Training Deep and Recurrent Networks with Hessian-Free Optimization

Teoria e Metodi dell'Ottimizzazione

Silvio Martinico

April 14, 2023

Abstract

In this report we analyze two papers by James Martens and Ilya Sutskever [1] [2]. The authors propose a method for training *Recurrent Neural Networks* based on second-order optimization: the *Hessian-Free method* (HF). Managing this kind of methods requires several expedients in order to solve the problems which affect them.

First, we want to have a cheap-to-compute positive semi-definite curvature matrix. For this reason, we rely on the *GGN matrix*, an approximation to the Hessian which is guaranteed to be positive semi-definite. Even with a positive semi-definite matrix, we don't know how much we can trust the second order approximation, so we introduce *Damping Methods*. Finally, since HF exploits *Conjugate Gradient* algorithm, we see how to improve the latter through the initialization and the preconditioning.

Contents

1 Neural Networks

In this introductory section we give some basic Machine Learning notions and notations which we will need later for getting to the heart of the topic.

1.1 Feedforward Neural Networks

An artificial *Neural Network* (NN) is a computing system inspired by the biological neural networks that constitute human's brain. In this system, neurons are modeled by the so called *units*, which are represented by functions, while the synapses, i.e. the connections between neurons, are represented by real numbers called *weights*. We can imagine this model as a directed graph where the nodes are the units and the weights are associated to edges between units. Usually, units are organized in layers, where the first layer, called *input layer*, takes an input and passes it to the successive layers, letting it flow through the network via the connections, until it arrives to the last layer, called *output layer*. The flowing of a vector x from a layer to a unit of the next layer is given by $net(x) := w^T x + b$, where w is the vector of weights from the current layer to the target unit and b is a real number, called bias. Once the *net* arrives to the unit, the latter applies its activation function and the flow continues, considering the obtained value as one component of the new x.

A **Feedforward Neural Network** (FNN) is an acyclic Neural Network where, typically, there are no inter-layer connections, but all the edges go only from one layer to the successive, from the input to the output layer. Having an abstract idea of what this kind of network is, we can formalize these concepts.

Definition 1.1. Feedforward Neural Network
Given an input $x \in \mathbb{R}^n$ and network's parameters θ determining the weights $(W_1, \ldots, W_{\ell-1})$ (where each W_i is a matrix) and biases $b_1, \ldots, b_{\ell-1}$ (where each b_i is a vector), the FNN computes its output y_ℓ inductively by the following recurrence:

$$y_{i+1} = s_i(W_i y_i + b_i) \ ,$$

where $y_1 = x$ and $s_i : \mathbb{R}^{n_i} \longrightarrow \mathbb{R}^{n_{i+1}}$ are (typically non-linear) activation functions.

Given a target t (i.e. the output we would like to approximate for a certain input x), the FNN's training objective function $f(\theta, \langle x, t\rangle)$ is given by:

$$f\big(\theta, \langle x, t\rangle\big) \ = \ L(y_\ell, t) \ ,$$

where L is a *Loss function*. The total training error is given by the average of f over the so called *training set* S, a finite set of pairs (x, t):

$$f(\theta) \ := \ \frac{1}{|S|} \sum_{\langle x,t\rangle \in S} f\big(\theta, \langle x, t\rangle\big) \ .$$

In the pseudocode in Figure 1 we can see how the gradient for a FNN is computed. We need to compute the gradient of the loss function in order to find its minima.

input: y_0; θ mapped to $(W_1, \ldots, W_{\ell-1}, b_1, \ldots, b_{\ell-1})$.
for all i **from** 1 **to** $\ell - 1$ **do**
  $x_{i+1} \leftarrow W_i y_i + b_i$
  $y_{i+1} \leftarrow s_{i+1}(x_{i+1})$
end for
Feedforward (compute output)

$dy_\ell \leftarrow \partial L(y_\ell; t_\ell)/\partial y_\ell$ (t_ℓ is the target)
for all i **from** $\ell - 1$ **downto** 1 **do**
  $dx_{i+1} \leftarrow dy_{i+1} s'_{i+1}(x_{i+1})$
  $dW_i \leftarrow dx_{i+1} y_i^\top$
  $db_i \leftarrow dx_{i+1}$
  $dy_i \leftarrow W_i^\top dx_{i+1}$
end for
Backpropagation (propagate error from output layer to all weights and biases)

output: $\nabla f(\theta)$ as mapped from $(dW_1, \ldots, dW_{\ell-1}, db_1, \ldots, db_{\ell-1})$.

Figure 1: FNN's gradient computation

1.2 Recurrent Neural Networks (RNN)

Recurrent Neural Networks are very powerful models which can represent way more complex systems than FNNs. In fact, a RNN is a Neural Network with feedback connections in its hidden units. These connections allows the net to use past information to compute new predictions. RNNs could model dynamical systems and, complex time-dependent functions, and, in general, they can handle structured inputs (graphs, sequences, etc...).

Definition 1.2. Recurrent Neural Network
A RNN is parametrized by three matrices and an hidden state vector, thus the parameters are $\theta = (W_{xh}, W_{hh}, W_{zh}, h_0)$, where W_{xh} are input-to-hidden connections, W_{hh} are the recurrent connections and W_{zh} are hidden-to-output connections, while h_0 is the initial state vector. Given time-dependent inputs $(x_1, \ldots, x_T) : \forall\ i \in \{1, \ldots, T\}\ x_i \in \mathbb{R}^n$ and target outputs $(t_1, \ldots, t_T)$, the RNN computes a sequence of hidden states h_i and predictions z_i as follows:

$$h_\tau = s(W_{xh} x_\tau + W_{hh} h_{\tau-1})$$
$$z_\tau = W_{zh} h_\tau$$

where s is a (typically non-linear) activation function.

The RNN learning objective function for a single input-target pair $\langle x, t \rangle$ is:

$$f\big(\theta, \langle x, t \rangle\big) = L(z, t) := \sum_{\tau=1}^{T} L_\tau(z_\tau, t_\tau)\ ,$$

where L_τ is a *Loss function*. As for FNNs, the total objective function is given by:

$$f(\theta)\ :=\ \frac{1}{|S|} \sum_{\langle x,t \rangle \in S} f\big(\theta, \langle x, t \rangle\big)\ .$$

Despite the promising capabilities of RNNs, they have seen a small adoption, owing to the difficulties of their training process. A RNN could be transformed into a FNN, called *encoding*

network of the original RNN, with a layer for each time-stamp of the sequence. Unlike FNNs, in the encoding networks the "unrolled" layers share their parameters. So, we can think to train a RNN in the same way we train a FNN. However, the classic approach based on the gradiend is not effective in the case of deep networks (and so, in the case of encoding networks representing long-term dependencies RNNs) due to the *vanishing gradient* problem ([3], [4]): the derivative terms can exponentially decay to zero or blow-up when stacking many layers because of the long-term dependencies which can arise from RNNs. For this reason, many alternative methods have been proposed in literature, such as LSTM, proposed by Hochreiter and Schmidhuber in 1997 and consisting in a RNN augmented with memory units which are responsible for the transmission of long-term information, or Echo-State-Networks, proposed by Jaeger and Haas in 2004, which use fixed randomly-generated sparse connections in the hidden layers. However, the power of these methods seems to be limited, so new approaches based on second-order methods have been proposed.

