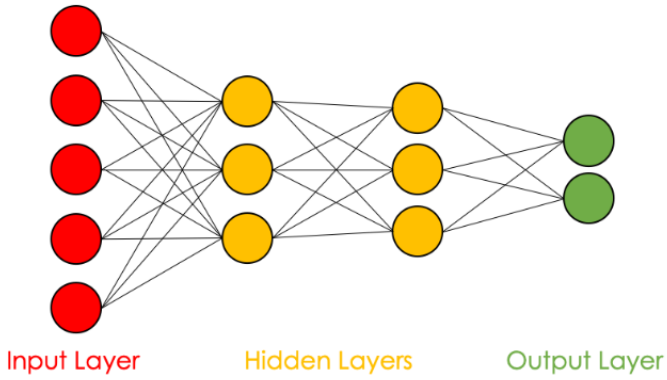
- Computation of `a`, which is equal to y at the output layer, using z

$$a = \psi(z)$$

w_i are the `weights`, b is the `bias` and ψ is said to be the `activation function`.

> *In general, neural networks better known as MLP, for 'Multi Layers Perceptron', is a type of direct formal neural network organized into several layers in which information flows from the input layer to the output layer only.*
>
> *Each layer consists of a defined number of neurons, we distinguish :*
>
> *- The input layer*
>
> *- The hidden layers*
>
> *- The output layer*

The following graph represents a neural network with 5 neurons at the input, 3 in the first hidden layer, 3 in the second hidden layer and 2 out.



Some variables in the hidden layers can be interpreted based on the input features: in the case of the house pricing and under the assumption that the first neuron of the first hidden layer pays more attention to the variables *x_1* et *x_2*, it can be interpreted as the quantification of the family size of the house for instance.

DL as a supervised task

In most DL problems, we tend to predict an output y using a set of variables X, in this case, we suppose that for each row of the database *X_i* we have the corresponding prediction *y_i*, thus the labeled data.

Applications: Real Estate, Speech Recognition, Image Classification …

The used data can be:

- Structured: explicit databases with features well defined
- Unstructured: Audio, Image, Text, …

Universal approximation theorem

Deep learning in real life is the approximation of a given function *f*. This approximation is possible and accurate thanks to the following theorem:

A multi-layer perceptron with a single hidden layer containing a finite number of neurons can approximate any continuous function f on compact$^{(*)}$ subsets of R^n.

The class of deep neural networks is a universal approximator $\iff$ the activation function is not polynomial.

(*) In finite dimension, a set is said to be compact if it is closed and bounded. Visit this link for more details.

The main take-out of this algorithm is that deep learning allows solving any problem which can be mathematically expressed

Data Preprocessing

In any machine learning project in general, we divide our data into 3 sets:

> *- **Train set**: used to train the algorithm and construct batches*
>
> *- **Dev set**: used to finetune the algorithm and evaluate bias and variance*
>
> *- **Test set**: used to generalize the error/precision of the final algorithm*

The following table sums up the repartition of the three sets according to the size of the data set m:

	Train	Dev	Test
$m = 10^4$	60%	20%	20%
$m = 10^6$	96%	2%	2%

Standard deep learning algorithms require a large dataset where the number of samples is around lines. Now that the data is ready we will see in the next section the training algorithm.
Usually, before splitting the data, we also normalize the inputs, a step detailed later in this article.

. . .

2. Learning algorithm

Learning in neural networks is the step of calculating the weights of the parameters associated with the various regressions throughout the network. In other words, we aim to find the best parameters that give the best prediction/approximation, starting from the input, of the real value.
For this, we define an objective function called the `loss function` and denoted `J` which quantifies the distance between the real and the predicted values on the overall

training set.
We minimize J following two major steps:

> - ***`Forward Propagation`*** *: we propagate the data through the network either in entirely or in batches, and we calculate the loss function on this batch which is nothing but the sum of the errors committed at the predicted output for the different rows.*
>
> - ***`Backpropagation`*** *: consists of calculating the gradients of the cost function with respect to the different parameters, then apply a descent algorithm to update them.*

We iter the same process a number of times called `epoch number` . After defining the architecture, the learning algorithm is written as follows:

- Initialization of the model parameters, a step equivalent to injecting noise into the model.
- `For` i=1,2...N: (N is the number of epochs)
  - Perform `forward propagation` :
    - $\forall i$, Compute the predicted value of x_i through the neural network: $\hat{y}_i^\theta$
    - Evaluate the function : $J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}_i^\theta, y_i)$
      where m is the size of the training set, θ the model parameters and $\mathcal{L}$ the cost$^{(*)}$ function
  - Perform `backpropagation` :
    - Apply a descent method to update the parameters :

$$\theta =: G(\theta)$$

(∗) The cost function L evaluates the distances between the real and predicted value on a single point.

. . .

3. Parameters' initialization

The first step after defining the architecture of the neural network is parameter initialization. It is equivalent to injecting initial noise into the model's weights.

- **Zero initialization:** one can think of initializing the parameters with 0's everywhere i.e: W=0 and b=0

Using the forward propagation equations, we note that all the hidden units will be symmetric which penalizes the learning phase.

