CM Project Report

Valerio Mariani. Group Id:1, Project Id:5, ML

March 2021

Introduction/Abstract

The project consists of a Neural Network with arbitrary topology and activation function, but mandatory l1 regularization, trained with:

- (A1) A standard momentum descent approach
- (A2) An algorithm of the class of proximal bundle methods

I start with the definition of a Multi-Layer Perceprton (MLP), and the computation of loss function and its gradient, then i introduce the two training methods and test them.

Contents

Chapter 1

Architecture of The MLP

Each network is made out of a **stack of layers of units**, in particular it has:

- N_l **hidden layers**, each of which has N_h^m units, with $m \in [1, N_l]$. Those have arbitrary activation function.
- Two layers with respectively N_u and N_y **input and output units**. Those units usually have identity function as activation function.[1]

To distinguish activation functions of the hidden layer with those of input and output ones, i refer to (pointwise) activation function of layer m as f^m.

To connect units one to the other, i use **weighted connections** (directed arcs). Each connection is referred to as w_{ij}^m, and represents the connection that goes from unit j of the layer m to unit i of layer m+1. This way i can **accumulate connections** between units of layer m and units of layer m+1 **in matrices** $w^m = \{w_{ij}^m$ for i $\in [0 : N_h^{m+1}]$, $j \in [0, N_h^m]\}$. In more, to do regularization i need to **compute the norm** of the vector that contains all of these parameters, so i simply take all the w^m, I **flatten** them into a 1-dimensional vector w and compute its norm.

Each unit has its own **activation and potential**, and i denote activation and potential of the units of layer m as $a^m \in \mathbb{R}^{N_h^m \times 1}$ and $v^m \in \mathbb{R}^{N_h^m \times 1}$, thereby $a^m = f^m(v^m)$, where potential v_i^m of the i-th unit of layer m is the weighted sum of the activations of units connected to the unit i of layer m, indeed i **compute them using the formula** $v^m = w^{m-1}a^{m-1}$.

To wrap up the notation that i introduced up to here:

- N_l is number of hidden layers
- N_h^m is number of units in m-th hidden layer
- N_u and N_y are respectively the number of input and output units

[1]Output units can have softmax activation in the case of classification tasks with cross-entropy loss function

- f^m is activation function of layer m
- w^m is matrix of weights connecting units in layer m-1 to those in layer m
- w is the vector that contains all the weights w_{ij}^m
- $v^m = w^{m-1}a^{m-1}$ is potential of activation of units in layer m
- $a^m = f^m(v^m)$ is activation of units in layer m

with the basic structure of the MLP defined, i use the network feeding the input units with external input, then iteratively compute activation of units of consecutive layers up to the output layer, and finally consider the output of my network as activation of the output units.

In more, i **fix a set X of samples** made out of inputs and relative desired outputs and compute the loss of my network using an arbitrary **loss function**, denoted as $l(w, X)$, that is composed of a term that mesures the **error** that the network does in his task **plus an optional l1-norm penality term** for regularization.

The aim of the project is to fix a set X of patterns to do supervised training and solve the optimization problem of minimizing the loss function $l(w, X) = l(w)$.

In order to do this, i need to compute the **gradient of my loss function w.r.t. w**, and i compute it using the standard techniques introduced in the Back-Propagation algorithm.

Chapter 2

BackPropagation & gradient computation

I want to minimize my loss

$$l(w) = E + \epsilon ||w||_1 = (\sum_{p=1}^{size(P)} E_p) + \epsilon ||w||_1 \tag{2.1}$$

where E is the total error computed **summing errors E_p made by the network over each pattern** p of X, so the gradient $\nabla l(w)$ is made out of the **derivatives of loss** $l(w)$ **w.r.t each** w_{ij}^m:

$$\frac{\partial l(w)}{\partial w_{ij}^m} = \left(\sum_{t=1}^{T} \frac{\partial E_p}{\partial w_{ij}^m} \right) + \frac{\partial\, \epsilon ||w||_1}{\partial w_{ij}^m}. \tag{2.2}$$