2 Hessian-Free Methods

Second-order methods, such as Hessian Free, are derived from the classical Newton's Method. What these methods do is basically to iteratively optimize a sequence of local quadratic models of the objective function.

Let's clarify the setting we are working with:

$$f : \mathbb{R}^n \longrightarrow \mathbb{R} \ , \quad f \in C^2 \ .$$

Starting from a point $\theta_0 \in \mathbb{R}^n$, at step $k > 0$, we define a recursive relation for computing a second-order model of f in a neighbourhood of θ_{k-1} (a ball $B(\theta_{k-1}, r)$ with $r > 0$), called M_{k-1}. From this, we can obtain θ_k. Let $\delta \in \mathbb{R}^n$ be such that $\|\delta\| < r$, then the iteration is defined as follows:

$$\begin{cases} M_{k-1}(\delta) = f(\theta_{k-1}) + \nabla f(\theta_{k-1})^T \delta + \frac{1}{2} \delta^T B_{k-1} \delta \\ \theta_k = \theta_{k-1} + \alpha_k \delta_k^* \end{cases}$$

where B_{k-1} is the so-called *curvature matrix*, $\alpha_k \in [0, 1]$ is typically chosen via a line-search and δ_k^* is the point which minimizes M_{k-1}. In the classical Newton's Method B_{k-1} is chosen to be the Hessian matrix of f at θ_{k-1}.

If B_{k-1} was positive definite, M_{k-1} would be bounded below and thus it would have a minimum. In particular, M_{k-1} would be a quadratic function and therefore would have a global minimum at its only stationary point.

Let's find the stationary point of M_{k-1}:

$$0 \ = \ \nabla M_{k-1} \ = \ \nabla f(\theta_{k-1}) + B_{k-1} \delta \quad \Longrightarrow \quad \delta_k^* = -B_{k-1}^{-1} \nabla f(\theta_{k-1}) \ .$$

However, for many good choices of B_{k-1} (such as the Hessian at θ_{k-1}), the computation would be unfeasible for big networks (at least $O(n^2)$ operations for computing the matrix and $O(n^3)$ for solving the linear system $B_{k-1}\delta = -\nabla f(\theta_{k-1})$). Hessian Free methods avoid these computations thanks to the linear *Conjugate Gradient* algorithm.

2.1 Conjugate Gradient method to improve second-order methods

Given the following quadratic objective function:

$$q(x) := \frac{1}{2}x^T Ax - b^T x \ ,$$

where $A \in \mathbb{R}^{n \times n}$ is a positive semi-definite matrix and $b \in \mathbb{R}^n$, Conjugate Gradient algorithm minimizes it by constructing the update from a sequence of vectors which have the property of being A-conjugate.

There is more than one reason behind the choice of this algorithm: it only requires matrix-vectors products (no matrix-matrix products or invertions), it has a fixed-size storage of a few vectors and it is very powerful since it can be seen as a Krylov method and thus it finds in i iterations the optimal solution in the subspace $\mathcal{K}_i(A, r_0) := Span\big(r_0, Ar_0, \ldots, A^{i-1}r_0\big)$, where $r_0 := Ax_0 - b$. Conjugate Gradient method also works very well in practice and usually converges for $i \ll n$. Furthermore, any other gradient-based method applied to a quadratic function like M can be shown to produce solutions which lie in the Krylov subspace and will be strictly outperformed by Conjugate Gradient method in terms of convergence.

Knowing why we have chosen this algorithm, we will use it taking $x = \delta$, $A = B_{k-1}$ and $b = \nabla f(\theta_{k-1})$. The constant term $f(\theta_{k-1})$ can be ignored since it would be killed by the derivative.

2.2 Addressing indefiniteness

In Neural Networks, the objective function f is non-convex and so, the curvature matrix B_{k-1} could sometimes be indefinite, which would imply the non-existence of the minimizer of M_{k-1}.

In order to address this problem, two kinds of solution were proposed. The first one is based on the so-called *damping methods*, which allows to restrict the optimization to *trust regions*, directly or by adding penalty terms. The second one makes use of the *Generalized Gauss-Newton matrix* (GGN) as curvature matrix, which is an approximation to the Hessian guaranteed to be positive semi-definite and it works much better than the Hessian in practice.

We are going to explore this second method since it is the best solution to the indefiniteness problem based on the authors' experience.

2.2.1 Generalized Gauss-Newton Matrix

We will denote the GGN matrix as G.

We need the objective function f to be expressed as the composition of a convex function L and a function F, so that $f(\theta) = L(F(\theta))$. In our case F is the function which maps the parameters θ into an m-dimensional vector of the neural networks outputs $z = F(\theta)$, while L is a convex loss function. Note that, in the case of RNNs, z will represent the outputs from all the time-stamps.

Martens and Sutskever [2] take the GGN matrix as the Hessian of a local convex approximation $\hat{f}$ of f at θ_{k-1} where $F(\theta)$ is replaced with its first order approximation:

$$\hat{F}(\theta) = F(\theta_{k-1}) + JF(\theta_{k-1})\underbrace{(\theta - \theta_{k-1})}_{\delta} \implies \hat{f}(\delta) = L(\hat{F}(\theta)) = L\big(F(\theta_{k-1}) + J\delta\big) ,$$

where we denote the Jacobian matrix of F at θ_{k-1} with just J for simplicity.
Computing the derivatives of $\hat{f}$, we obtain:

$$\nabla\hat{f}(\delta) = J^T \cdot \nabla L(\hat{F}(\theta)) \implies H\hat{f}(\delta) = J^T HL(\hat{F}(\theta))J =: G(\theta) .$$

2.2.2 Choosing L and F

The two functions L and F could be chosen in different ways which can give rise to slightly different GGN matrices. For Neural Networks a natural choice of the output vector z would be the output of the final layer (y_ℓ). However, this may not result in a convex L. One general rule could be that of choosing L in a way that it performs as much of the computation of f as possible, remaining anyway convex. This is obviously a non-mathematical definition and the choice has to be adapted to particular cases.
For instance, in the case of NN with a *softmax* output layer and *cross-entropy* error, it is better to define L to compute both the loss and the activation function, as showed by Schraudolph in [5].

2.2.3 Multiplying by the Gauss-Newton Matrix

As we already said, computing the GGN matrix and exploiting it to compute the updates for our method are really expensive operations. In reality, since we rely on Conjugate Gradient, it is sufficient to find an efficient algorithm for computing the products Gv instead of computing and inverting whole matrices.
Let's recall a useful fact:

Lemma 2.1. Let $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ be differentiable in $x_0 \in \mathbb{R}^n$. Then, it holds:

$$\frac{\partial f(x_0)}{\partial v} = Jf(x_0)v .$$

So, the product between the GGN matrix and a vector v can be written as:

$$Gv = J^T \cdot HL \cdot Jv \overset{2.1}{=} J^T \cdot HL \cdot \frac{\partial F}{\partial v} .$$

The directional derivative can be computed efficiently and in a numerical-stable way with *forward differentiation*, a method which makes repeated use of the chain rule like in the backpropagation algorithm and exploits computational graphs. After this, we have to multiply by the Hessian matrix of L. This, in general, could be expensive; however, the loss function L is usually simple and the multiplication by its Hessian is computationally cheap. For instance, the Hessian of the squared error loss with identity output activation function (w.r.t. the output of the network z) is the identity matrix, while the Hessian matrix of Cross-entropy error with sigmoid output activation function is a simple diagonal matrix. Finally, we have to multiply the resulting vector by J^T. For doing so, we can exploit backpropagation algorithm.