- **Random initialization:** it's an alternative commonly used and consists of injecting random noise in the parameters. If the noise is too large, some activation functions might get saturated which might later affect the computation of the gradient.

Two of the most famous initialization methods are:

- `Xavier` 's: it consists of filling the parameters with values randomly sampled from a centered variable following the normal distribution:

$$\mathcal{N}(0, \frac{2}{n_i})$$

- `Glorot` 's: the same approach with a different variance:

$$\mathcal{N}(0, \frac{2}{n_i + n_{i+1}})$$

. . .

4. Forward and Backpropagation

Before diving into the algebra behind deep learning, we will first set the annotation which will be used in expliciting the equations of both the forward and the backpropagation.

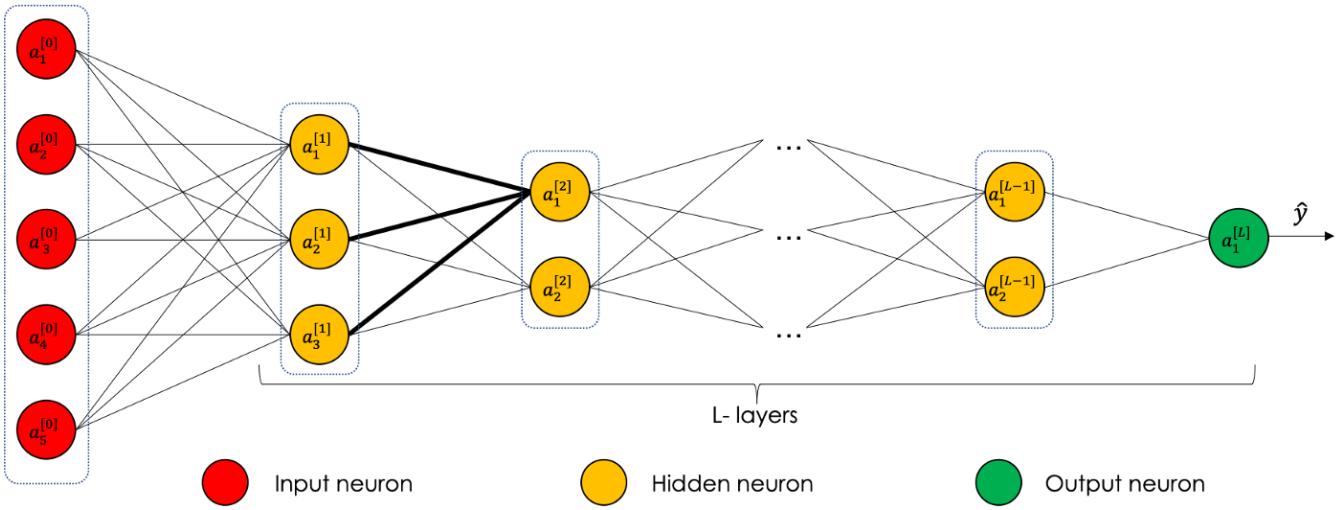
Neural Network's representation

The neural network is a sequence of `regressions` followed by an `activation function`. They both define what we call the forward propagation. and are the **learned parameters** at each layer. The backpropagation is also a sequence of algebraic operations carried out from the output towards the input.

Forward propagation

- **Algebra through the network**

Let us consider a neural network having `L layers` as follows:



We consider the 1^{st} node of the 2^{nd} hidden layer denoted $a_1^{[2]}$.
It's computed using the all the neurons of the previous layer as follows:

$$z_1^{[2]} = \sum_{l=1}^{3} w_{1,l}^{[2]} a_i^{[1]} + b^{[2]}$$
$$\rightarrow a_1^{[2]} = \psi^{[2]}(z_1^{[2]})$$

In general, considering the j^{th} node of the i^{th} layer we have the following equations:

$$z_j^{[i]} = \sum_{l=1}^{n_{i-1}} w_{j,l}^{[i]} a_l^{[i-1]} + b_j^{[i]}$$
$$\rightarrow a_j^{[i]} = \psi^{[i]}(z_j^{[i]})$$

with n_{i-1} being the number of neurons in the $(i-1)^{th}$ layer and W^T is the transpose of the matrix W.

Finally, we denote:

- $W^{[i]} = [w_1^{[i]}, w_2^{[i]}, .., w_{n_i}^{[i]}]$ where $dim(w_j^{[i]}) = [n_{i-1}, 1]$
- $b^{[i]} = {}^T[b_1^{[i]}, b_2^{[i]}, .., b_{n_i}^{[i]}]$
- $\mathcal{Z}^{[i]} = {}^T[z_1^{[i]}, z_2^{[i]}, .., z_{n_i}^{[i]}]; \mathcal{A}^{[i]} = {}^T[a_1^{[i]}, a_2^{[i]}, .., a_{n_i}^{[i]}]$
- $\mathcal{A}^{[i]} = \psi^{[i]}(\mathcal{Z}^{[i]}) = {}^T[\psi^{[i]}(z_1^{[i]}), \psi^{[i]}(z_2^{[i]}), .., \psi^{[i]}(z_{n_i}^{[i]})]$

Thus:

$$\mathcal{A}^{[i]} = \psi^{[i]}(\mathcal{Z}^{[i]}) = \psi^{[i]}({W^{[i]}}^T \mathcal{A}^{[i-1]} + b^{[i]})$$

where

$$dim(\mathcal{Z}^{[i]}) = dim(\mathcal{A}^{[i]}) = [n_i, 1]$$
$$dim({W^{[i]}}^T) = {}^T dim(W^{[i]}) = [n_i, n_{i-1}]$$

$$dim(b^{[i]}) = [n_i, 1]$$

Algebra through the training set

Let us consider the prediction of the output of a single row data frame, through the neural network.

