Heavy Ball Gradient and Conjugate Gradient Analysis for Neural Network Training

Andrea Roncoli, Bianca Ziliotto

May 22nd, 2023

Abstract

This work is focused on the analysis of Heavy Ball Gradient (HBG) and Conjugate Gradient (CG) methods, with particular attention to convergence properties. We will initially expose theoretical bounds for simpler minimization settings, such as the convex one, and then apply it to the harder case of optimization of a neural network. Finally, we ran experiments by training a Multi Layer Perceptron to observe the behaviour of the algorithms.

1 Problem Setting and Notation

We are considering optimization in the context of the training of a Multi Layer Perceptron (MLP) with L layers. Our MLP is involved in a regression task, with a dataset $\mathcal{D} = (\boldsymbol{X}, \boldsymbol{Y})$ consisting of K samples, with input data in $\mathbb{R}^m$, $\boldsymbol{X} = [\boldsymbol{x}_1^T; ...; \boldsymbol{x}_K^T] \subset \mathbb{R}^{Kxm}$ and labels in $\mathbb{R}^n$, $\boldsymbol{Y} = [\boldsymbol{y}_1^T; ...; \boldsymbol{y}_K^T] \subset \mathbb{R}^{Kxn}$ [1] The model thus implements a function $\mathcal{M}_\theta : \mathbb{R}^m \to \mathbb{R}^n$ parameterized by $\boldsymbol{\theta}$, which are the weights and biases of each layer, $\boldsymbol{W}_l, \boldsymbol{b}_l$.

Each layer will perform a nonlinear transformation of its input defined as:

$$\boldsymbol{o}_l = f_l(\boldsymbol{W}_l \boldsymbol{o}_{l-1} + \boldsymbol{b}_l)$$

This is a composition of a linear transformation of the output of the previous layer and an element wise application of a non linear function, the activation function f_l:

$$\boldsymbol{z}_l = \boldsymbol{W}_l \boldsymbol{o}_{l-1} + \boldsymbol{b}_l$$
$$\boldsymbol{o}_l = f_l(\boldsymbol{z}_l)$$

[1] In general, for weights, biases, gradients, ... , we'll indicate matrices (for example in batch applications) with capital letters, and vectors with lower case letters. The notation is greatly inspired, also in algorithms, by [4].

1.1 Activation Function

We choose as activation function the sigmoid function, which is differentiable $\forall x \in \mathbb{R}$.

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

1.2 Objective Function

The objective function comprises two terms: the error function and the penalty term.

$$J(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{Y}) = \mathcal{L}(\hat{\boldsymbol{Y}}, \boldsymbol{Y}) + \lambda \boldsymbol{\Omega}(\boldsymbol{\theta})$$

where $\mathcal{L}(\hat{\boldsymbol{Y}}, \boldsymbol{Y})$ is the error function and $\lambda \boldsymbol{\Omega}(\boldsymbol{\theta})$ is the penalty term.

We choose the Mean Square Error between the real targets $\boldsymbol{Y}$ and the predicted targets $\hat{\boldsymbol{Y}}$ as our error function, where $\hat{\boldsymbol{y}}_k = \mathcal{M}_\theta(\boldsymbol{x}_k)$.

$$\mathcal{L}(\hat{\boldsymbol{Y}}, \boldsymbol{Y}) = \frac{1}{2K}\left\|\hat{\boldsymbol{Y}} - \boldsymbol{Y}\right\|_F^2 = \frac{1}{2K}\sum_{k=1}^{K}\|\hat{\boldsymbol{y}}_k - \boldsymbol{y}_k\|^2$$

As for the penalty term we consider L2, also known as Weight Decay or Tikhonov Regularization.

$$\lambda \boldsymbol{\Omega}(\boldsymbol{\theta}) = \lambda \sum_{l=1}^{L} \|\boldsymbol{W}_l\|_2^2$$

The objective function and the whole architecture result differentiable, as the L2 is a smooth regularization and both MSE and the activation function are differentiable; nevertheless, the sigmoid function provides a non linearity which puts us in a non convex setting.

2 Optimization Algorithms

We are using two iterative optimization algorithms to minimize our objective function, as the properties of the latter do not satisfy the hypothesis for any closed-form solution. The two implemented algorithms both belong to the class of deflected gradient methods.

2.1 Heavy Ball Gradient

Heavy Ball Gradient (HBG) is a gradient descent algorithm based on momentum. This technique is based on the idea of accelerating the descent by accumulating a velocity vector in directions of persistent reduction in the objective across iterations. As presented in [8], this is implemented by the update rule:

$$\boldsymbol{v}_{t+1} = \beta \boldsymbol{v}_t - \alpha \nabla J(\boldsymbol{\theta}_t)$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{v}_{t+1}$$

The direction of the update is computed as a linear combination of the velocity of the previous step and the gradient calculated on the loss of the current step with respect to the parameters. The coefficients of this linear combination are the momentum decay coefficient β and the learning rate α. One can observe this in a clear manner in this image, courtesy of the same paper [8].