2.2.4 Non-convex loss functions

In order to apply Conjugate Gradient, we have assumed that L is convex (L convex $\implies$ HL positive semidefinite), which, in NN, is generally true. If this were not the case, the matrix $J^T \cdot HL \cdot J$ would not be positive semi-definite (in general, $B^T AB$ is positive semi-definite $\iff$ A is positive semi-definite). In case of non-convexity, there are many possible, often case-dependent, solutions. For instance, we can make F do a part of the computation of L, e.g. if $L(y,t) = \|\tanh y - t\|^2$, we can include the tanh function inside F so that L becomes convex. Another possible solution would be that of approximating the Hessian of L with a positive semi-definite matrix.

3 Damping

As we said in the previous section, basic Newton's method performs poorly when applied directly to highly non-linear objective functions such as those which arise in the training of NNs. This is a consequence of the local nature of the Newton's method: the neighborhood where the approximation is reliable could be small, especially in the earlier phases of the training where we are far from a minimum and thus, the minimizer δ^* of the quadratic approximation M may be located outside the trusted region.

For neural networks (especially for the deep ones) the necessity of the proximity to minimum assumption becomes clear with basic experiments where it can be observed how the naive second order optimization methods tend to diverge.

One can then think of combining a more stable and reliable method such as gradient-descent for the initial phase and second-order methods for a successive "fine-convergence" phase. However, in practice, fine-convergence phase in not necessary and may also be dangerous, leading the optimizer to overfitting. Furthermore, we would like to exploit curvature information in the earlier phase. For this reason, we now introduce *Damping methods*.

Damping methods refer to methods which modify the second-order model M or constraint its optimization to ensure that the update will lie in a region where M can be trusted. The further difficulty in managing the damping methods is that of not excessively limiting the updates, otherwise, we risk to slow down the convergence too much.

Let's explore some damping methods to understand their strengths and weaknesses, so that we can later introduce a damping method more suitable for training neural networks.

3.1 Tikhonov Damping

Tikhonov Damping is one of the most common and simplest damping methods, which operates -like the well-known Tikhonov regularization- by adding to the objective function the squared norm of the variable with respect to which one wishes to minimize. So, in practice, Tikhonov Damping aims to minimize the following function:

$$\hat{M}(\delta) \;:=\; M(\delta) + \frac{\lambda}{2}\delta^T\delta \;,$$

where $\lambda > 0$ is a parameter which determines the strength of the damping. Recalling that $M(\delta) = f(\theta) + \nabla f(\theta)^T \delta + \frac{1}{2}\delta^T B \delta$ (omitting indices) and defining $\hat{B} := B + \lambda I$, we can rewrite $\hat{M}$ as:

$$\hat{M}(\delta) \;=\; f(\theta) + \nabla f(\theta)^T \delta + \frac{1}{2}\delta^T \hat{B} \delta \;.$$

Tikhonov Damping does not add much computation to the method since computing $\hat{B}v$ for a vector v requires just to multiply v by λ and to add this to the result of Bv.

The incentive to keep δ small is clear. However, we want to observe a more subtle effect of the Tikhonov Damping. Let $\{\lambda_j : j = 1, 2, \ldots, n\}$ be the spectrum of B and $B = V \Sigma V^T$ be a spectral decomposition (remember that B is symmetric and so the columns v_i of V form an orthonormal basis of $\mathbb{R}^n$); then, $\hat{B} = V(\Sigma + \lambda I)V^T$ is a spectral decomposition of $\hat{B}$ and $\hat{B}^{-1} = V^T(\Sigma + \lambda I)^{-1} V$. Let's compute the minimizer δ^* of $\hat{M}$:

$$\frac{\partial \hat{M}}{\partial \delta} \;=\; \frac{\partial M}{\partial \delta} + \lambda\delta \;=\; \nabla f + (B + \lambda I)\delta \;=\; 0 \quad \Longleftrightarrow \quad \delta = -(B + \lambda I)^{-1} \nabla f(\theta_{k-1})$$

and so, rewriting $\hat{B}$ as its spectral decomposition, we have:

$$\delta^* \;=\; -V^T(\Sigma + \lambda I)^{-1} V \nabla f(\theta_{k-1}) \;=\; \sum_{j=1}^{n} \frac{v_j^T \nabla f(\theta_{k-1})}{\lambda_j + \lambda} v_j \quad , \tag{1}$$

where $v_j s$ are the rows of V and form an orthonormal basis of $\mathbb{R}^n$.

From (1) it is clear that the effect of Tikhonov Damping is more noticeable in correspondence of the eigenvalues of B which are small compared to λ. In particular, small eigenvalues λ_j tends to blow-up the component in v_j of δ^* and Tikhonov Damping helps to keep these components smaller. Thus, Tikhonov Damping should be appropriate when M is most untrustworthy in correspondence of directions with very low curvature, along which δ^* will tend to travel very far in absence of damping.

For this reason, choosing a good value of λ in Tikhonov Damping is crucial, but it is not trivial. Furthermore, λ is sensitive to the overall scale of f and it varies over the parameter space. These factors make Tikhonov Damping hard to use in practice with complex objective functions.

Reasoning about Tikhonov Damping, it is clear that a good approach should weigh different directions in different ways, assigning more weight to directions associated with more serious violations of the the approximation quality of M, without slowing down the convergence too much.

3.2 Scale-Sensitive Damping

As we said in the previous section, Tikhonov Damping is scale-sensitive, which means that for $a \in \mathbb{R}$, taking $a\lambda$ as Tikhonov Damping's parameter for af would produce the same update of λ for f. Scale-sensitivity of Tikhonov Damping is similar to that which affects first-order methods and is precisely the kind of problem we would like to avoid with second-order methods. In fact, gradient methods can be viewed as a special case of second-order methods with zero curvature and Tikhonov Damping, i.e. $\hat{B} = \lambda I$ and $\delta^* = -\frac{1}{\lambda}\nabla f(\theta)$.

One can think of using only parametrizations which exhibit approximately uniform sensitivity properties in order to solve the scale-sensitivity problem. However, this is clearly limiting.

A possible solution to the problem is to use quadratic penalty function which depends on θ_{k-1} and respects the local scale properties of $f(\theta_{k-1})$. The idea is to use the following penalty term:

$$\frac{\lambda}{2}\|\delta\|^2_{D_{k-1}} := \frac{\lambda}{2}\delta^T D_{k-1}\delta \quad ,$$

where D_{k-1} is called *damping matrix* and it is a symmetric, positive definite matrix that depends on θ_{k-1}.

Possible choices for D are to take it as the diagonal matrix which diagonal is equal to that of the curvature matrix B or to take $D = B$.

Despite the good scale-invariance properties associated with this kind of damping matrix, they are not used in practice, if not under certain modifications and in combo with other approaches. In fact, while Tikhonov Damping makes too few assumptions about the local behaviour of f, damping approaches based on $D = B$ or its diagonal may make too many as the original undamped quadratic approximation M. For instance, if B is not injective, M will be unbounded along directions in $\ker B$ and this problem is not solved by damping with $D = B$, whatever λ we take. Also if B is full-rank, there may be directions of near-zero curvature which can cause a similar situation to that of a non-injective B. Taking D as the diagonal matrix with the same diagonal as B would solve these problems because it's more likely full-rank or is at least better-conditioned, but it's anyway far from an ideal solution.