We set $a^{[0]} = x^{(j)}$, at each layer $[i]$, we compute:

$$z^{[i][j]} = {W^{[i]}}^T a^{[i-1][j]} + b^{[i]} \text{ and } a^{[i][j]} = \psi^{[i]}(z^{[i][j]})$$

Until $\hat{y}^{(j)} = \psi^{[L]}(a^{[L]})$, where L is the number of layers

When dealing with a *m*-row data set, repeating these operations separately for each line is very costly.
We have, at each layer $[i]$:

$$z^{[i][1]} = {W^{[i]}}^T a^{[i][0]} + b^{[i]} \text{ and } a^{[i][1]} = \psi^{[i]}(z^{>[i][1]})$$
$$\vdots$$
$$z^{[i][m]} = {W^{[i]}}^T a^{[i][m-1]} + b^{[i]} \text{ and } a^{[i][m]} = \psi^{[i]}(z^{[i][m]})$$

We can use linear algebra to parallelize it as follows:

$$Z^{[i]} = {W^{[i]}}^T A^{[i-1]} + b^{[i]}$$
$$A^{[i]} = \psi^{[i]}(Z^{[i]})$$

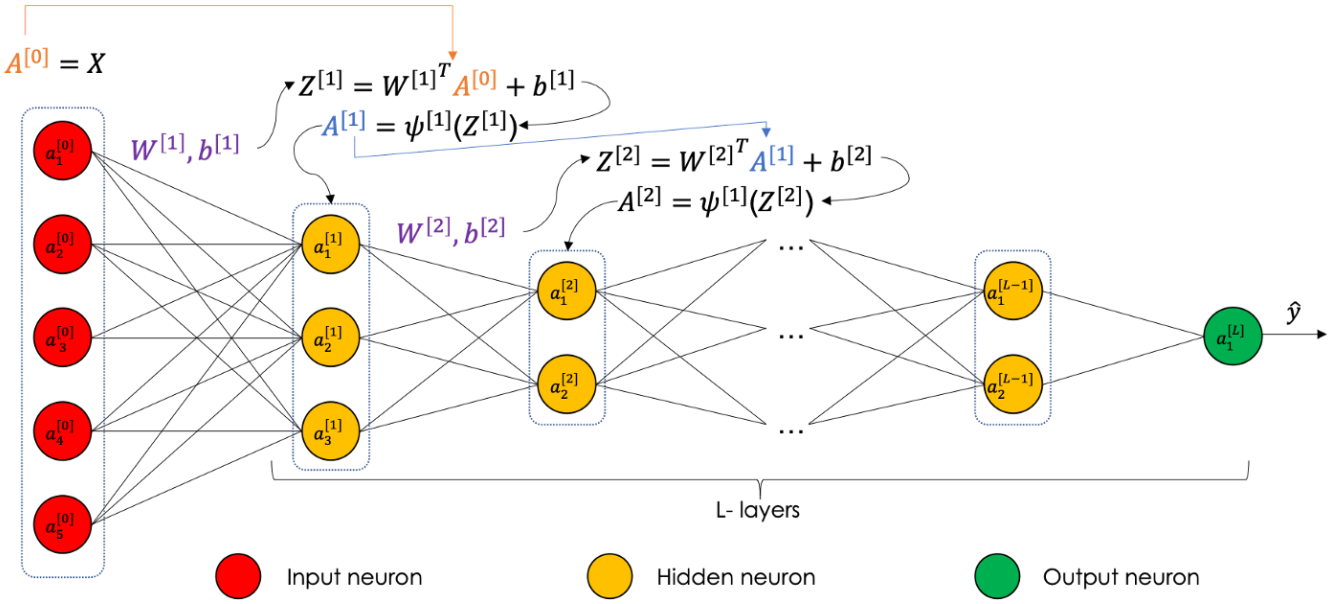
Considering n_i the number of neuron in the i^{th} layer:

$$Z^{[i]} = \left[z^{[i][j]}\right]_{(i,j)\in[n_i,m]}$$
$$A^{[i]} = \left[a^{[i][j]}\right]_{(i,j)\in[n_i,m]}$$

Where:

$$dim(Z^{[i]}) = dim(A^{[i]}) = [n_i, m]$$
$$dim({W^{[i]}}^T) = {}^T dim(W^{[i]}) = [n_i, n_{i-1}]$$
$$dim(b^{[i]}) = [n_i, 1]$$

The parameter b_i uses broadcasting to repeat itself through the columns. This can be summarized in the following graph:



Backpropagation

The backpropagation is the second step of the learning, which consists of injecting the error committed in the prediction (forward) phase into the network and update its parameters to perform better on the next iteration.
Hence, the optimization of the function J, usually through a descent method.