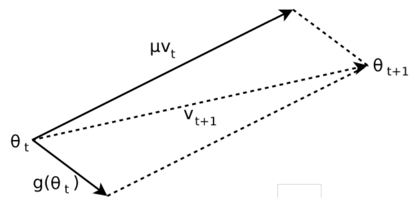

Figure 1: Update rule for HBG

In the case of our MLP, the updates to be made are to weights and biases. The gradients of the objective function with respect to both are calculated across layers through backpropagation. The algorithm for backpropagation is presented in Algorithm 1, and requires to store the outputs $\hat{\boldsymbol{Y}}$ and the latent representations in the layers $\boldsymbol{O}_l$, both of which are computed during the forward pass. The dimension of the batch M can be set from 1 to K, varying from stochastic / mini-batch / full batch versions of the descent steps. Because this is a regression task, there is no activation in the last layer, and thus there is no elementwise multiplication with the gradient of the layer's activation.

Algorithm 1 Backpropagation with batch size M

```
Require: $\boldsymbol{X}_{batch}, \boldsymbol{Y}_{batch}$ batch of inputs and labels given as matrices of shape Mxm, Mxn.
Require: $\boldsymbol{O}_l, \hat{\boldsymbol{Y}}$ calculated during forward pass.
  $\boldsymbol{O}_0 = \boldsymbol{X}_{batch}$
  $\boldsymbol{G} \leftarrow \nabla_{\hat{\boldsymbol{Y}}} J = \nabla_{\hat{\boldsymbol{Y}}} \mathcal{L}(\hat{\boldsymbol{Y}}, \boldsymbol{Y}_{batch})$
  for $l = L, \ldots, 1$ do
      if $l \neq L$ then
          $\boldsymbol{G} \leftarrow \boldsymbol{G} \odot \begin{bmatrix} \boldsymbol{f}'(\boldsymbol{z}_1^l) \\ \ldots \\ \boldsymbol{f}'(\boldsymbol{z}_M^l) \end{bmatrix}$
      end if
      $\nabla_{\boldsymbol{W}_l} J \leftarrow \boldsymbol{O}_{l-1}^T \boldsymbol{G} + \lambda \nabla_{\boldsymbol{W}_l} \Omega(\boldsymbol{\theta})$
      $\nabla_{\boldsymbol{b}_l} J \leftarrow \boldsymbol{1G}$        ▷ $\boldsymbol{1}$ is a $1xm$ matrix of 1s, to sum the columns of $\boldsymbol{G}$
      $\boldsymbol{G} \leftarrow \boldsymbol{G} \boldsymbol{W}_l^T$
  end for
```

Given the gradients for each step, we can update the weights and biases with the HBG strategy, as shown in Algorithm 2, first updating the velocities and then the parameters at each iteration, for each layer.

The stopping condition for the algorithm will be explained in 2.3.

Algorithm 2 Heavy Ball Gradient

```
Require: stopping_condition the stopping condition
Require: $\nabla_{\boldsymbol{\theta}} J_b$ the gradients on the batches w.r.t. parameters given by back-
    propagation
Require: $\boldsymbol{\theta}_0$ the initialized parameters
Require: $\beta, \alpha$ momentum decay coefficient and learning rate
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_0$
  $\boldsymbol{v}^- \leftarrow 0$          ▷ Velocity for parameters of all layers is null at the beginning
  repeat
      for b in batches do
          $\boldsymbol{v} \leftarrow \beta \boldsymbol{v}^- - \alpha \nabla_{\boldsymbol{\theta}} J_b$
          $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
          $\boldsymbol{v}^- \leftarrow \boldsymbol{v}$
      end for
  until stopping_condition
```

2.2 Conjugate Gradient

Conjugate Gradient (CG) methods started out for solving symmetric, positive-definite linear systems of equations, but were soon extended to nonlinear unconstrained optimization. In the quadratic case, the conjugate directions are assured to remain at the minimum of the objective for previous directions, which guarantees convergence in a number of steps equal to the dimension of the parameter space n_p. However, this is no longer true for our kind of objective function, which is far from quadratic.