Let's try to understand why these choices of D fail: the quadratic approximation could be unreliable due to higher-order effects or even due to some second-order effects which are not well-modeled by GGN matrix. By taking $D = B$, we are penalizing directions according to their curvature and so we are assuming that the relative strength of the contribution to f from the higher-order terms along two different directions can be predicted reasonably well by looking at the ratio of their respective curvature, which could be a too strong assumption.

Without other information on the local behavior of f, it is hard (if not impossible), to make further assumption. However, the objective functions we want to optimize are not completely random functions: we need an approach able to exploit the particular structure of deep and recurrent neural networks in order to build suitable damping matrices.

3.3 Structural Damping

As we said erlier, Tikhonov Damping does not perform well when applied to RNNs, as Martens and Sutskever observed in [2]. In order to avoid large and untrustworthy updates, Tikhonov Damping needs very high λ values, which make the optimization much slower. RNNs, having often a long sequence of hidden states, are very sensible to parameters change (it is as if the sensitivity grows exponentially with the number of layers). This high and non-linear sensitivity makes local quadratic approximations inaccurate in some directions, even at small distances; Tikhonov Damping tries to solve this problem by imposing a static penalty term, equal in every direction, so it is a very rigid approach.

Structural damping imposes a quadratic penalty not only to change in parameters, but also to some intermediate values calculated by the RNN, in order to have a more selective way

to penalize direction. There is also another more subtle benefit of structural damping in RNN's training. In some works ([2], [6]), it was observed that good random initializations give rise to nontrivial hidden state dynamics that can carry useful information about the past inputs even before the learning phase. In this case, structural damping may encourage the updates to preserve these dynamics at least until the hidden-to-output weights have had enough time to be adapted to the point where the long-range information contained in the hidden activities actually gets used to inform future predictions.

For structural damping, we need an objective function in the form $f(\theta) = L\big(z\big(h(\theta), \theta\big)\big)$, where L is the loss function, $h(\theta)$ is the function which computes the output of the hidden layer, while $z(h, \theta)$ computes the output of the RNN. As we anticipated, we want to penalize an intermediate quantity; in particular, we penalize the following function:

$$S(\delta) := d\big(h(\theta_{k-1} + \delta), h(\theta_{k-1})\big) \ ,$$

where d is a distance function. This means that we are going to penalize excessive changes in RNN's hidden state.

The function that structural damping ideally aims to minimize is:

$$\hat{M}^{\dagger}(\delta) = M(\delta) + \mu S(\delta) + \lambda I \ ,$$

where both μ and λ are strength constants. However, Conjugate Gradient minimizes quadratic functions. Thus, instead of S, we use its local quadratic approximation:

$$\frac{1}{2}\delta^T D_{k-1}\delta \ ,$$

where D_{k-1} is the Gauss-Newton matrix of S at $\delta = 0$. Note that the quadratic approximation of S does not have a linear term because $\delta = 0$ is a minimum of S.

Multiplying by the GGN matrix of S is computationally cheap thanks to the methods we analyzed in Section 2.2.3, so adding this term is not a problem. Thus, the total computation time is approximately doubled since we have to compute both Bv and $\mu D_{k-1}v$ by Gauss-Newton matrix-vector products and then to sum them.

The next step is to manage the two damping's strength parameters λ and μ.

3.3.1 The Levenberg-Marquardt Heuristic

For the structural damping to work well, we need to constantly adapt the two parameters λ and μ. Let's focus on λ for simplicity. For this purpose, we will exploit the famous *Levenberg-Marquardt heuristic*.

The LM heuristic is based on a fundamental quantity, the so-called *reduction ratio*, defined as:

$$\rho \ := \ \frac{f(\theta_{k-1} + \delta_k) - f(\theta_{k-1})}{M_{k-1}(\delta_k)} \ .$$

It measures the ratio between the real change produced by the update δ_k in the objective function and the amount of change of the quadratic model. The relevance of the reduction ratio is due to the following observation:

- $\rho \ll 1 \implies$ the quadratic model overestimates the reduction, so we need to be more conservative, i.e. we need to increase λ;
- ρ close to $1 \implies$ the quadratic approximation is accurate enough, so we can relax the constraints on the update, i.e. we can reduce λ.

The Levenberg-Marquardt heuristic sets the values for the above conditions in the following way:

- $\rho > \frac{3}{4} \implies \lambda \leftarrow \frac{2}{3}\lambda$
- $\rho < \frac{1}{4} \implies \lambda \leftarrow \frac{3}{2}\lambda$

Although there is no formal mathematical motivation behind the choice of these constants, they showed to work well in practice. Furthermore, we can also use $\hat{M}$ instead of M in the denominator of the definition of ρ, which gave similar result in practice, slightly favoring smaller λ values.

If Conjugate Gradient converges, LM heuristic would be very effective and gives provable strong local convergence guarantees. If the updates are produced by unconverged runs of CG, LM heuristic could be negatively affected since ρ is a function of how much progress CG tends to make when optimizing M. However, as long as the local properties of f change slowly enough, an early stopping of CG should result is a relatively stable and well-chosen value of λ. This approach hides a dangerous behaviour: the early stop of CG will affect ρ, which will affect the choice of λ, which will affect the damping and so the value of f as CG iterates, but the latter will affect the decision of when to stop CG, taking the algorithm to a circular behaviour. For this reason, we want to explore another class of methods which can help us in the update.

3.4 Trust-Region Methods

Instead of penalizing the objective function in finding a minimizer far from the current point θ_{k-1}, we can think of forcing updates to be small directly by limiting the set where we search them. This would transform the problem from quadratic unconstrained optimization on $\hat{M}$ to a constrained one on M. The quantity we look for becomes:

$$\delta_R^* := \arg\min_{\delta \in R} M(\delta) \ ,$$

where $R \subset \mathbb{R}^n$ is a neighbourhood of $\delta = 0$ called *Trust Region*. The generic property we want to hold inside R is that M should remains a reasonable approximation to f for every $\delta \in R$, while not being over-restrictive.

There is a connection between trust region methods and penalty-based damping methods such as Tikhonov Damping, given by the following more general theorem:

Theorem 3.1. Let $r \in \mathbb{R}^+$ and $Q \in \mathbb{R}^{n \times n}$ a positive definite matrix. Let $R = \{x \in \mathbb{R}^n : \|x\|_Q \leq r\}$ (i.e. R is a Q-elliptical ball of radius r). Given a quadratic function $M : \mathbb{R}^n \longrightarrow \mathbb{R}$, with $M(x) = \frac{1}{2}x^T Bx + g^T x + c$ and $B \in \mathbb{R}^{n \times n}$ is positive semi-definite, then:

$$\exists\ \lambda \in \mathbb{R}\ :\ \arg\min_{\delta \in R} M(\delta)\ =\ \arg\min_{\delta \in \mathbb{R}^n} M(\delta) + \frac{\lambda}{2}\|\delta\|_Q^2$$

Proof:
Consider the Lagrangian function associated with the minimization problem $\min_{\delta \in R} M(\delta)$:

$$L(\delta, \lambda) = M(\delta) + \lambda \left(\frac{1}{2}\delta^T Q\delta - r\right)\ .$$

