Computational Mathematics for learning and data analysis

Implementation of a Neural Network optimized through Stochastic Gradient Descent and Conjugate Gradient Descent

Sabrina Briganti - 465214 Gianmarco Ricciarelli - N.a.N

# Contents


**1 The network** **2**
- 1.1 Struttura della rete . . . 2
- 1.2 Forward e back propagation . . . 2
- 1.3 Loss function is differentiable? . . . 2

**2 Optimizers** **3**
- 2.1 Stochastic Gradient Descent . . . 3
  - 2.1.1 Momentum . . . 3
  - 2.1.2 Regolarizzazione . . . 3
- 2.2 Nonlinear Conjugate Gradient . . . 3
  - 2.2.1 Search Direction . . . 3
  - 2.2.2 Beta . . . 4
  - 2.2.3 Line Search . . . 5
  - 2.2.4 Beta . . . 6
  - 2.2.5 Direction . . . 6

**3 Test** **8**
- 3.1 Monk . . . 8
- 3.2 Cup . . . 8

# Chapter 1

# The network

## 1.1 Struttura della rete

## 1.2 Forward e back propagation

## 1.3 Loss function is differentiable?

# Chapter 2

# Optimizers

## 2.1 Stochastic Gradient Descent

### 2.1.1 Momentum

### 2.1.2 Regolarizzazione

## 2.2 Nonlinear Conjugate Gradient

An intresting optimization ables to lead to an improvement of the performances of the Neural Network, is the use of high-order information during the training phase.

We can, indeed, approximate the *loss function* in a quadratic form, around a given point $\mathbf{W}$, using the Taylor approximation:

$$\mathcal{E}(\mathbf{W} + \Delta\mathbf{W}) = \mathcal{E}(\mathbf{W}) + \mathbf{g}\mathcal{E}^T\Delta\mathbf{W} + \frac{1}{2}\Delta\mathbf{W}^T\mathbf{H}\Delta\mathbf{W}, \tag{2.1}$$

where $\mathbf{H}$ is the *Hessian* matrix and $\mathbf{g}$ the gradient vector.

Given Eq. 2.1, the optimum adjustment to apply to the weights of the network should be

$$\Delta\mathbf{W}^* = \mathbf{H}^{-1}\mathbf{g}, \tag{2.2}$$

carrying the weight of the computation of the inverse of the Hessian.

In order to avoid this expensive computation, we can use the *Conjugate Gradient* method, an iterative second-order optimization method, derived from the steepest-descent algorithm, which ensure low memory requirements.

In fact, through this method, the adjustment to the synaptic weights of the network is computed as:

$$\Delta\mathbf{W} = \alpha\mathbf{d}, \tag{2.3}$$

where $\alpha$ is the learning rate and $\mathbf{d}$ is the new direction found.

### 2.2.1 Search Direction

The *Conjugate Gradient* method searches for a direction $\mathbf{d}_k$ that holds the sequent property:

$$\mathbf{d}_k^T\mathbf{H}\mathbf{d}_{tk-1} = 0 \tag{2.4}$$

, that means it is conjugate to the previous direction $\mathbf{d}_{k-1}$. Furthermore, it doesn't need to know all the previous directions, but it only needs the last one, which is why it requires very little storage and computation.

When dealing with quadratic functions, this method keeps the progress obtained so far in the minimization of the loss function, by ensuring that the gradient along the previous

direction does not increase. Anyway, it's worth to underline that this method can also be applyed with nonlinear functions: in this case, it should be necessary to restart the process, since there is no assumption that the conjugate directions previously found are still at the minimum of the function.

Each new direction it's a linear combination of the steepest descent -**g** and the previous direction $\mathbf{d}_{k-1}$, and it is defined as:

$$\mathbf{d}_k = \begin{cases} -\mathbf{g}_0, & \text{if } k = 0; \\ -\mathbf{g}_k + \beta_k \mathbf{d}_{k-1}, & \text{otherwise,} \end{cases} \tag{2.5}$$

where $\beta_k$ is a scalar, to be determined, that says how much of the previous direction should be added to the newest one.

Another modified search direction, that has been proposed by Zang et al., ensures sufficient descent and has been tested in the project:

$$\mathbf{d}_k = -(1 + \beta_k \frac{\mathbf{g}_k^T \mathbf{d}_k}{\|\mathbf{g}_k\|})\mathbf{g}_k + \beta_k \mathbf{d}_{k-1}. \tag{2.6}$$

### 2.2.2 Beta

What really makes the difference in the computation of the conjugate gradient algorithm, is the choice of the method used to compute the $\beta$.