The procedure, illustrated in Algorithm 3, is based on the following update rule:

$$\boldsymbol{d}_{t+1} = -\nabla J(\boldsymbol{\theta}_t) + \beta_{t+1} \boldsymbol{d}_t$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha_{t+1} \boldsymbol{d}_{t+1}$$

There exist different possible β-formulae, each with different convergence properties, as will be explained in the next section. We chose to implement the following three variants of the algorithm:

1. $\beta_t^{FR} = \frac{\|\boldsymbol{g}_t\|^2}{\|\boldsymbol{g}_{t-1}\|^2}$ *(Fletcher-Reeves)*

2. $\beta_t^{HS+} = \max\{0, \beta_t^{HS}\}$, where $\beta_t^{HS} = \frac{\boldsymbol{g}_t^T \boldsymbol{y}_{t-1}}{\boldsymbol{y}_{t-1}^T \boldsymbol{d}_{k-1}}$ *(Hestenes-Stiefel)*

3. $\beta_t^{PR+} = \max\{0, \beta_t^{PR}\}$, where $\beta_t^{PR} = \frac{\boldsymbol{g}_t^T \boldsymbol{y}_{t-1}}{\|\boldsymbol{g_{t-1}}\|^2}$ *(Polak-Ribire)*

where $\boldsymbol{g}_t = \nabla J(\boldsymbol{\theta}_t)$ and $\boldsymbol{y}_{t-1} = \boldsymbol{g_t} - \boldsymbol{g_{t-1}}$.

Algorithm 3 Conjugate Gradient (Fletcher-Reeves variant)

```
Require: stopping_condition the stopping condition
Require: $J$ the objective function
Require: $\nabla_{\boldsymbol{\theta}} J$ the gradients given by back-propagation
Require: $\boldsymbol{\theta}_0$ the initialized parameters
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_0$
  $\nabla_{\boldsymbol{\theta}} J^- \leftarrow 0$
  $\boldsymbol{d}^- \leftarrow 0$
  repeat
      if $\nabla_{\boldsymbol{\theta}} J^- == 0$ then $\boldsymbol{d} \leftarrow -\nabla_{\boldsymbol{\theta}} J$
      else
          $\beta^{FR} \leftarrow \|\nabla^2_{\boldsymbol{\theta}} J\| / \|\nabla_{\boldsymbol{\theta}} J^-\|^2$
          $\boldsymbol{d} \leftarrow -\nabla_{\boldsymbol{\theta}} J + \beta^{FR} \boldsymbol{d}^-$
      end if
      $\alpha \leftarrow$ Armijo-Wolfe LS$(J, \boldsymbol{\theta}, \boldsymbol{d})$
      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{d}$
      $\boldsymbol{d}^- \leftarrow \boldsymbol{d}$
      $\nabla_{\boldsymbol{\theta}} J^- \leftarrow \nabla_{\boldsymbol{\theta}} J$
  until stopping_condition
```

Since our function is not quadratic, it is not possible to use an exact line search to choose the step-size. Therefore, after $\boldsymbol{d}_{t+1}$ has been determined we run an Armijo-Wolfe inexact line search(Algorithm 4) to compute α_{t+1}. In this way we force our step to satisfy the Armijo and strong Wolfe conditions, i.e.:

$$\phi(\alpha) \leq \phi(0) + m_1 \alpha \phi(0)' \tag{1}$$

$$|\phi(\alpha)| \leq -m_2 \phi(0)' \tag{2}$$

where $\phi(\alpha), \phi'(\alpha)$ are the tomography and its derivative, and $0 < m_1 < m_2 < 1$ are hyperparameters of the algorithm.

We do not perform batch variations on the CG algorithm, as this would imply loosing all guarantees of convergence. In fact, both the Conjugate Gradient method and the Armijo-Wolfe Line Search are designed to enforce specific properties of the step-size and direction; such properties are defined with respect to the function J that we are optimizing, which is computed on the whole training dataset.

In mini-batch versions, the algorithm evaluates one batch at a time, thus we are not actually evaluating the objective function J (or its gradient), but rather a different function J_{batch} which depends only on the batch samples. For this reason, introducing batch variations in our implementation of the CG algorithm would stop ensuring all conditions needed for convergence.