Now i have to compute the two main components of this summation: i start with the **computation of derivative of error on pattern p**. I use the **chain rule** to derive:

$$\frac{\partial E_p}{\partial w_{ij}^m} = \frac{\partial E_p}{\partial v_i^{m+1}} \frac{\partial v_i^{m+1}}{\partial w_{ij}^m} \tag{2.3}$$

that becomes:

$$\frac{\partial E_p}{\partial w_{ij}^m} = \delta_i^m a_j^{m-1}, \tag{2.4}$$

where δ_i^m is the so-called **error-propagation-term**, a_j^{m-1} is activation of the unit as described in the previous chapter, and both of them are computed feeding the network with p-th pattern of X.

I compute error-propagation term of unit i in layer m during submission of pattern p as:

$$\delta_i^{Nl+1} = \frac{\partial E_p}{\partial a_i^m} f^{m'}(v_i^m) \quad \textbf{if} \quad \textbf{m} = \textbf{Nl+1}, \tag{2.5}$$

$$\delta_i^m = -(\sum_k w_{ki}^m \delta_k^{m+1}) f^{m'}(v_i^m) \quad \textbf{otherwise.} \tag{2.6}$$

For the purpose of this report, i can assume E_p is **sum of squared error** $E_p(d, y = x^{Nl+1}) = ||d - y||_2^2$ (with d desired output) because that will be always used later, so formula 2.5 becomes:

$$\delta_i^m = -(d_i - a_i^m) f^{m'}(v_i^m) \quad \text{with } m = Nl + 1. \tag{2.7}$$

In **matrix terms**, the computation of the error propagation terms becomes:

$$\delta^{Nl+1} \in \mathbf{R}^{Ny \times 1} = (d - x^{Nl+1}) f^{Nl+1'}(v^{Nl+1}), \tag{2.8}$$

$$\delta^m \in \mathbf{R}^{N_h^m \times 1} = w^{m^t} f^{m'}(v^m), \tag{2.9}$$

and then i can compute the **matrices of the derivatives of error** E_p w.r.t. each w_{ij}^m as:

$$\frac{\partial E_p}{\partial w^m} = \{\frac{\partial E_p}{\partial w_{ij}^m}\} = \delta^{m+1} a^{m^t}. \tag{2.10}$$

Now for **regularization's norm-1 penality term** in equation 2.2, the problem with this term is that $||x||_1$ is **non-differentiable** when the element(s) $x_i = 0$, so in the case of a 0-weight i would actually need a **subdifferential or rather something that belongs to it**, that i could obtain e.g. by putting $\frac{\partial ||x||_1}{\partial x_i} = 0$. Although, in the case of machine computation, an exact 0 is **practically impossible** to obtain trough floating-point-operations, so i think the **two main routes** i can take to tackle the almost-0-weights problem are:

1. i **manually put each weight to 0** when the lasso regularization sends it **below some threshold**, and then when i go and compute the gradient i actually take an element in the subdifferential, putting $\frac{\partial ||w^m||_1}{\partial w_{ij}^m} = 0$. This way, i can technically obtain exactly-0-weights through l1-regularization, but on the other hand i would need an **hyper-parameter for the threshold** and the calculation would probably become even less exact.

2. i **assume that i am never going to encounter an exact-0-weight** and so i do not consider this case in the gradient computation, and i use the formula in equation 2.11:

$$\frac{\partial \epsilon ||w||_1}{\partial w_{ij}^m} = \epsilon \frac{\partial ||w||_1}{\partial w_{ij}^m} = \epsilon \, sign(w_{ij}^m). \tag{2.11}$$

   This way i expect the lasso regularization to **send some weights towards zero** until they reach a point where they are close in absolute value to the epsilon parameter, and then if the training does not stop, those weights start alternatively changing sign at each iteration.