Computational graph

Most of the descent methods require the computation of the gradient of the loss function denoted $\nabla J(\theta)$.

In a neural network, the operation is carried out using a computational graph which decomposes the function J into several intermediate variables.

Let us consider the following function: $f(x,y,z) = (x+y).z$

The main objective is to calculate $\nabla f(x, y, z)$ in $(-2, 5, -4)$ where:

$$\nabla f(x, y, z) = {}^{T}\left[\frac{\partial f}{\partial x} \quad \frac{\partial f}{\partial y} \quad \frac{\partial f}{\partial z}\right]$$

Let $q = x + y \rightarrow f = q.z$

We carry out the computation using two passes:

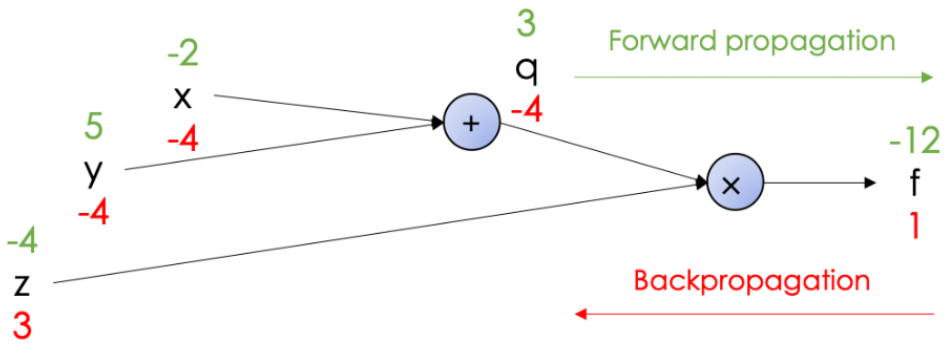
- **Forward propagation**: computes the value of f from inputs to output: $f(-2,5,-4)=-12$
- **Backpropagation**: recursively apply chain-rule to compute gradients from output to inputs:

$$\begin{aligned}
\frac{\partial f}{\partial f} &= 1 \\
\frac{\partial f}{\partial q} &= z = -4 \\
\frac{\partial f}{\partial z} &= q = 3 \\
\frac{\partial f}{\partial x} &= \frac{\partial f}{\partial q}.\frac{\partial q}{\partial x} + \frac{\partial f}{\partial z}.\frac{\partial z}{\partial x} = z.1 + q.0 = z = -4 \\
\frac{\partial f}{\partial y} &= \frac{\partial f}{\partial q}.\frac{\partial q}{\partial y} + \frac{\partial f}{\partial z}.\frac{\partial z}{\partial y} = z.1 + q.0 = z = -4
\end{aligned}$$

Hence:

$$\nabla f(x,y,z)|_{(-2,5,-4)} = {}^{T}\begin{bmatrix} -4 & -4 & 3 \end{bmatrix}$$

The derivatives can be resumed in the following *computational graph*:



Equations

Mathematically, we compute the gradients of the cost function J, w.r.t the architecture's parameters W and b.

For a given parameter α, we set $d\alpha^{[i]} = \frac{\partial J}{\partial \alpha^{[i]}}$ and we have at the i^{th} layer:

$$\begin{aligned}
dZ^{[i]} &= dA^{[i]} \star \psi'^{[i]}(Z^{[i]}) \\
dA^{[i-1]} &= {W^{[i]}}^{T} dZ^{[i]} \\
dW^{[i]} &= dZ^{[i]} A^{[i-1]} \\
db^{[i]} &= dZ^{[i]}
\end{aligned}$$

where $(\star)$ is the element-wise multiplication.
We recursively apply these equations for $i=L, L-1,\ldots,1$

Gradient Checking

When carrying out the backpropagation, an additional checking is added to make sure that the algebraic computations are correct.

Algorithm:

- We first reshape and stack all the parameters $W^{[i]}$ and $b^{[i]}$ into one vector denoted θ
- We carry out the same manoeuvre for their derivatives $dW^{[i]}$ and $db^{[i]}$ and we denote $d\theta$ the resulting vector.
- $\forall i$, We compute:

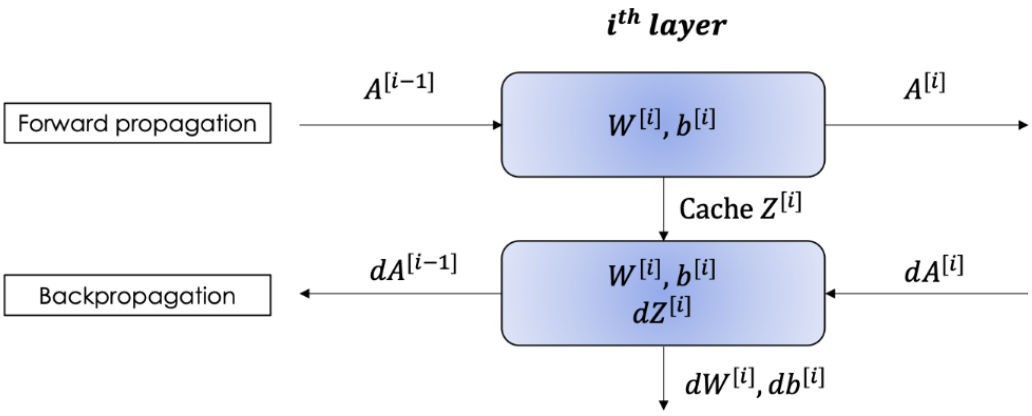
$$d\theta^{[i]}_{approx} = \frac{J(\theta_1, \theta_2, ..., \theta_i + \epsilon, ...) - J(\theta_1, \theta_2, ..., \theta_i - \epsilon, ...)}{2\epsilon}$$