Algorithm 4 Armijo-Wolfe LS

```
Require: $J$ the cost function and $\nabla J$ its gradient
Require: $\boldsymbol{\theta}$ the starting point of the LS
Require: $\boldsymbol{d}$ the direction
Require: $\alpha_s > 0$ the first value we want to test
Require: $\tau$ the factor by which we increase $\alpha$ if AW conditions are not satisfied
Require: $m_1, m_2$ the standard Armijo-Wolfe parameters
Require: $\mathcal{SG}$ safeguard parameter
Require: MaxFeval the maximum number of function evaluations
Require: $\delta, \epsilon$ for the stopping conditions

  $\phi_0 \leftarrow J(\boldsymbol{\theta})$; $\phi'_0 \leftarrow \boldsymbol{d}\nabla J(\boldsymbol{\theta})$    ▷ $1^{st}$ phase
  while feval < MaxFeval do

      $\phi_s \leftarrow J(\boldsymbol{\theta} + \alpha_s \boldsymbol{d})$; $\phi'_s \leftarrow \boldsymbol{d}\nabla J(\boldsymbol{\theta} + \alpha_s \boldsymbol{d})$
      feval = feval + 1

      if $\phi_s \leq \phi_0 + m_1 \alpha_s \phi'_0$ & $|\phi'_0| \leq -m_2 \phi'_0$ then
          $\alpha \leftarrow \alpha_s$
          return $\alpha$    ▷ AW satisfied, we are done
      else if $\phi'_s \geq 0$ then break
      else
          $\alpha_s \leftarrow \alpha_s / \tau$    ▷ Increase $\alpha_s$
      end if
  end while

  $\alpha \leftarrow \alpha_s$; $\alpha_m \leftarrow 0$; $\phi'_m \leftarrow \phi'_0$    ▷ $2^{nd}$ phase
  while feval < MaxFeval & $(\alpha_s - \alpha_m) > \delta$ & $\phi'_s > \epsilon$ do

      $\alpha = (\alpha_m \phi'_s - \alpha_s \phi'_m)/(\phi'_s - \phi'_m)$
      $\alpha = \max\{\alpha_m + (\alpha_s - \alpha_m)\mathcal{SG}, \min\{\alpha_s - (\alpha_s - \alpha_m)\mathcal{SG}, \alpha\}\}$
          ▷ Compute the new $\alpha$ value by safeguarded quadratic interpolation

      $\phi \leftarrow J(\boldsymbol{\theta} + \alpha \boldsymbol{d})$; $\phi' \leftarrow \boldsymbol{d}\nabla J(\boldsymbol{\theta} + \alpha \boldsymbol{d})$
      feval = feval + 1

      if $\phi \leq \phi_0 + m_1 \alpha \phi'_0$ & $|\phi'_0| \leq -m_2 \phi'_0$ then
          return $\alpha$    ▷ AW satisfied, we are done
      else if $\phi' < 0$ then    ▷ Restrict the interval
          $\alpha_m \leftarrow \alpha$
          $\phi'_m \leftarrow \phi'$
      else
          $\alpha_s \leftarrow \alpha$
          $\phi'_s \leftarrow \phi'$
      end if
  end while
  return $\alpha$
```

2.3 Stopping Conditions

As a stopping criterion, we have implemented different possibilities:

- `obj_tol`: stop when objective function between two consecutive steps differs of a quantity inferior to the tolerance
- `grad_norm`: stop when gradient norm drops below a certain threshold
- `max_epochs`: stop after a certain number of epochs
- `n_evaluations`: stop after a certain number of forward/backward passes (useful in considering the difference in complexity of a step with or without the line search)

For the first option, there is a *patience* in terms of steps to ensure that we have reached convergence and the stopping isn't due to a particular fluctuation. This framework is clearly inspired by the one of early stopping for machine learning, even though it has a different objective. Moreover, to ensure that the training will end even if convergence doesn't occur, there is always a maximum number of epochs (default 5×10^3) after which the algorithm stops.

3 Theoretical Analysis

In this section, we report the closest theoretical results about convergence and efficiency of the implemented algorithms, specifying the assumptions under which such results are valid. We then check which of these assumptions actually hold in our case, and consequently which results we expect to observe in our experiments.

Our objective function is in general neither convex nor concave. Due to the choice of activation and regularization functions, we know that our objective function $J(\boldsymbol{\theta})$ is continuously differentiable for all degrees of differentiation, i.e. $J(\boldsymbol{\theta}) \in \mathcal{C}^\infty$. This means of course also $J(\boldsymbol{\theta}) \in \mathcal{C}^2$, which implies that $\nabla J(\boldsymbol{\theta})$ is continuous, i.e. $J(\boldsymbol{\theta})$ is smooth. However, we will see that for proving most convergence results we need a stronger condition on $J(\boldsymbol{\theta})$: L-smoothness. As the second derivative exists and is continuous, we have that $J(\boldsymbol{\theta})$ is L-smooth if and only if the second derivative of $J(\boldsymbol{\theta})$ is bounded.

We can prove that the latter condition is verified as long as the parameters θ remain bounded. In fact, the second derivative of the objective function w.r.t. to any parameter θ_i^l can be written as:

$$\frac{\partial^2 J}{\partial \theta_i^l} = \frac{1}{K} \sum_{k=1}^{K} \Big[\Big\| \frac{\partial \hat{\mathbf{y}}_k}{\partial \theta_i^l} \Big\|^2 + \Big(\hat{\mathbf{y}}_k - \mathbf{y}_k \Big)^T \frac{\partial^2 \hat{\mathbf{y}}_k}{\partial^2 \theta_i^l} \Big] + 2\lambda \tag{3}$$

As illustrated in 1, $\hat{\mathbf{y}}_k$ are obtained by compounding linear combinations (with parameters θ) and sigmoidal activation functions. The outputs of the activation layers are always bounded by $(0, 1)$; the outputs of the fully connected layers are also bounded as long as the inputs and the weights remain bounded. Hence, we can expect $\hat{\mathbf{y}}_k$ to be bounded. The same can be inferred for the first and second derivatives of $\hat{\mathbf{y}}_k$, as both the sigmoid and the linear combination have bounded derivatives with respect to their inputs as long as θ are bounded. Therefore, each term in 3 is bounded and J is L-smooth if the iterates remain in a bounded region.
However, when this condition is no longer valid we loose every guarantee on the L-smoothness assumption. We can then limit ourselves to observe that if the weights do not diverge during our training, L-smoothness condition holds during the run. Even in that case, we know nothing of the value of L, which may be (and usually is) very large.

