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

Chapter 1

The network

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 $\mathbf{w}(n)$, using the Taylor approximation:

$$\mathcal{E}(\mathbf{w}(n) + \Delta\mathbf{w}(n)) = \mathcal{E}(\mathbf{w}(n)) + \mathbf{g}(n)\mathcal{E}^T\Delta\mathbf{w}(n) + \frac{1}{2}\Delta\mathbf{w}^T\mathbf{H}(n)\Delta\mathbf{w}(n) \qquad (2.1)$$

where $\mathbf{H}(n)$ 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 $\Delta\mathbf{w}^*(n) = \mathbf{H}^{-1}(n)\mathbf{g}(n)$, carrying the weight of the expensive computation of the inverse of the Hessian.

Here it comes the *Conjugate Gradient* method: it is an iterative second-order optimization method which allows to avoid the computation of the inverse of the Hessian, getting low memory requirements.

It derives from the steepest-descent algorithm, in which each new direction is ortoghonal with respect to the previous direction found. The choice of orthogonality does not preserve the minimum along the previous directions found, but it involves the continous minimization of the *loss function* at each step, that is also the cause of the common zig-zag pattern.

2.2.1 Search Direction

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

$$\mathbf{d}_k^T\mathbf{H}\mathbf{d}_{tk-1} = 0 \qquad (2.2)$$

, 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.

By ensuring that the gradient along the prevous direction does not increase, it keeps the progress obtained so far in the minimization of the loss function.

So, in a k-dimensional space, this method allows to get the minimum in k iterations.

It's worth to underline that this method can be applyed also in case of 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 -$\mathbf{g}$ and the previous direction $\mathbf{d}_{k-1}$, and it is defined as:

$$\mathbf{d}_k = \begin{cases} -\mathbf{g}_0, & \text{if } k = 0; \\ -\mathbf{g}_k + \beta_k \mathbf{d}_{k-1}, & \text{otherwise,} \end{cases} \tag{2.3}$$

where β_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:

$$\mathbf{d}_k = -(1 + \beta_k \frac{\mathbf{g}_k^T \mathbf{d}_k}{\|\mathbf{g}_k\|})\mathbf{g}_k + \beta \mathbf{d}_{k-1}. \tag{2.4}$$

2.2.2 Line Search

In order to find the right step size which minimize the loss function, a line search has to be implemented.

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 α that is able to minimize the loss function:

$$\min_{\alpha} \quad \mathcal{E}(\mathbf{W} + \alpha \mathbf{d}). \tag{2.5}$$

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 α is led by two condition:

- the *Armijo* one:
$$\mathcal{E}(W_k + \alpha_k d_k) \leq \mathcal{E}(W_k) + \sigma_1 \alpha \nabla \mathcal{E}_k^T d_k \tag{2.6}$$

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

  The constant σ_1 has been set $\sigma_1 = 10^{-4}$, since it is suggested in literature to be quite small.

- the *Strong Wolfe* condition:
$$|\nabla \mathcal{E}(W_k + \alpha_k d_k)^T d_k| \leq \mathcal{E}(W_k) + \sigma_2 |\nabla \mathcal{E}_k^T d_k| \tag{2.7}$$

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 α along the given direction, the slope of our function if greater than σ_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 σ_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 α 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 α are too close or if the maximum number of iterations has been reached.

The `line_search` function try to find and return a good α; 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 α inside the given interval.

Algorithm 1 Line Search

1: **procedure** LINE_SEARCH
2:
3:     $\alpha_0 \leftarrow 0$;
4:   $i \leftarrow 1$;
5:   **while** $i \leq max\_iter$ **do**
6:     Evaluate$\mathcal{E}(\alpha_i)$;
7:     **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**
8:       $\alpha_* \leftarrow$ **zoom**(α_{i-1}, α_i); **return** α_*;
9:     Evaluate $\nabla \mathcal{E}_i$
10:    **if** $|\nabla \mathcal{E}_i| \leq -\sigma_2 \nabla \mathcal{E}_0^T d_0$ **then**
11:      $\alpha_* \leftarrow \alpha_i$; **return** α_*;
12:    **if** $\nabla \mathcal{E}_i \geq 0$ **then**
13:      $\alpha_* \leftarrow$ **zoom**(α_i, α_{i-1}); **return** α_*;
14:    **if** $(|\mathcal{E}_i - \mathcal{E}_{i-1}| \leq threshold$ **then**
15:      $\alpha_* \leftarrow \alpha_i$ **return** α_*;
16:    Choose $\alpha_{i+1} \in (\alpha_i, \alpha_{max})$;
17:    $i \leftarrow i + 1$;

2.2.3 Beta

2.2.4 Direction

Algorithm 2 Zoom

1: **procedure** ZOOM
2:     **repeat**
3:         $\alpha_j \leftarrow interpolate\_alpha(\alpha_{lo}, \alpha_{hi})$;
4:         Evaluate $\mathcal{E}(\alpha_j)$;
5:         **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**
6:             $\alpha_{hi} \leftarrow \alpha_j$;
7:         **else**
8:             Evaluate $\nabla \mathcal{E}_j^T d_j$;
9:             **if** $|\nabla \mathcal{E}_j^T d_j| \leq -\sigma_2 \nabla \mathcal{E}_0^T d_0$ **then**
10:                $\alpha_* \leftarrow \alpha_j$; **return** α_*;
11:             **if** $\nabla \mathcal{E}_j^T d_j (\alpha_{hi} - \alpha_{lo}) \geq 0$ **then**
12:                $\alpha_{hi} \leftarrow \alpha_{lo}$;
13:             **if** $(|\mathcal{E}_j - \mathcal{E}_0| \leq threshold$ **then**
14:                $\alpha_* \leftarrow \alpha_j$ **return** α_*;
15:         $\alpha_{lo} \leftarrow \alpha_j$;
16:

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** α_{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