  an $O(\epsilon^2)$ approximation of $\frac{\partial J}{\partial \theta_i} = d\theta^{[i]}$ (where ϵ is very small $\approx 10^{-7}$)
- We check the following quantity:

$$\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$$

It should be close to the value of ϵ, an error is suspected when the value of the quantity is a *thousand* times higher than ϵ.

We can sum up the Forward and Backward propagation in the following block:



Parameters vs Hyperparameters

-*Parameters*, denoted θ, are the elements that we learn through the iterations and on which we apply backpropagation and update: W and b.

-*Hyperparameters* are all the other variables we define in our algorithm which can be *tunned* in order to improve the neural network:

- Learning rate α
- Number of iterations
- Choice of activation functions
- Number of layers L
- Number of units in each layer

. . .

5. Activation functions

Activation functions are a kind of transfer functions that select the data propagated in the neural network. The underlying interpretation is to allow a neuron in the network to propagate learning data (if it is in a learning phase) only if it is sufficiently excited.

Here is a list of the most common functions:

- **ReLU**:

$$\psi(x) = x\mathbf{1}_{x\geq 0}$$

- **Sigmoid**:

$$\psi(x) = \frac{1}{1+e^{-x}}$$

- **Tanh**:

$$\psi(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$$

- **LeakyReLU**:

$$\psi(x) = x\mathbf{1}_{x\geq 0} + \alpha x\mathbf{1}_{x\leq 0}$$

Remark: if the activation functions are all linear, the neural network is precisely equivalent to a simple linear regression

. . .

6. Optimization algorithm

Risk

Let us consider a neural network denoted by f. The real objective to optimize is defined as the expected loss over all the corpora:

$$R(f) = \int p(X,Y)\mathcal{L}(f(X),Y)dXdY$$

Where X is an element from a continuous space of observables to which correspond a target Y and $p(X,Y)$ being the marginal probability of observing the couple (X,Y).

Empirical risk

Since we can not have all the corpora and hence we ignore the distribution, we restrict the estimation of the risk on a certain dataset well representative of the overall corpora and consider all the cases equiprobable.
In this case: and where m is the size of the representative corpora. Hence, we iteratively optimize the loss function defined as follows:

Plus we can assert that:$\int = \sum$ and $p(X,Y) = 1/\text{m}$ where m is the size of the representative corpora. Hence, we iteratively optimize the loss function defined as follows:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}_i^{\theta}, y_i)$$

Plus we can assert that:

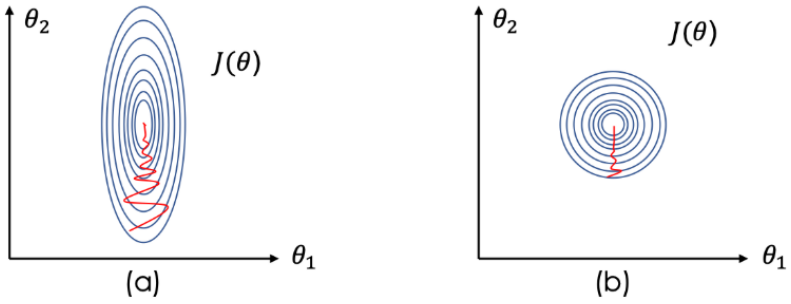
$$min_f R(f) \approx min_\theta J(\theta)$$

There exist many techniques and algorithms, mainly based on gradient descent, which carries out the optimization. In the sections below, we will go through the most famous

ones. It is important to note that these algorithms might get stuck in local minima and nothing assures reaching the global one.

Normalizing inputs

Before optimizing the loss function, we need to normalize the inputs in order to speed up the learning. In this case, $J(\theta)$ becomes tighter and more symmetric which helps gradient descent to find the minimum faster and thus in fewer iterations.
Standard data is the commonly used approach which consists of subtracting the mean of the variables and dividing by their standard deviation. Considering, the following image illustrates the effect of normalizing the input on the contour lines of -standard data on the right-:



Let X be a variable in our database, we set:

$$X := \frac{X-\mu}{\sigma}$$

Where $\mu = \frac{1}{m}\sum_{i=1}^{n} x^{(i)}$ and $\sigma = \frac{1}{m}\sum_{i=1}^{n}(x^{(i)} - \mu)^2$