Before going into details of the different conditions for the convergence of HBG and CG methods, we clarify the notion of *global convergence* to which we will refer throughout the whole section. If $\{\boldsymbol{\theta}_k\}$ is obtained by our algorithm with initial condition $\boldsymbol{\theta}_0$, we say our algorithm is convergent if and only if

$$\lim_{k\to\infty} \inf \|\mathbf{g}_k\| = 0 \tag{4}$$

where $\boldsymbol{g}_k = \nabla J(\boldsymbol{\theta}_k)$. Such convergence is *global* if the previous limit holds for any choice of the initial condition $\boldsymbol{\theta}_0$.

Notice that such condition does not imply that our algorithm ends up in a *global* minimum of J. In general, it only guarantees the convergence to a stationary point.

3.1 Heavy Ball Gradient

3.1.1 Convergence analysis

In [5] (p.168), Ochs provides convergence results for iPiano methods, of which HBG constitutes a particular case.

For general (not necessarily convex) L-smooth functions, global convergence of HBG is guaranteed provided that the hyperparameters are chosen within the following ranges:

$$\beta \in [0, 1)$$
$$\alpha \in (0, 2(1-\beta)/L)$$

Our objective function satisfies the conditions for the validity of this result as long as the iterates remain in a bounded region (thus ensuring L-smoothness): this is not guaranteed a priori, but can be enforced algorithmically by introducing restarts, as presented in [2]. In that case, one can expect the HBG algorithm to converge as long as α and β stay in the indicated ranges. However, as the value of L is unknown at optimization time, values for α and β are actually selected through a grid search.

3.1.2 Efficiency

Regarding the algorithm efficiency, we only have a result on the convergence rate for functions that are both L-smooth and τ-convex. Under such conditions, optimal values for α and β can be computed analytically as:

$$\alpha = (\frac{2}{\sqrt{L} + \sqrt{\tau}})^2$$

$$\beta = (\frac{\sqrt{L} - \sqrt{\tau}}{\sqrt{L} + \sqrt{\tau}})^2$$

For L-smooth τ-convex functions, provided that initial conditions are sufficiently close to the local optimum, it has been proved that this choice of the hyper-parameters guarantees linear convergence of HBG with the following optimal ratio ([6], Theorem 9):

$$r = \frac{\sqrt{L} - \sqrt{\tau}}{\sqrt{L} + \sqrt{\tau}} = \frac{\sqrt{\mathcal{K}} - 1}{\sqrt{\mathcal{K}} + 1}$$

Unfortunately, our objective function does not satisfy the strong convexity condition, hence we cannot expect the previous result to hold. Furthermore, even if we forced our network to learn such a well-conditioned function, the values of L and τ would still be unknown at optimization time.

3.2 Conjugate Gradient

3.2.1 Convergence analysis

Global convergence of conjugate gradient methods greatly depends on the choice of the β-formula as well as on the function conditions. In this section, we report the theoretical convergence results for the three CG methods that we implemented, namely Fletcher-Reeves (β^{FR}), Hestenes-Stiefel (β^{HS+}) and Polak-Ribire (β^{PR+}). The assumptions needed to prove these results are the following:

Assumption 1 *The level set $\mathcal{S} = \{\boldsymbol{\theta} \in \mathbb{R}^{n_p} | J(\boldsymbol{\theta}) \leq J(\boldsymbol{\theta}_0)\}$ is bounded.*

Assumption 2 *In some neighborhood $\mathcal{N} \subset \mathcal{S}$, $J(\boldsymbol{\theta})$ is differentiable and its gradient $\nabla J(\boldsymbol{\theta})$ is Lipschitz continuous.*

Notice that our objective function positively diverges when θ goes to infinity, so 1 always holds.

Moreover, in some neighborhood $\mathcal{N} \subset \mathcal{S}$, i.e. for bounded θ, the second derivative of J is bounded as explained at the beginning of 3, thus also 2 is valid in our case.

Furthermore, since J is continuously differentiable and bounded below in $\mathbb{R}^{n_p}$ by zero, it is always possible to find an α satisfying the Armijo-Wolfe conditions. Therefore the following convergence result can be applied to our specific case.

Al-Baali ([1], Theorem 2) proves the global convergence property (4) for Fletcher-Reeves algorithm under the following assumptions:

1. Assumptions 1 and 2 are valid;
2. Objective function is twice continuously differentiable;
3. The step-length satisfies the strong Wolfe conditions (1 and 2, with $m_2 < 1/2$).