Let δ^* be the minimizer of the constrained optimization problem. From KKT conditions, it follows that it exists λ^* such that:

$$\frac{\partial L}{\partial \delta}(\delta^*, \lambda^*) = 0 \tag{2}$$

M is a quadratic function, w.l.o.g. $M(\delta) = \frac{1}{2}\delta^T B\delta + g^T\delta$ because the constant term is irrelevant to the arg min function. So, from (2), it follows:

$$B\delta^* + g + \lambda^* Q\delta^* = 0 \implies (B + \lambda^* Q)\delta^* + g = 0\ .$$

Now, the matrix $B + \lambda^* Q$ is clearly positive definite, since it is the sum between the positive definite matrix $\lambda^* Q$ and the positive semi-definite matrix B. Thus, we can conclude by observing that δ^* can be expressed as the solution of the unconstrained optimization problem:

$$\min_{\delta \in \mathbb{R}^n} \left(\frac{1}{2}\delta^T\big(B + \lambda^* Q\big)\delta + g^T\delta\right)\ ,$$

which can be rephrased as:

$$\min_{\delta \in \mathbb{R}^n} M(\delta) + \frac{1}{2}\lambda^*\|\delta\|_Q^2\ .$$

□

Theorem 3.1 holds in our setting since the matrix of our quadratic model is the GGN matrix, which is positive semi-definite.

Trust region methods may have same advantages over penalty-based damping methods in which it can be easier to reason about the effect of an explicit trust region with a given radius r and, primarily, the trust region is invariant to changes in the scale of the objective function. However, trust region methods have the disadvantage that constrained optimization is way harder than unconstrained one. We can think of transforming the trust region problem into a corresponding Tikhonov damping problem, but finding the λ of the 3.1 (which is a non-constructive theorem) is not easy and algorithms which do that often involve expensive operations.

If we had a way to effectively solve (or partially optimize) the constrained problem, we have still the problem of adjusting r. The good thing is that r is a more stable parameter than λ and we already know a good method to adjust it: the Levenberg-Marquadt heuristic we saw for structural damping could be exploited also for r.

An effective method to find an approximate solution of the trust-region subproblem when R is a ball of radius r is the Steihaug-Toint method, which consists in running CG initialized at zero and truncating it as soon as the solution exceeds r, with a possible modification of the last step so that the final solution has norm exactly r. In [7] it is showed that in this way the obtained solution $\delta_k^\dagger$ will satisfy:

$$M(\delta_k^\dagger) \leq \frac{1}{2} M(\delta_k^*) \tag{3}$$

where δ_k^* is the optimal solution inside the trust region (we are assuming the optimal value is < 0, otherwise the constant becomes a 2). This result gives us a guarantee on the effectiveness of this method, which is much easier than solving exactly the trust-region problem.
However, this method have some cons. Firstly, the restriction that CG must be initialized from zero cannot be removed easily. Secondly, it can happen that the trust region is leaved after few iterations and in this case it may be that few (or none) of the low-curvature directions have converged sufficiently, which is not a good news even if we have the guarantee of the bound (3). Finally, Steihaug-Toint method cannot be used with preconditioned CG, which we will see has a fundamental role for us. However, we can deceive this problem by enforcing an elliptical trust-region $\{x \in \mathbb{R}^n : \|x\|_P < r\}$ via a preconditioning matrix P and it can be shown that this method remains valid.

3.5 Conjugate Gradient Truncation

We would like to truncate CG before convergence because, as iterations advance, the return in terms of optimization would be lower and lower, while the computational cost would increase. However, truncation could have a second role as damping method itself.

It was observed that the improvement in the value of the objective function f could be non-monotonic in the number of CG steps, as showed in Figure 2, and it may peak before we reach the threshold we fix on the reduction. It is important to emphasize that this behaviour is the consequence of the non-exact machine arithmetic. In fact Conjugate Gradient, being a Krylov method, is guaranteed to converge and to improve at each iteration.
Ideally, we would like to stop CG when the reduction is minimized, so that we both minimize f as much as we can with the Conjugate Gradient and stop the CG avoiding useless iterations. In order to do this, Martens in [6] proposed the *CG-backtracking*, a method for selecting the "best" iterate among some manageable subset.

The damping effect of CG truncation is not suddenly understandable as that of penalty-based damping methods and it depends both on preconditioning scheme and CG initialization. To understand the effect of the preconditioning, we have to look at it in the following way: preconditioning reparametrizes the quadratic optimization problem and so it creates new eigenvectors and eigenvalues, changing the Krylov subspace. This affects the order in which CG tends to prioritize convergence along different directions and so preconditioning heavily

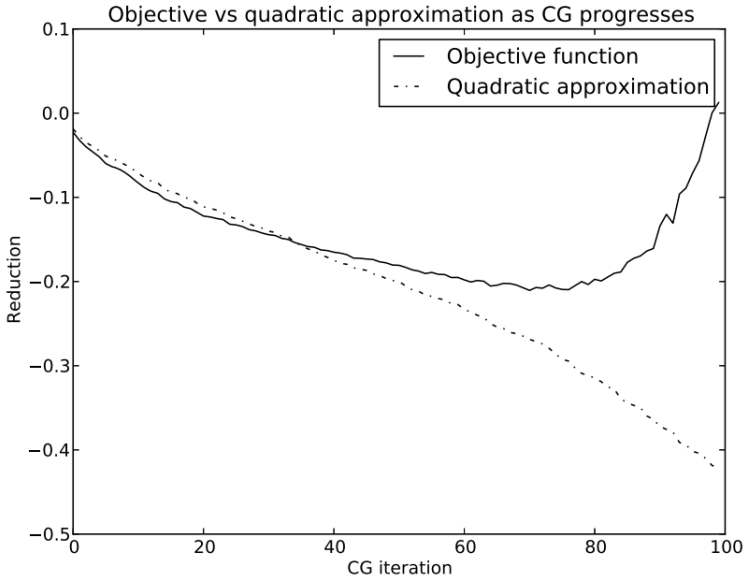

Figure 2: Training of a deep auto-encoder

impacts on the damping effect of the early termination of CG. Looking at preconditioning from a trust-region/Steihaug-Toint method point of view, it determines the shape of the trust-region (enforced by truncation) so that the trust region will be $R = \{x : \|x\|_P < r\}$, where P is the preconditioning matrix. In a similar way, we can think to the initialization of CG from non-zero points as shifting the center of the trust-region away from 0. In both cases (preconditioning and non-zero initialization) the bound (3) holds.

3.6 Line searching

One of the most common and used form of damping is *Line Search*, which consists in multiplying the update δ_k by a "step size", i.e. a real number α_k, so that we obtain $\theta_k = \theta_{k-1} + \alpha_k \delta_k$. This is basically the idea used in first order methods.

Line search in second order methods is not used as a main damping method which can substitute other methods, but rather as a "recovery method" used for adjusting temporary inaccuracies caused by other damping methods. This is a consequence of the aggressive nature of the line-search, which works by equally re-scaling each conjugate direction of the update, that is a limited and not very flexible approach. If we abuse it, the convergence could be slowed-down consistently. If line search very often chooses $\alpha \ll 1$, then there is a problem with the main method, which can be caused both by an inappropriate damping method and/or heuristic or by insufficient data used for the update (small batch).