Infact, there have been proposed various choices for computing it, each one giving different efficiency and properties.

The formulas tested in our implementation are three: the Hestenes-Stiefel (HS), the Fletcher-Reeves (FR) and the Polak-Ribierère (PR)

$$\beta_k^{HS} = \frac{\mathbf{g}_k^T(\mathbf{g}_k - \mathbf{g}_{k-1})}{(\mathbf{g}_k - \mathbf{g}_{k-1}^T \mathbf{d}_{k-1})}, \ \beta_k^{PR} = \frac{\mathbf{g}_k^T(\mathbf{g}_k - \mathbf{g}_{k-1})}{\|\mathbf{g}_{k-1}\|^2}, \ \beta_k^{FR} = \frac{\|\mathbf{g}_k\|^2}{\|\mathbf{g}_{k-1}\|^2}. \tag{2.7}$$

One of the properties that must be garanteed, is the global convergence of the method. Since, in our network, we are dealing with a nonquadratic loss function, all the methods have been modified in order to ensure the global convergence:

$$\beta^+ = max\{\beta, 0\}. \tag{2.8}$$

This change provides a sort of restart of the algorithm, in case the $\beta$ found is negative. This is equivalent to forget the last search direction and start again the search from the steepest descent direction.

The HS and the PR methods have very similar performance, and they are some of the most efficient conjugate gradient methods. However, using those method with nonlinear functions, there's no garancy that the $\mathbf{d}_k$ found is always a descent direction: that's why it has been decided to use the modified versions 2.8, which ensures that the descent property holds.

For what concernes the FR method, it requires that $\sigma_1 < \sigma_2 < 0.5$ in order to garantee that the Armijo Wolfe conditions (described in the line search procedure of section 2.2.3) are satisfied, and it seems to be less efficient and robust than the other methods.

### 2.2.3 Line Search

Once computed the new direction **d** involved in the new point $\mathbf{W}+\alpha\mathbf{d}$, a line search has to be implemented in order to find the right step size which minimize the loss function.

The step size $\alpha$ is nothing more than a scalar: the learning rate for the conjugate gradient algorithm, which tells how far is right to move along a given direction.

So, fixed the values of the weights $\mathbf{W}$ and the descent direction $\mathbf{d}$, the main goal is to find the right value for $\alpha$ that is able to minimize the loss function:

$$\min_{\alpha} \quad \mathcal{E}(\mathbf{W} + \alpha \mathbf{d}). \tag{2.9}$$

Of course, we have to deal with a tradeoff: we want a good reduction, but we can't spend too much time computing the exact value for the optimum solution. So, the smarter way to get it is to use an inexact line search, that try some candidate step size and accepts the first one satisfying some conditions.

This search is performed in two phases:

- a *bracketing phase*, that finds an initial interval containing a minimizer;
- an *interpolation phase* that, given the interval, finds the right step length in it.

We decided to use one of the most popular line search condition: the *Armijo-Wolfe* condition.

The search for the better $\alpha$ is led by two condition:

- the *Armijo* one:

  $$\mathcal{E}(W_k + \alpha_k d_k) \leq \mathcal{E}(W_k) + \sigma_1 \alpha \nabla \mathcal{E}_k^T d_k \tag{2.10}$$

  which ensure that $\alpha$ gives a sufficient decrease of the objective function, being this reduction proportional to the step length $\alpha$ and the directional derivative $\nabla \mathcal{E}_k^T d_k$.

  The constant $\sigma_1$ has been set $\sigma_1 = 10^{-4}$, since it is suggested in literature to be quite small.

- the *Strong Wolfe* condition:

  $$|\nabla \mathcal{E}(W_k + \alpha_k d_k)^T d_k| \leq \mathcal{E}(W_k) + \sigma_2 |\nabla \mathcal{E}_k^T d_k| \tag{2.11}$$

  which garantees to choose steps whose size is not too small.

  It is also known as curvature condition and ensures that, moving of a step $\alpha$ along the given direction, the slope of our function if greater than $\sigma_2$ times the original gradient (if the slope is only slightly negative, the function cannot decrease rapidly along that direction, so it's better to stop the search).

  In this case, the constant $\sigma_2$ is equal to 0.1, since a smaller value gives a more accurate line search. Futhermore, having choosen the strong condition, which doesn't allow the derivative to be too positive, we are sure that the $\alpha$ found lies close to a stationary point of the function.

The algorithm satisfing the Strong Wolfe conditions is implemented through three functions, as described in the pseudocodes 1, 2, 3: `line_search`, `zoom` and `interpolate_alpha`.