In [3] (Theorem 4.3, Corollary 4.4), Gilbert and Nocedal extend this result and prove global convergence for the non-negative Hestenes-Stiefel and Polak-Ribire methods (β^{HS+} and β^{PR+}), under the following conditions:

1. Assumptions 1 and 2 are valid;
2. The step-length satisfies the strong Wolfe conditions (1 and 2);
3. The sufficient descent condition holds, i.e. $\exists\ 0 < m_3 \leq 1$ s.t. $\langle g_k, d_k \rangle \leq -m_3 \|g_k\|$, $\forall k \geq 1$.

The first two conditions are already satisfied. Furthermore, Gilbert and Nocedal underline that if $\langle g_k, d_{k-1} \rangle \leq 0$, the nonnegativity of β_k implies the validity of the sufficient descent condition. We can easily observe by the way we implemented the Armijo-Wolfe Line Search that $\langle g_k, d_{k-1} \rangle \leq 0$ is verified at every step, as we force $\phi'(\alpha)$ to be nonpositive at every step.

Notice that the sufficient descent condition is also necessary for the convergence of the Fletcher-Reeves algorithm, but in that case it is automatically guaranteed by the strong Wolfe conditions (as shown in [1], Theorem 2).
Notice that this global convergence result only holds for the non-negative Hestenes-Stiefel and Polak-Ribire methods, which guarantee d to be a direction of descent, while the original algorithms (β^{HS} and β^{PR}) do not converge for some objective functions. It is also worth noting that using β^{HS+} or β^{PR+} implies introducing restarts, as deflection term is reset from time to time.

3.2.2 Efficiency

Conjugate gradient methods are characterized by the simplicity of their iteration, numerical efficiency and their low memory requirements. Regarding the space efficiency, CG are widely used in optimization problems with a large number of variables with strict memory constraints, as they do not require to compute and store the Hessian matrix or its approximation.
Despite never failing under previously specified conditions (global convergence has been proven for both exact and inexact line search), Fletcher-Reeves method

is often much slower than Hestenes-Stiefel and Polak-Ribire. An explaination to this lower performance is given by Powell in [7].

However, we have seen that in order to guarantee global convergence for Hestenes-Stiefel and Polak-Ribire we are obliged to use their nonnegative variants: this implies introducing restarts, which are likely to slow down the convergence.

4 Experiments

We propose experiments to examine the behaviour and properties of the two algorithms.

4.1 Dataset and Machinery

The dataset we are working on is the *ML-CUP22*, designed for the Machine Learning course student competition during the 2022/2023 academical year. This dataset comprises 1485 training samples, each of which has 9 numerical input features and 2 numerical target labels. The computations are run on a MacBook Air 2022, with a Apple M2 processor and a 16 GB memory.

4.2 Methodology and General Settings

We are going to analyze mainly the convergence properties of the algorithms, as we are not addressing the problem of generalization, but rather the one of optimization. For this reason, we are not going to consider, as it would be typical in the ML setting, the validation error, but only the training objective function.

On both HBG and CG, for each experimental setting we perform a preliminary grid search and/or some initial tests to find reasonable parameters to analyze the algorithm behaviour. We do this in order to obtain models which lead to a significant comparison.

Furthermore, we fixed reasonable parameters for the line search in CG as in the grid search the ones that yield a finer line search tend to be chosen, allowing the algorithm to make a line search as accurate as possible. This happens, though, at a cost in terms of the number of function evaluations done during the search.

As already stated, we are interested in the optimization of the training objective function, not in the generalization capabilities. For this reason, we have no interest in varying the coefficient λ of the regularization term, as we are not dealing with the problem of defining the function that we want to minimize. In fact, taking the λ which minimizes the objective function disregarding validation would always yield $\lambda = 0$, as it multiplies a strictly positive quantity.

To estimate the value of J^* we initially run the algorithm for a maximum of 5×10^4 epochs (or until convergence in terms of gradient norm is reached). We then run 5 random runs to analyze the convergence of the algorithm, using the `grad_norm` stopping condition as described in 2.3, with a tolerance of 10^{-6} and a smaller number of max epochs.

To reasonably assume that all the random runs are converging to this same minimum, we will preliminarly observe both the mean and the variance of the value reached. If this doesn't happen, we will calculate a different J^* for each run, and then analyze the convergence with respect to each run's minimum.

Unless otherwise specified (i.e. in the minibatch experiment) we will be using full batch.

4.3 Experiment 1: Convex Setting

As a preliminary experiment, we want to analyze the convergence properties of the algorithms on a convex optimization problem. In order to have a convex objective function, we set the activation to the identity, making the whole neural network learn a linear function of the input. Obviously, such a neural network is pretty useless in practice; however, focusing on convex optimization can be interesting to empirically test the convergence properties under convexity assumption.