Provided that δ_k is a descent direction, for α_k sufficiently small we can for sure obtain a reduction on the objective function. One method for finding a good α (which is sufficiently small but not excessive small, so that it does not slow-down the convergence) is the back-tracking approach, which consists of starting with $\alpha = 1$ and multiply it for a constant

$\beta \in (0, 1)$ until the following condition applies:

$$f(\theta_{k-1} + \alpha\delta_k) \leq f(\theta_{k-1}) + c\alpha\nabla f(\theta_{k-1})^T\delta_k \ ,$$

where c is a small constant (usually $\approx 10^{-2}$). Note that, if δ_k is a descent direction, by definition we have: $\nabla f(\theta_{k-1})^T\delta_k < 0$. It can be shown that this approach will produce updates with strong convergence properties.

Note that approximated second-order methods like HF tends to converge way faster than first-order methods, so that the expense of line-search has less impact.

4 Managing Conjugate Gradient

We talked about preconditioning and initialization with refer to the Conjugate Gradient method. Let's analyze this two aspects in details, given that the function CG is optimizing is $q(x) = \frac{1}{2}x^TAx - b^Tx$.

4.1 Initializing Conjugate Gradient

Conjugate Gradient method works for whatever initial vector x_0 we take. However, the choice of the starting point could have a huge impact on the performance of the algorithm. The importance of CG initialization is evident if we think about it as a Krylov method: the Krylov subspace at the kth iteration of CG is $\mathcal{K}_k(A, r_0)$, where $r_0 = Ax_0 - b$, so it is strictly dependent on x_0.

What empirically emerges is that CG initialized with $x_0 = \delta_{k-1}$ "starts slow" but then eventually catch up and surpass CG initialized from 0. An hypothesis for this behaviour is that $x_0 = \delta_{k-1}$ may be more converged along low-curvature direction, which are that directions harder to optimize.

The non-zero initialization approach resembles the *momentum* used in gradient descent optimization. It is helpful to decay the initialization taking $x_0 = \zeta\delta_{k-1}$, where $\zeta \in [0, 1]$ is typically near 1 (e.g. 0.95), i.e. we use a slight decay. Differently from momentum, HF is less sensible to variation of ζ because momentum modifies the current update by a single gradient descent step, while HF uses an entire run of CG, which is more robust. A proper tuning of the decay constant becomes more important under aggressive truncation of CG or weak preconditioning. In the first case it is better to take a smaller ζ value, because early truncated CG produce a less converged (and so less trustworthy) δ_{k-1}, while in the second case it can help to increase the value of ζ, especially in later stages of optimization where CG struggles to optimize q along low curvature directions.

4.2 Preconditioning

Preconditioning, as we already anticipated, has a huge role in HF methods. A good preconditioner could help CG to exploit the structure of the curvature matrix. In fact, nonetheless CG is one of the best algorithm in his field, it is not able to exploit the specific structure of the matrix. For instance, if the curvature matrix is a diagonal invertible matrix, CG would

converge in i iterations, where i is the number of different values on the diagonal, taking thus $O(in)$ time; on the other side, the system could easy be solved by inverting the matrix in $O(n)$ time.

Preconditioning consists in a reparametrization of $\hat{M}$: given an invertible matrix C, we transform $\hat{M}(\delta)$ by a change of coordinates $\delta = C^{-1}\gamma$ and we optimize it w.r.t. the new variable γ. The new reparametrized objective function is:

$$\hat{M}(C^{-1}\gamma) \;=\; \frac{1}{2}\gamma^T C^{-T}\hat{B}C^{-1}\gamma + \nabla f(\theta)^T C^{-1}\gamma + f(\theta)\ .$$

In Figure 3 we can see the preconditioned conjugate gradient algorithm. The additional computations are given by solving $Py_i = r_i$ at each iteration, where $P := C^T C$. In many practical cases (e.g. when P is a diagonal approximation of $\hat{B}$) this is a cheap operation.

inputs: b, A, x_0, P
$r_0 \leftarrow Ax_0 - b$
$y_0 \leftarrow$ solution of $Py = r_0$
$p_0 \leftarrow -y_0$
$i \leftarrow 0$
while termination conditions do not apply **do**
  $\alpha_i \leftarrow \dfrac{r_i^\top y_i}{p_i^\top A p_i}$
  $x_{i+1} \leftarrow x_i + \alpha_i p_i$
  $r_{i+1} \leftarrow r_i + \alpha_i A p_i$
  $y_{i+1} \leftarrow$ solution of $Py = r_{i+1}$
  $\beta_{i+1} \leftarrow \dfrac{r_{i+1}^\top y_{i+1}}{r_i^\top y_i}$
  $p_{i+1} \leftarrow -y_{i+1} + \beta_{i+1} p_i$
  $i \leftarrow i + 1$
end while
output: x_i

Figure 3: Preconditioned Conjugate Gradient algorithm

CG prioritizes convergence along the eigen-directions both according to their weights ω_j (total decrease in q which can be obtained in that direction by completely optimizing it) and their associate curvature (eigenvalue), preferring in both cases greater values (we want to prioritize directions which offer more optimization possibilities and with higher curvatures, which are that where the second-order approximation is more reliable). So, in order to understand how the preconditioning influences CG's convergence, we can look at the eigenvalues and eigenvectors of the transformed curvature matrix $C^{-T}\hat{B}C^{-1}$ and at its weights, which depend on the transformed initial residual $C^{-T}(\hat{B}x_0 - \nabla f)$. In particular, it would be useful if preconditioning creates tight clusters of eigenvalues, which will lead to a faster convergence.

When using preconditioning, we must be careful to not transform directions which would be untouched by CG in the original parametrization because of their low curvature or weights into direction which will be optimized by preconditioned CG. For this reason, designing a good preconditioner is relevant and it is not a trivial task.

When we design a preconditioner, we usually look for a trade-off between computational expense and effectiveness. A wide used and effective preconditioner in CG optimization is

to take P as an easy-to-invert approximation of the curvature matrix $\hat{B}$. For this reason, diagonal preconditioners are a convenient choice (full rank, easy to invert, cheap to store). However, as reported by Martens and Sutskever in [2], RNNs don't benefit enough from classical diagonal preconditioners such as the diagonal matrix d whose diagonal is equal to that of the curvature matrix $\hat{B}$. This choice proves to be ineffective when $\hat{B}$ has a strong non-diagonal component.

Martens found in [6] that it can be beneficial to use, instead of d, a diagonal preconditioner between d and a multiple of the identity matrix:

$$P = \kappa d^{\xi} \ ,$$

where $k \in \mathbb{R}^+$ and $0 < \xi < 1$. In this way, $\xi \to 0 \implies \kappa d^{\xi} \to \kappa I$. This is a more conservative and gentle choice and works considerably better in practice. In cases where we don't have diagonal damping penalty terms like Tikhonov Damping (or when they are very weak), we can modify it in the following way:

$$P = (d + \kappa I)^{\xi} \ .$$

Even if we can avoid to compute the full curvature matrix and rely on the technique reported in the Section 2.2.3 to compute Hessian-vector products, there exists no efficient algorithm for computing the diagonal of the Hessian of a general nonlinear function, so we must rely on an approximation. In particular, we can think of using the GGN matrix to approximate the Hessian and then approximate its diagonal.