At the end i **choose to take the second route** because i think manually putting weights to 0 when they fall below an arbitrary threshold would be too inexact of a procedure, and even supposing that i could find some appropriate treshold (e.g. 10^{-16}), **depending on the ϵ parameter, the conditions for putting weights to 0 would anyway verify very rarely**, resulting in a lot of pointless checks.

To wrap up, to compute the partial gradient of the error function on a pattern p of X, I do the following steps:

1. **Forward Pass**: I **supply the network with the input in pattern p** and iteratively **compute activation** and potentials of units in each layer, proceeding **forward** through the network,
2. **Backward Pass**: I **use the desired output** of pattern p to compare it to the output of the network and **compute the error propagation terms** of units of each layer, proceeding backwards through the network,
3. **Actual computation of partial gradient of error on pattern p**: i use activations, potentials and propagation-terms to **compute the partial gradient**,

Then i **sum up the partial derivatives** of errors and and **add the regularazion term** - computed as in equation 2.11 - like in equation 2.2 to come up with the gradient of my loss function.

Finally, to **compute the norm of my gradient** i **flatten** all the components into a 1-dimensional vector and compute its norm.

Chapter 3

(A1) Momentum descent

As for **momentum-based gradient descent** approach, I started from the formulae described in [4] for **Classical Momentum (CM)**:

$$d = -\alpha_{CM} \nabla f(x_i) + \beta_{CM} d^{-}, \tag{3.1}$$

$$x_{i+1} = x_i + d, \tag{3.2}$$

where d and d^{-} are the direction respectively chosen at step i and i-1, while α_{CM} and β_{CM} are two parameters and ∇f is the gradient of the function f to be optimized.

The two fixed parameters α_{CM} and β_{CM} in Formulae 3.1 and 3.2 suggest **either** using an **ML-ish approach** (**grid-search** in hyper-parameters' space) **or** using the **closed formulae** cited in the CM course when approaching **Heavy ball method and estimation of the best possible values of α_{CM} and β_{CM} parameters at each step using eigenvalues of the hessian matrix** (Strategy that i believe to be at least inconvenient in the case of the loss function of a neural network, even with involved strategies for estimation of eigenvalues of the hessian matrix, in particular if the loss function is non-differentiable).

Because **neither the approaches satisfied me**, i tried to think of a **way to mix in** some sort of **line-search** that could replace the fixed step-sizes proposed for CM and speed up convergence, but I came to the conclusion that **adding a line-searching mechanism to the CM method could not result in much more than a brutally simplified version of Conjugate Gradient (CG)**; indeed i think a natural comparison can be made beetween CM and CG methods.

CG method can be written like in formulae 3.3 and 3.4:

$$d = -\nabla f(x_i) + \beta_{CG} d^{-}, \tag{3.3}$$

$$x_{i+1} = x_i + \alpha_{CG} d, \tag{3.4}$$

where the parameters α_{CG} and β_{CG} can be computed at each iteration respectively by a line search and closed formulae (like for example Fletcher-Reeves).
The comparison can be made in the sense that one **can obtain CG from CM** by setting

$alpha_{CM}$ to 1, adding a line-searched step-size coefficient α_{CG} in formula 3.2, and computing a different β_{CM} at each iteration with (e.g.) Fletcher-Reeves.

The main differences i can identify are that the hyper-parameter selection for CG is generally more expensive than the one for CM because one has to set more hyper-parameters[1]. At the same time, CG theoretically promises a (possibly[2]) much faster convergence near a minimum[3] in change of an higher[4] computational cost per-iteration. At the end of the day, given that one manages to properly set the choosen method, the number of function/gradient-computation needed to converge can be very high for both methods, and both methods will for sure behave differently on different practical applications, especially because CG has been seen to behave much differently depending on the formulae used for β_{CG} computation.

All of this said, i decided to simply **opt for ML-ish approach** and select the hyper-parametrical fixed step-sizes by **grid-searching**.