HBG	$\alpha = 0.005$ $\beta = 0.8$	$3.0236 \pm 1.52 \times 10^{-4}$
CG (β^{FR})	$m_1 = 0.1$ $m_2 = 0.3$	$3.0167 \pm 4.97 \times 10^{-12}$
CG (β^{HS+})	$m_1 = 0.1$ $m_2 = 0.3$	$3.0167 \pm 1.15 \times 10^{-11}$
CG (β^{PR+})	$m_1 = 0.1$ $m_2 = 0.3$	$3.0167 \pm 1.75 \times 10^{-11}$

Table 1: We report the values of the hyper-parameters used in the convex optimization experiments. The mean and standard deviation (over 5 random runs) of the minimum J reached with such parameters is shown in the last column.

Algorithm	Time (s)	Steps	Function evaluations	Time per step (ms)
HBG	3.113	1000.0	1000.0	3.113
CG (β^{FR})	3.234	484.0	1947.4	6.681
CG (β^{HS+})	3.416	397.2	1630.0	8.603
CG (β^{PR+})	4.173	611.0	2465.0	6.830

Table 2: Time complexity analysis of the convex optimization experiment. Function evaluations are the number of forward and backward passes: for HBG full batch they coincide with the epochs, whilst for CG the line search performs multiple function evaluations per gradient step (epoch).

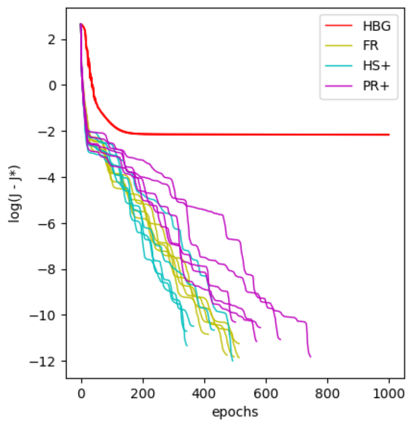

Figure 2: Convex optimization for CG and HBG.

4.3.1 Experiment 1: Observations

We verify that both HBG and all three variants of CG converge to the same value, as indicated by the negligible variance.

Also, we can observe different behaviours, with CG converging at least linearly (expressing this trend after being close enough to the minimum, after about 30 epochs) and HBG converging sublinearly.

We can see that CG converges in fewer steps than HBG, but it actually requires more function evaluations, as expected by the fact that it performs a line search at each step.

Notice that it only makes sense to compare the convergence time amongst the CG variants, as HBG stops before reaching convergence due to the 1000 epochs limit, which makes the time uninformative.

4.4 Experiment 2: Architecture Size Influence

In this experiment and in all the following, we are going to abandon the convex setting, introducing non linearity by using a sigmoid activation function.
In this section we want to analyze the effects of the architectural structure of the network on the speed of convergence of the algorithms. In particular, we want to analyze how the time complexity varies in HBG and CG when increasing the number of parameters in the network, i.e. the number of variables of our optimization problem.
To make this comparison, we trained the following architectures:

- NN: [9, 20, 2]
- NN: [9, 40, 20, 2]

	Run 1	Run 2	Run 3	Run 4	Run 5	Mean $\pm\sigma$
HBG	1.7386	1.7429	1.7616	1.7450	1.7402	1.7457 $\pm$ 8.25 $\times 10^{-3}$
CG (β^{FR})	1.7215	1.7171	**1.7205** (45015)	**1.7159** (42962)	**1.7188** (31133)	1.7188 $\pm$ 2.07 $\times 10^{-3}$
CG (β^{HS+})	**1.7123** (2842)	**1.7133** (4272)	**1.7162** (4272)	**1.7174** (2254)	**1.7190** (1404)	1.7157 $\pm$ 2.54 $\times 10^{-3}$
CG (β^{PR+})	**1.7163** (3385)	**1.7171** (2476)	**1.7163** (6543)	**1.7208** (2922)	**1.7203** (2833)	1.7182 $\pm$ 1.99 $\times 10^{-3}$

Table 3: Estimated values of J^* for the [9, 20, 2] architecture. The algorithm is run for 50k epochs, stopping early if the gradient norm drops below 10^{-6}. The runs stopped for the latter reason are reported in bold, with the stopping epoch in parentheses.

	Run 1	Run 2	Run 3	Run 4	Run 5	Mean $\pm\sigma$
HBG	1.7455	1.7463	1.7402	1.7662	1.7483	1.7493 $\pm$ 8.89 $\times 10^{-3}$
CG (β^{FR})	1.6553	**1.6614** (42183)	1.6693	1.6590	1.6689	1.6608 $\pm$ 4.69 $\times 10^{-3}$
CG (β^{HS+})	**1.6632** (19881)	**1.6580** (13519)	**1.6604** (6948)	**1.6624** (28136)	**1.6621** (13662)	1.6612 $\pm$ 1.83 $\times 10^{-3}$
CG (β^{PR+})	**1.6608** (28681)	**1.6614** (21490)	**1.6640** (11982)	**1.6747** (27383)	**1.6565** (16428)	1.6635 $\pm$ 6.10 $\times 10^{-3}$

Table 4: Estimated values of J^* for the [9, 40, 20, 2] architecture. The algorithm is run for 50k epochs, stopping early if the gradient norm drops below 10^{-6}. The runs stopped for the latter reason are reported in bold, with the stopping epoch in parentheses.