One approach to approximate the diagonal of the curvature matrix is the *Empirical Fisher Diagonal*, given by:

$$\text{diag}(F) := \sum_i \text{sq}(\nabla f_i) \ ,$$

where the sum is over the minibatch examples and $sq(\cdot)$ is the element-wise square. This definition comes from the *Empirical Fisher Information matrix* F, defined as:

$$F := \sum_i \nabla f_i \nabla f_i^T \ ,$$

which is an approximation to the *Fisher Information matrix*, related to the GGN matrix. The computation of $diag(F)$ is not expensive since the ∇f_is are available from the gradient computation over the batch:

$$\nabla f = \sum_i \nabla f_i \ ,$$

and from there, we can easily compute it in parallel without extra costs.

Another possible approach to compute an approximation of the diagonal of the GGN matrix is that of using an unbiased estimator for it. Chapelle and Erhan in [8] gave a randomized algorithm for computing this unbiased estimator with the same computational cost of the gradient computation. The algorithm is very simple and it can be resumed in the following main steps:

Algorithm 4.1.

i. Sample $v \in \mathbb{R}^m$ from a distribution such that $\mathbb{E}[vv^T] = I$

ii. output $sq\big(J^T(L''^{\frac{1}{2}})^T v\big)$

Theorem 4.1. Algorithm 4.1 is correct.

Proof:
It is sufficient to prove that the expected value of the output is $diag(G)$, where G is the GGN matrix.

$$\begin{aligned}
\mathbb{E}\left[\mathrm{sq}\left(J^T(L''^{\frac{1}{2}})^T v\right)\right] &= \mathbb{E}\left[\mathrm{diag}\left(\left(J^T(L''^{\frac{1}{2}})^T v\right)\left(J^T(L''^{\frac{1}{2}})^T v\right)^T\right)\right] \\
&= \mathrm{diag}\left(J^T(L''^{\frac{1}{2}})^T \underbrace{\mathbb{E}(vv^T)}_{I} L''^{\frac{1}{2}} J\right) \\
&= \mathrm{diag}\left(J^T(L''^{\frac{1}{2}})^T L''^{\frac{1}{2}} J\right) \\
&= \mathrm{diag}\big(J^T L'' J\big) \;=\; \mathrm{diag}(G)
\end{aligned}$$

□

It was showed that, in the first step of the algorithm, sampling v from a uniform distribution in $[-1, 1]$ produces better results (lower variance) than sampling from $\mathcal{N}(0, 1)$.

Regarding the costs, as we already stated previously, the multiplication of a vector by the Jacobian J of F can be done efficiently via back-propagation algorithm, while the multiplication by L'' is usually simple due to its structure, and so it is the multiplication by its square root.

Observation 4.1. We are assuming L to be convex, which implies that L'' is positive semidefinite, so its square root is well-defined.

In practice, both Unbiased Estimator and Diagonal Fisher Information seems to produce preconditioners with similar properties and performance characteristic, with the Unbiased Estimator which performed better in certain situations. One advantage of Unbiased Estimator is that it accounts for structural damping, while a disadvantage is that there could be parameters with non-zero gradients whose diagonal estimate could be very small or zero (it clearly does not happen with Diagonal Fisher Information). Finally, Diagonal Fisher Information has the advantage that it actually has no additional cost since it can be computed in parallel while we compute the gradient, paying nothing extra.

4.3 Convergence of Conjugate Gradient

We want to conclude this work formalizing some concepts we introduced along the report, such as weight of a direction or convergence along a specific direction.

The setting in this analysis is the following: we want to optimize $q(x) = \frac{1}{2}x^T Ax - b^T x$ with A symmetric and positive definite, $A \in \mathbb{R}^{n\times n}$ and $b \in \mathbb{R}^n$. In our HF method, as we already did, we would take $A = \hat{B}$ and $b = -\nabla f$.

Let $\{\lambda_j : j = 1, 2, \ldots, n\}$ be the spectrum of A and $\{v_j : j = 1, 2, \ldots, n\}$ an orthonormal basis of $\mathbb{R}^n$ of eigenvectors for A, such that v_j is an eigenvector relative to λ_j. Let x^* be the minimizer of q and x_0 the starting point of CG.

As we anticipated, we want to give a mathematical ground to the concept of CG converging along different directions.

First, we rewrite $q(z + x_0) - q(x_0)$ for a certain $z \in \mathbb{R}^n$:

$$\begin{aligned} q(z+x_0) - q(x_0) &= \frac{1}{2}(z+x_0)^T A(z+x_0) - b^T(z+x_0) - \frac{1}{2}x_0^T Ax_0 + b^T x_0 \\ &= \frac{1}{2}\left(z^T Az + x_0^T Az + z^T Ax_0\right) - b^T z \qquad (4) \\ &= \frac{1}{2}z^T Az + (x_0^T A - b^T)z \;=\; \frac{1}{2}z^T Az + r_0^T z \;. \end{aligned}$$

Now we define $\eta_j := r_0^T v_j$, which represents the size of the eigenvector v_j in direction r_0. For each $\alpha \in \mathbb{R}$ it holds:

$$q(\alpha v_j + x_0) - q(x_0) \overset{(4)}{=} \frac{1}{2}\alpha^2 v_j^T Av_j + \alpha r_0^T v_j \;=\; \frac{1}{2}\alpha^2\lambda_j + \alpha\eta_j \;. \qquad (5)$$

Given two A-conjugate vectors u and w (i.e. $u^T Aw = 0$), it is easy to observe that the following equality holds:

$$q(u+w+x_0) - q(x_0) \overset{(4)}{=} \big(q(u+x_0) - q(x_0)\big) + \big(q(w+x_0) - q(x_0)\big) \;. \qquad (6)$$

Let $x - x_0 = \sum_{j=1}^{n} \beta_j v_j$; note that β_j represents the size of $x - x_0$ in the direction of the eigenvector v_j. Putting everything together, we have:

$$\begin{aligned} q(x) - q(x_0) &= q\left(\sum_{j=1}^{n}\beta_j v_j + x_0\right) - q(x_0) \\ &\overset{(6)}{=} \sum_{j=1}^{n}\big(q\left(\beta_j v_j + x_0\right) - q(x_0)\big) \qquad (7) \\ &\overset{(5)}{=} \sum_{j=1}^{n}\frac{1}{2}\beta_j^2\lambda_j + \beta_j\eta_j \end{aligned}$$

We can consider the equation (7) restricted to the direction v_j:

$$\varphi(\beta_j) := q(\beta_j v_j + x_0) - q(x_0) = \frac{1}{2}\beta_j^2 \lambda_j + \beta_j \eta_j \ .$$

In order to find the β_j which minimizes it, we compute its derivative:

$$\varphi'(\beta_j) = \lambda_j \beta_j + \eta_j = 0 \iff \beta_j^* = -\frac{\eta_j}{\lambda_j}$$

and its minimum value is:

$$\varphi(\beta_j^*) = -\frac{1}{2}\frac{\eta_j^2}{\lambda_j} = -\omega_j \ ,$$

where we define $\omega_j := \frac{\eta_j^2}{\lambda_j}$. We already talked about the "weight" of a direction, and this is now a formal definition of it. As we already said, ω_j indicates the total decrease in q which can be obtained in the direction v_j by completely optimizing it.
We can write the difference between the current value of $q(\beta_j v_j + x_0)$ and its minimum in a nice and useful form:

$$\begin{aligned} q(\beta_j v_j + x_0) - q(\beta_j^* v_j + x_0) &= \big(q(\beta_j v_j + x_0) - q(x_0)\big) - \underbrace{\big(q(\beta_j^* v_j + x_0) - q(x_0)\big)}_{-\omega_j} \\ &\overset{(7)}{=} \frac{1}{2}\beta_j^2 \lambda_j + \beta_j \eta_j + \omega_j \\ &= \omega_j \left(\frac{1}{2}\frac{\beta_j^2 \lambda_j^2}{\eta_j^2} + \frac{\beta_j \lambda_j}{\eta_j} + 1\right) \ . \end{aligned} \tag{8}$$