Gradient descent

In general, we tend to construct a `convex` and `differentiable` function J where any local minima is a global one. Mathematically speaking finding the global minimum of a convex function is equivalent to solving the equation $\nabla J(\theta)=0$, we denote $\theta\star$ its solution.
Most of the used algorithms are of the kind:

$$\theta_{k+1} = \theta_k + \alpha_k d_k$$

with θ_0 an initial guess, where α_k is the step size and d_k the descent direction.
We can assert that:

$$J(\theta_{k+1}) = J(\theta_k) + \alpha_k \nabla J(\theta_k) d_k + o(\theta_k)$$

$J(\theta_{k+1}) = J(\theta_k) + \alpha_k \nabla J(\theta_k) d_k + o(\theta_k)$

Since we seek to have $J(\theta_{k+1}) << J(\theta_k)$ then we need $\nabla J(\theta_k) d_k$ as negative as possible, meaning $d_k = -\nabla J(\theta_k)$.

Algorithm:

- θ_0 is given
- for $k = 1, ...,$stopping criterion:
  - $\theta_{k+1} = \theta_k - \alpha_k \nabla J(\theta_k)$

Choice of α_k:

- $\alpha_k = \alpha$ a fixed step size
- α_k minimizes $t \to J(\alpha_k - t \nabla J(\theta_k))$
- α_k follows a certain decay law (see Learning rate decay section)

Mini-batch gradient descent

This technique consists of dividing the training set to batches:

Given the batches $(X^{\{1\}}, y^{\{1\}}), (X^{\{2\}}, y^{\{2\}}), ..., (X^{\{n\}}, y^{\{n\}})$

- for t=1,...,n:
  - Carry out forward propagation on $X^{\{t\}}$
  - Compute the cost function normalized on the size of the batch
  - Carry out the backpropagation using $(X^{\{t\}}, y^{\{t\}}, \hat{y}^{\{t\}})$
  - Update the weight $W^{[l]}$ and $b^{[l]}; \forall l$

Choice of the mini-batch size:

- A small number of rows ~2000 lines
- Typical size: the power of 2 which is good for memory
- Mini-batch should fit in CPU/GPU memory

Remark: in the case where there is only one data line in the batch, the algorithm is called stochastic gradient descent

Gradient descent with momentum

A variant of gradient descent which includes the notion of momentum, the algorithm is as follows:

- Initialize $V_{dW} = 0_{dW}, V_{db} = 0_{db}$
- On iteration k:
  - Compute dW and db on the current mini-batch
  - $V_{dW} = \beta V_{dW} + (1-\beta)dW; V_{db} = \beta V_{db} + (1-\beta)db$
  - Update the parameters:
    - $W := W - \alpha dW$
    - $b := b - \alpha db$

(α,β) are hyperparameters.
Since $d\theta$ is calculated on a mini-batch, the resulting gradient ∇J is very noisy, this exponentially weighted averages included by the momentum give a better estimation of derivatives.

RMSprop

Root Mean Square prop is very similar to gradient descent with momentum, the only difference is that it includes the second-order momentum instead of the first-order one, plus a slight change on the parameters' update:

- Initialize $S_{dW} = 0_{dW}, S_{db} = 0_{db}$
- On iteration k:
  - Compute dW and db on the current mini-batch
  - $S_{dW} = \beta S_{dW} + (1-\beta)dW^{\mathbf{2}}; S_{db} = \beta S_{db} + (1-\beta)db^{\mathbf{2}}$
  - Update the parameters:
    - $W := W - \frac{\alpha}{\sqrt{S_{dW}}+\epsilon}dW$
    - $b := b - \frac{\alpha}{\sqrt{S_{db}}+\epsilon}db$

(α,β) are hyperparameters and ϵ assures numerical stability ($\approx 10-8$)

Adam

Adam is an adaptive learning rate optimization algorithm designed specifically for training deep neural networks. Adam can be seen as a combination of RMSprop and gradient descent with momentum.
It uses square gradients to set the learning rate at scale as RMSprop and takes advantage of momentum by using the moving average of the gradient instead of the gradient itself as the gradient descends with momentum.
The main idea is to avoid oscillations during optimization by accelerating the descent in the right direction.
The algorithm of Adam optimizer is the following:

- Initialize: $V_{dW} = 0, S_{dW} = 0, V_{db} = 0, S_{db} = 0;$
- On iteration k:
  - Computation of dW and db through backpropagation
  - Momentum:
    - $V_{dW} = \beta_1 V_{dW} + (1 - \beta_1) dW$
    - $V_{db} = \beta_1 V_{db} + (1 - \beta_1) db$
  - RMSprop:
    - $S_{dW} = \beta_2 S_{dW} + (1 - \beta_2) dW^2$
    - $S_{db} = \beta_2 S_{db} + (1 - \beta_2) db^2$
  - Correction:
    - $V_{dW} = \frac{V_{dW}}{1 - \beta_1^k}$
    - $S_{dW} = \frac{S_{dW}}{1 - \beta_2^k}$
    - $V_{db} = \frac{V_{db}}{1 - \beta_1^k}$
    - $S_{db} = \frac{S_{db}}{1 - \beta_2^k}$
  - Parameters' update:
    - $W = W - \alpha \frac{V_{dw}}{\sqrt{S_{dW}} + \epsilon};$
    - $b = b - \alpha \frac{V_{db}}{\sqrt{S_{db}} + \epsilon}$