As for **convergence** of the CM method, we know that CM (a.k.a. heavy ball method) has been proven in [1] to be **globally convergent for optimization of convex functions with Lipschitz continuous gradient** (with linear convergence if the function has also strong convexity) but unfortunately this is not the case for loss function of neural networks. Although, **local convergence** can be proven in the case of **non-convex optimization** [2], and [3] showed that CM can considerably accelerate convergence to a local (or global) minimum with respect to steepest descent method.

In my case, the function $f(x)$ to be minimized is the loss function $l(w)$, so $\nabla l(w)$ is the gradient of the loss function w.r.t weights of the network, computed as in chapter 2.

Overall, my training routine is described by Algorithm 1.

[1]I think of line-search and selection of the formula for β_{CG} at least

[2]Not always much faster because speed of theoretical convergence is strongly conditioned by the number of parameters to be optimized, and those can be a lot in the case of a neural network

[3]In the sense that CG is shown to converge n-step quadratically when a minimum is near

[4]Indefinitely higher, because the cost of an iteration depends on the time spent on each line-search, which in turn depends on the nature of the function and on the parameters of the line-search. Has to be said that if one was to quit line-searching to save some number of function/gradient-computations, he would have to pay back by erratically proceeding in the search space by probably too short/long constant step-sizes.

Algorithm 1: Classical Momentum training

```
Input:
$w$: weights of the network to be trained
$train_x$: input patterns
$train_y$: desired output patterns
$\alpha$: learning rate
$\beta$: momentum parameter
$\epsilon$: regularization parameter
$\omega$: stopping treshold
$e_{max}$: maximum number of epochs to be done
1 Procedure MomentumTrain($w$, $train_x$, $train_y$, $\alpha$,$\beta$,$\epsilon$,$\omega$,$e_{max}$):
2     $e = 0$, $v_0 = 0$, $t = 1$,
3     while $||\nabla l(w_t)||/||\nabla l(w_0)|| > \omega$ and $t < e_{max}$ do
4         $d =$ -$\alpha\nabla l(w_t) + \beta d^-$
5         $w = w + d$
6         $t = t + 1$, $d^- = d$
```

Chapter 4

(A2) Proximal Bundle Method

In this case i define a **bundle** β of information about my loss function (encoded in a series of planes that are tangent to the function to be optimized), that will be **increasingly enriched** during the flow of the algorithm.
The bundle β is used by a **bundle function** as described in equation 4.1.

$$f_\beta(x) = max\{f_i + g_i^t(x - x_i) : (x_i, f_i, g_i) \in \beta\} \tag{4.1}$$

Equation 4.1 **defines a model of the loss function that gets better and better as the algorithm proceeds**.

To find the next iterate, i have to solve the optimization problem of finding $min\{f_\beta(x)\}$, and because equation 4.1 defines a **polyhedral function**, my problem can be written as a **linear program** formulated like in equation 4.2, making it easy to solve:

$$min\{f_\beta(x)\} = min\{v : v \geq f_i + g_i^t(x - x_i), (x_i, f_i, g_i) \in \beta\}. \tag{4.2}$$

To turn the method described - that is known as Cutting Plane Method (CPM) - **into Proximal Bundle Method (PBM)** i need to **stabilize** the linear program described in equation 4.2 adding a stabilization term with relative stability parameter μ, obtaining the so called **Master Problem (MP)**, as in equation 4.3:

$$min\{f_\beta(x) + \mu||x - \bar{x}||_2^2/2\} = min\{v + \mu||x - \bar{x}||_2^2/2, v \geq f_i + g_i^t(x - x_i), (x_i, f_i, g_i) \in \beta\} \tag{4.3}$$

Because MP is a **quadratic program** to be optimized by **constrained optimization**, one can just use an off the shelf solver (in my case CVXPY) to (relatively[1]) efficiently **find the optimal solution (stability center) of MP at each iteration**.