Now, we recall that:

$$x - x_0 \in \mathcal{K}_i(A, r_0) \implies \exists\ s \in \mathbb{R}[x] \ : \deg(s) = i - 1 \ \wedge\ x - x_0 = s(A) r_0 \,, \tag{9}$$

and so:

$$\beta_j \ = \ v_j^T (x - x_0) \ \overset{(9)}{=} \ v_j^T s(A) r_0 \ = \ \big(s(A) v_j\big)^T r_0 \ = \ \big(s(\lambda_j) v_j\big)^T r_0 \ = \ s(\lambda_j)\eta_j \ . \tag{10}$$

Putting together (8) and (10), we have:

$$\begin{aligned} q(\beta_j v_j + x_0) - q(\beta_j^* v_j + x_0) &\overset{(8)}{=} \omega_j \left(\frac{1}{2}\frac{\beta_j^2 \lambda_j^2}{\eta_j^2} + \frac{\beta_j \lambda_j}{\eta_j} + 1\right) \\ &\overset{(10)}{=} \omega_j \left(\frac{1}{2}\lambda_j^2 s(\lambda_j)^2 + \lambda_j s(\lambda_j) + 1\right) \\ &= \frac{1}{2} w_j \Big(\big(\lambda_j^2 s(\lambda_j)^2 + 2\lambda_j s(\lambda_j) + 1\big) + 1\Big) \\ &= \frac{1}{2}\omega_j \Big(\big(\lambda_j s(\lambda_j) + 1\big)^2 + 1\Big) \ = \ \frac{1}{2}\omega_j \Big(p(\lambda_j)^2 + 1\Big) \ , \end{aligned} \tag{11}$$

where p is a polynomial of degree i with constant term equal to 1.

Summing up, from (7) and (11) and recalling that CG applied to q with starting point x_0 finds $x_i \in \mathcal{K}_i(A, r_0)$ such that it minimizes $q(x_i)$, we obtain an expression for $q(x_i) - q(x^*)$:

$$q(x_i) - q(x^*) \;=\; \frac{1}{2} \min_p \sum_{j=1}^{n} \omega_j \Big(p(\lambda_j)^2 + 1 \Big) \; . \tag{12}$$

A consequence of equation (12) is the following convergence result:

Theorem 4.2. CG will always converge to x^* after a at most m iterations, where m is the number of distinct eigenvalues of A.

Proof:
The proof is pretty straightforward. It is sufficient to observe the following facts:

- $\omega_j > 0$ because $\omega_j = \dfrac{\eta_j^2}{\lambda_j}$ and, since A is positive definite, $\lambda_j > 0$
- x_i converges to $x^* \iff q(x_i) - q(x^*) \to 0$. In fact, q is a quadratic function and A is positive definite, thus it has a unique minimum in x^*

So, there exists a polynomial p of degree m such that $p(\lambda_j) = 0$ for $j = 1, 2, \ldots, m$.

□

In a similar way, if the eigenvalues of A are clustered into m groups we can design, by placing a root of p at the center of each cluster, a degree m polynomial p which is relatively small in proximity of each cluster.

In the sum in equation (12), each term corresponds to a direction-restricted objective function $q(\beta_j v_j + x_0) - q(\beta_j^* v_j + x_0) = \omega_j(p(\lambda_j)^2 + 1)$, which is optimized by CG w.r.t. β_js. The smaller these terms, the more progress CG makes in optimizing q. We recall that the weight ω_j measures the total decrease in q that can be obtained by fully optimizing q along the direction v_j. Therefore, by choosing the best polynomial p, CG effectively prioritizes directions with bigger weights.

Finally, we want to present an argument to motivate the tendency of CG to give priority to highly curvature directions, i.e. directions v_j associated with eigenvalues λ_j with higher values.

The proximity to the minimum, measured by $q(\beta_j v_j + x_0) - q(\beta_j^* v_j + x_0)$, is given by $\frac{1}{2}\omega_j \Big(p(\lambda_j)^2 + 1 \Big)$ and depends on the polynomial p. Since p is a polynomial of degree i with constant term equal to 1, it will have the following form:

$$p(z) = \prod_{k=0}^{i} \left(1 - \frac{z}{\nu_k} \right) \; ,$$

where $\nu_k \in \mathbb{R}$. CG looks for a polynomial which minimizes the difference between the value of q at the current point x_i and its optimal value. If ν_k is a root of p, we can consider the effect of this root on $q(x_i) - q(x^*)$:

- if ν_k is near to a large eigenvalue λ_j (corresponding to high curvature), we have that $1 - \frac{\lambda_j}{\nu_k}$ is relative small due to the closeness of ν_k to λ_j, while if λ_i is a small eigenvalue (low curvature), $1 - \frac{\lambda_i}{\nu_k}$ has a small value (≈ 1) because $\lambda_i \ll \nu_k$.

- if ν_k is next to a small eigenvalue λ_i (low curvature), $1 - \frac{\lambda_i}{\nu_k}$ is small as in the previous case due to the closeness of λ_i and ν_k, but this time, for a large eigenvalue λ_j (high curvature), $1 - \frac{\lambda_j}{\nu_k}$ could be very large due to the fact that $\lambda_j \gg \nu_k$ and ν_k is close to 0.

For this reason, in order to optimize q, CG should prefer high-curvature directions (at least if two directions, one with high curvature and one with low curvature, have the same total loss).

Finally, we gave a mathematical ground to some desirable properties of Conjugate Gradient, so that the analysis we did through this report has a stronger foundation.

References

[1] James Martens and Ilya Sutskever. "Training Deep and Recurrent Networks with Hessian-Free Optimization". In: (2012).

[2] James Martens and Ilya Sutskever. "Learning Recurrent Neural Networks with Hessian-Free Optimization". In: *Proceedings of the 28th International Conference on Machine Learning* (Jan. 2011), pp. 1033–1040.

[3] Shun-ichi Amari. "Natural Gradient Works Efficiently in Learning". In: *Neural Computation* 10 (Nov. 2000). DOI: 10.1162/089976698300017746.

[4] Suzanna Becker and Yann Lecun. "Improving the Convergence of Back-Propagation Learning with Second-Order Methods". In: (Jan. 1989).

[5] Nicol Schraudolph. "Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent". In: *Neural computation* 14 (Aug. 2002), pp. 1723–38. DOI: 10.1162/08997660260028683.

[6] James Martens. "Deep learning via Hessian-free optimization". In: *ICML 2010 - Proceedings, 27th International Conference on Machine Learning* (Aug. 2010), pp. 735–742.

[7] Nicholas Gould, Stefano Lucidi, and Massimo Roma. "Solving the Trust-Region Subproblem using the Lanczos Method". In: *SIAM Journal on Optimization* 9 (Feb. 1970). DOI: 10.1137/S1052623497322735.

[8] Olivier Chapelle. "Improved Preconditioner for Hessian Free Optimization". In: 2011.