Since two consecutive values may be similar in finite-precision arithmetic, we set a threshold in both the `line_search` and `interpolate_alpha` functions, which garantees that the algorithm stops if two values of $\alpha$ are too close or if the maximum number of iterations has been reached.

The `line_search` function try to find and return a good $\alpha$; if it fails, it returns an interval in which continue the searching, invoking the `zoom` function, which decreases the size of the interval, until it finds and returns a good step length.

`Zoom` invokes another function, `interpolate_alpha`, which is nothing more than the implementation of a bisection interpolation in order to find a trial $\alpha$ inside the given interval.

### 2.2.4 Beta

### 2.2.5 Direction

**Algorithm 1** Line Search

**procedure** LINE_SEARCH

$\alpha_0 \leftarrow 0$;
$i \leftarrow 1$;
**while** $i \leq max\_iter$ **do**
  Evaluate$\mathcal{E}(\alpha_i)$;
  **if** $[\mathcal{E}(\alpha_i) > \mathcal{E}(0) + \sigma_1 \alpha_i \nabla \mathcal{E}_0^T d_0]$ or $[\mathcal{E}(\alpha_i) \leq \mathcal{E}(\alpha_{i-1})$ and $i > 1]$ **then**
    $\alpha_* \leftarrow$ **zoom**$(\alpha_{i-1}, \alpha_i)$; **return** $\alpha_*$;
  Evaluate $\nabla \mathcal{E}_i$
  **if** $|\nabla \mathcal{E}_i| \leq -\sigma_2 \nabla \mathcal{E}_0^T d_0$ **then**
    $\alpha_* \leftarrow \alpha_i$; **return** $\alpha_*$;
  **if** $\nabla \mathcal{E}_i \geq 0$ **then**
    $\alpha_* \leftarrow$ **zoom**$(\alpha_i, \alpha_{i-1})$; **return** $\alpha_*$;
  **if** $(|\mathcal{E}_i - \mathcal{E}_{i-1}| \leq threshold$ **then**
    $\alpha_* \leftarrow \alpha_i$ **return** $\alpha_*$;
  Choose $\alpha_{i+1} \in (\alpha_i, \alpha_{max})$;
  $i \leftarrow i + 1$;

**Algorithm 2** Zoom

**procedure** ZOOM
  **repeat**
    $\alpha_j \leftarrow interpolate\_alpha(\alpha_{lo}, \alpha_{hi})$;
    Evaluate $\mathcal{E}(\alpha_j)$;
    **if** $[\mathcal{E}(\alpha_j) > \mathcal{E}(0) + \sigma_1 \alpha_j \nabla \mathcal{E}_0^T d_0]$ or $[\mathcal{E}(\alpha_j) \leq \mathcal{E}(\alpha_{lo})$ **then**
      $\alpha_{hi} \leftarrow \alpha_j$;
    **else**
      Evaluate $\nabla \mathcal{E}_j^T d_j$;
      **if** $|\nabla \mathcal{E}_j^T d_j| \leq -\sigma_2 \nabla \mathcal{E}_0^T d_0$ **then**
        $\alpha_* \leftarrow \alpha_j$; **return** $\alpha_*$;
      **if** $\nabla \mathcal{E}_j^T d_j (\alpha_{hi} - \alpha_{lo}) \geq 0$ **then**
        $\alpha_{hi} \leftarrow \alpha_{lo}$;
      **if** $(|\mathcal{E}_j - \mathcal{E}_0| \leq threshold$ **then**
        $\alpha_* \leftarrow \alpha_j$ **return** $\alpha_*$;
    $\alpha_{lo} \leftarrow \alpha_j$;

**Algorithm 3** Interpolate

1: **procedure** INTERPOLATE_ALPHA
2: $i \leftarrow 1$;
3: **while** $i \leq max\_iter$ **do**
4: $\alpha_{mid} \leftarrow (\alpha_{hi} - \alpha_{lo})/2$
5: Evaluate $\mathcal{E}(\alpha_{mid})$;
6: **if** $[\mathcal{E}(\alpha_{mid}) == 0]$ or $[(\alpha_{hi} - \alpha_{lo})/2 < threshold]$ **then return** $\alpha_{mid}$;
7: Evaluate $\mathcal{E}(\_midalpha_{lo})$;
8: **if** $sign(\mathcal{E}(\alpha_{mid})) == sign(\mathcal{E}(\alpha_{lo}))$ **then**
9: $\alpha_{lo} \leftarrow \alpha_{mid}$;
10: **else**
11: $\alpha_{hi} \leftarrow \alpha_{mid}$;
12: $i \leftarrow i + 1$;
=0

# Chapter 3

# Test

## 3.1 Monk

## 3.2 Cup