Learning rate decay

The main objective of the learning rate decay is to slowly reduce the learning rate over time/iterations. It finds justification in the fact that we afford to take big steps at the beginning of the learning but when approaching the global minimum, we slow down and thus decrease the learning rate.
There exist many learning rate decay laws, here are some of the most common:

- We decrease the learning rate by epoch i.e 1 pass through the data (all the mini-batches):

$$\alpha(epoch\_num) = \frac{1}{1 + \beta . epoch\_num} \alpha_0$$

- We can exponentially decrease the learning rate:

$$\alpha(epoch\_num) = 0.95^{epoch\_num} \alpha_0$$

- We can also consider the following decay law:

$$\alpha(epoch\_num) = \frac{k}{\sqrt{epoch\_num}} \alpha_0$$

(α_0, k, β) are hyperparameters

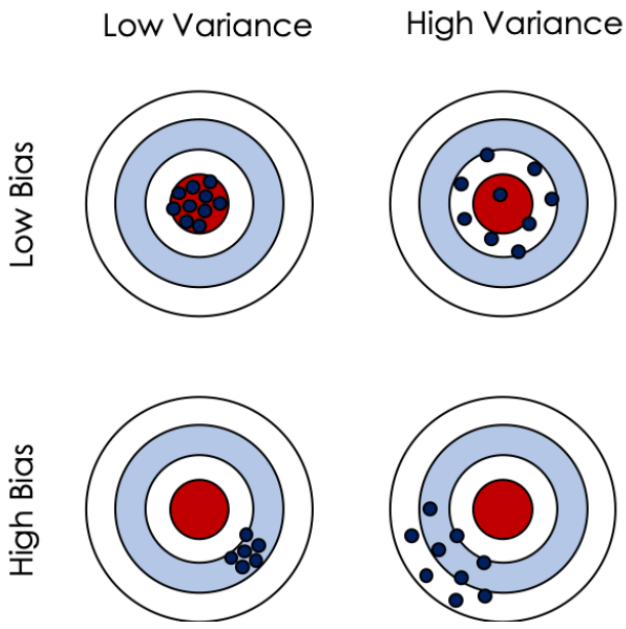
Regularization

Variance/bias

When training a neural network, it might suffer from:

- **High bias**: or underfitting, where the network fails to find the path in the data, in this case, *J_train* is very high the same as *J_dev*. Mathematically speaking, when performing cross-validation; the mean of *J* on all the considered folds is high.
- **High variance** or overfitting, the model fits perfectly on the training data but fails to generalize on unseen data, in this case, *J_train* is very low and *J_dev* is relatively high. Mathematically speaking, when performing cross-validation; the variance of *J* on all the considered folds is high.

Let's consider the dartboard game, where hitting the red target is the best-case scenario. Having a **low bias** (first line) means that **on average** we are close to the goal. In case, of a **low variance,** the hits are all concentrated around the target (the variance of the hits' distribution is low). When the variance is high, under the assumption of a low bias, the hits are spread out but still around the red circle.
Vice-versa, we can define the high bias with a low/high variance.



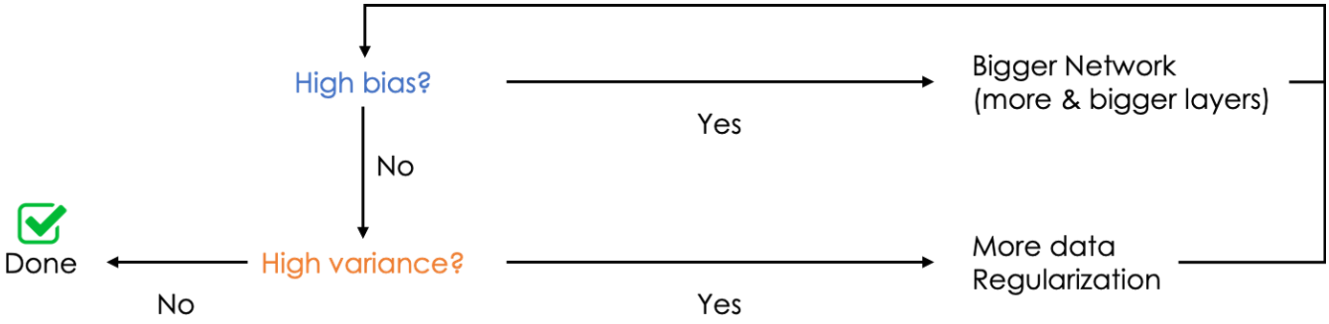
Mathematically speaking, let f be a true regression function: $y=f(x)+\epsilon$ where: $\epsilon \sim N(0,\sigma^2)$

We fit a hypothesis $h(x)=Wx+b$ with MSE and consider x_0 be a new data point, y_0 $=f(x\_0)+\epsilon$: the expected error can be defined by:

$$\mathbb{E}[(y_0 - h(x_0))^2] = \mathbb{E}[(h(x_0) - \bar{h}(x_0))^2]\textbf{(Variance)} + (\bar{h}(x_0) - f(x_0))^2\textbf{(bias)} + \mathbb{E}[(y_0 - f(x_0))^2]\textbf{(Intrinsic)}$$

where $\bar{Z} = \mathbb{E}[Z]$

A trade-off must be found between variance and bias to find the optimum complexity of the model either by using the *AIC* criteria or using cross-validation.
Here is a simple schema to follow to solve bias/variance issues:



L1 — L2 regularization

Regularization is an optimization technique that prevents overfitting.
It consists of adding a term in the objective function to minimize as follows:

- L1 regularization: J becomes:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} cost(\hat{y}_i^{\theta}, y_i) + \frac{\lambda}{2m}\|\theta\|_1^2$$

Where $\|\theta\|_1 = \sum_i |\theta^{[i]}|$

- L2 regularization: J becomes:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} cost(\hat{y}_i^{\theta}, y_i) + \frac{\lambda}{2m}\|\theta\|_2^2$$

Where $\|\theta\|_2^2 = \theta^T\theta$

λ is the hyperparameter of the regularization.

Backpropagation and regularization

The update of the parameters during backpropagation depends on the gradient ∇J, to which is added a new regularization term. In L2 regularization, it becomes as follows:

$$d\theta^{reg} = d\theta + \frac{\lambda}{m}\theta \rightarrow \theta := \theta(1 - \frac{\lambda}{m}\alpha) - \alpha d\theta$$

Considering $\lambda >> 1$, minimizing the cost function leads to weak values of parameters because of the term $(\lambda/2m)\|\theta\|$ which simplifies the network and makes more consistent, hence less exposed to overfitting.

Dropout regularization

Roughly speaking, the main idea is to sample a uniform random variable, `for each layer for each node`, and have p chance of keeping the node and $1-p$ of removing it which diminishes the network.
The main intuition of dropout is based on the idea that the network shouldn't rely on a specific feature but should instead spread out the weights!
Mathematically speaking, when dropout is off and considering the *j th* node of the *i th* layer, we have the following equations:

$$\begin{gathered} z_j^{[i]} = {W_j^{[i]}}^T \mathcal{A}^{[i-1]} + b_j^{[i]} \\ \rightarrow a_j^{[i]} = \psi^{[i]}(z_j^{[i]}) \end{gathered}$$

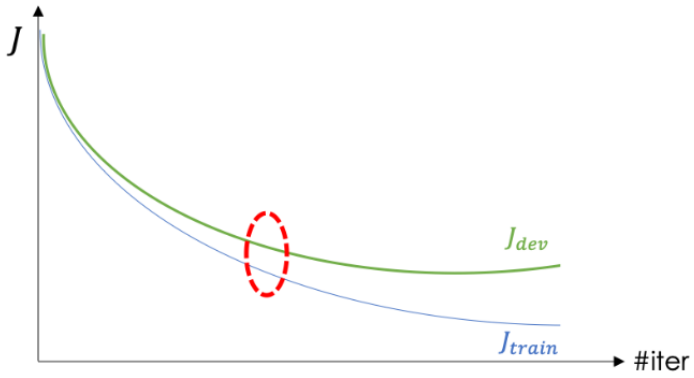
When dropout is on, the equations become as follows:

$$\begin{gathered} r_j^{[i-1]} \sim Bernoulli(p^{(i-1)}) \\ \hat{\mathcal{A}}^{[i-1]} = \mathcal{A}^{[i-1]}.r_j^{[i-1]} \\ \hat{z}_j^{[i]} = {W_j^{[i]}}^T \hat{\mathcal{A}}^{[i-1]} + b_j^{[i]} \\ \rightarrow a_j^{[i]} = \psi^{[i]}(\hat{z}_j^{[i]}) \end{gathered}$$

Where $p^{(i-1)}$ is a hyperparameter.

Early stopping

This technique is quite simple and consists of stopping the iteration around the area when *J_train* and *J_dev* start separating:



Gradient problems

The computation of gradients suffers from two major problems: gradient vanishing and gradient exploding.
To illustrate both of the situations, let's consider a neural network where all the activation functions $\psi[i]$ are linear and:

$$W^{[i]} = \begin{bmatrix} 1,5 & 0 \\ 0 & 1,5 \end{bmatrix}, b^{[i]} = 0, \forall i = 1, ..., L-1$$

Thus:

$$\hat{y} = W^{[L]}. \begin{bmatrix} 1,5^{L-1} & 0 \\ 0 & 1,5^{L-1} \end{bmatrix}$$

We note that 1,5^($L-1$) will explode exponentially as a function of the depth L. If we use 0.5 instead of 1,5 then 0,5^(L-1) will vanish exponentially as well.
The same issue occurs with gradients.

Conclusion

As a data scientist, it is very important to be aware of the mathematics turning in the background of the neural networks. This allows better understanding and faster debugging.
Happy Machine Learning!

References

- Deep Learning Specialization, Coursera, Andrew Ng
- Optimization course, Mines Nancy, Antoine Henrot

- Machine Learning, Loria, Christophe Cerisara

. . .

Originally published at https://www.ismailmebsout.com on January 31, 2020.

Deep Learning Optimization Data Science Machine Learning Mathematics

About Help Legal