All in all, my training routine for PBM method is described in algorithm 2:

[1]In the sense that ad-hoc strategies exist

Algorithm 2: Proximal Bundle Method

```
Input:
$w$: weights of the network to be trained
$train_x$: input patterns
$train_y$: desired output patterns
$\epsilon$: regularization parameter
$\omega$: stopping treshold
$m1$: parameter for armijo-type SS/NS decision
$\mu$: initial stability parameter
$\tau$: rescaling coefficent for stability parameter, $< 1$
$e_{max}$: maximum number of epochs to be done
1  Procedure PBM($w$, $train_x$, $train_y$, $\epsilon$,$\omega$,$m1$,$\mu$,$\tau$,$e_{max}$):
2      $e = 0$, $\bar{w} = w$
3      $\beta = \{(\bar{w}, l(\bar{w}), \nabla l(\bar{w}))\}$
4      while $true$ do
5          $w^* = argmin\{f_\beta(\bar{w}) + \mu||w - \bar{w}||_2^2/2\}$
6          if $\mu||w^* - \bar{w}||_2^2 < \omega$ or $e > e_{max}$ then
7              break
8          if $l(w^*) \leq l(\bar{w}) + m1(f_\beta(w^*) - l(\bar{w}))$ then
9              $\bar{w} = w^*$
10             $\mu = \tau\,\mu$
11         else
12             $\mu = (1 + \tau)\,\mu$
13         $\beta = \beta \bigcup \{(w^*, l(w^*), \nabla l(w^*))\}$
14         $e = e + 1$
```

Algorithm 2 describes in line 8 how the PBM **distinguishes the so called Serious Steps (SS) and Null Steps (NS)** on the base of whether or not $\bar{w} - w^*$ constitues a **nice enough direction of descent**, that is $l(w^*)$ is better than the previous point $l(\bar{w})$ w.r.t. the distance between the solution of MP and the current value of the loss function $l(\bar{w})$. This mechanism makes the series $\{l(\bar{w})\}$ a decreasing series.

In more, the stopping condition in line 6 practically reduces on deciding if the distance $||w^* - \bar{w}||$ is small enough that algorithm can terminate with **a solution that has been provided by a good enough model of the function**, and (if the algorithm is properly set and the loss is not particularly nasty) this **will eventually happen** because

- the series $\{l(\bar{w})\}$ is decreasing,
- the optimal value of $f_\beta(x)$ is always less than the nearest optimum of $l(w)$ because of how the MP is incrementally formulated,
- I can assume that the loss function of a neural network is bounded below.

As for **convergence** of the PBM, global convergence to (global) minimum can be proven only in the case of convex optimization and - more importantly - it can be said that the **algorithm works well on non-differentiable** functions as long as one is able to find an element in the subgradient in case he needs. On non-convex loss functions, I **can count on both resilience to non-differentiability and on the fact that the sequence** $\{f(\bar{w})\}$ **is decreasing, in more because the problem is bounded below, the algorithm will not produce an infinite series of SS** and eventually $||w^* - \bar{w}|| \to 0$, reaching an accumulation point on a (at least local) minimum.

Bibliography

[1] Euhanna Ghadimi, Hamid Reza Feyzmahdavian, and Mikael Johansson. “Global convergence of the heavy-ball method for convex optimization”. In: *2015 European control conference (ECC)*. IEEE. 2015, pp. 310–315.

[2] Peter Ochs. “Local convergence of the heavy-ball method and ipiano for non-convex optimization”. In: *Journal of Optimization Theory and Applications* 177.1 (2018), pp. 153–180.

[3] Boris T Polyak. “Some methods of speeding up the convergence of iteration methods”. In: *Ussr computational mathematics and mathematical physics* 4.5 (1964), pp. 1–17.

[4] Ilya Sutskever et al. “On the importance of initialization and momentum in deep learning”. In: *International conference on machine learning*. PMLR. 2013, pp. 1139–1147.