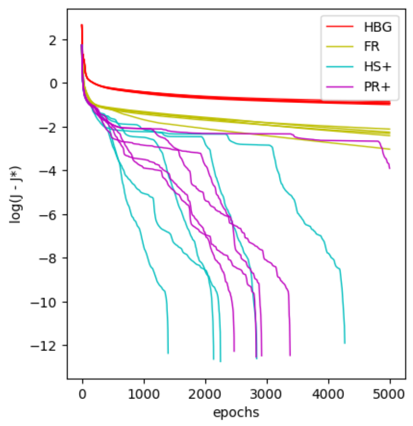

Figure 3: HBG and CG variants behaviour over 5 random runs on the [9, 20, 2] architecture.

During preliminary analysis, we observed high variance amongst the value of J reached during the runs for different initializations, which reasonably suggests that the algorithm finds different local minima for each run. Hence, we

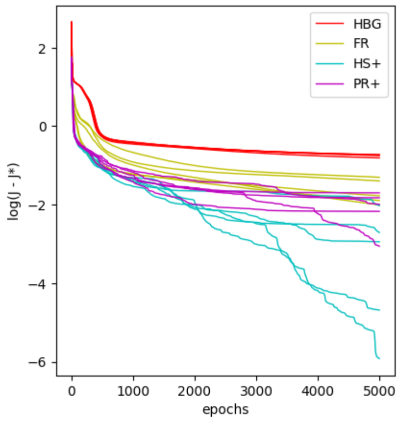

Figure 4: Comparison of HBG and CG variants behaviour over 5 random runs on the [9, 40, 20, 2] architecture.

	Algorithm	Time (s)	Steps	Function evaluations	Time per step (ms)
[9, 20, 2]	**HBG**	20.197	5000.0	5000.0	4.039
	CG (β^{FR})	56.807	5000.0	20333.4	11.361
	CG (β^{HS+})	62.181	2582.2	25947.2	24.082
	CG (β^{PR+})	77.527	3323.2	32255.0	23.329

	Algorithm	Time (s)	Steps	Function evaluations	Time per step (ms)
[9, 40, 20, 2]	**HBG**	51.417	5000.0	5000.0	10.283
	CG (β^{FR})	172.499	5000.0	21622.6	34.500
	CG (β^{HS+})	171.512	5000.0	23489.2	34.302
	CG (β^{PR+})	165.251	5000.0	22095.6	33.050

Table 5: Time complexity analysis of the non-convex optimization experiments, on two different architectures.

computed a different value of J^* for each random initialization, as shown in 3 and 4.

4.4.1 Experiment 2: Observations

We can notice that optimizing with CG a bigger network yields better J^*, whilst with HBG the value stays pretty much the same, which is probably due to the fact that we always stop the algorithm before convergence (after 50k epochs).

The CG algorithms perform, in terms of minimum reached, pretty much equally, with the difference that HS+ and PR+ tend to get there much faster, with HS+ being the fastest, as in both cases they always converge before the maximum of 50k epochs, as opposed to FR.

In the random runs, we can appreciate clearly the difference in time complexity between the average time per step in HBG and CG algorithms, as shown in 5.

Running random runs on the [9, 20, 2] architecture, HS+ and PR+ tend to converge before the `max_epochs`$(5k)$ limit, whilst HBG and FR don't. HS+ is observed to be the fastest in terms of steps. As for the time per step, HS+ and PR+ perform similarly, whilst FR is significantly faster.

In the [9, 40, 20, 2] architecture, no algorithm is able to converge in terms of gradient norm within `max_epochs`. The time per step becomes pretty much the same for all three CG variants and so does the number of function evaluations.

In both architectures, we can observe that the order of (increasing) speed of convergence is: HBG, FR, PR+, HS+.

4.5 Experiment 3: Minibatch

In this experiment, we try to understand the effects of the batch size on HBG. What we would like to observe is the tradeoff between speed and effectiveness that the minibatch versions can achieve.

We initially estimate for each initialization the value of J^* with the value reached at the limit of 50k epochs (as the algorithm never converges before with the gradient norm criterion).

Batch size	Time (s)	Steps	Time per step (ms)	Mean $\pm\sigma$
full batch	5.044	1000	5.044	1.9868 ± 0.0104
256	5.820	6000	0.970	1.7859 ± 0.0134
128	7.410	12000	0.618	1.7528 ± 0.0109
64	8.597	24000	0.358	1.7637 ± 0.0069

Table 6: Time complexity analysis with different batch sizes.

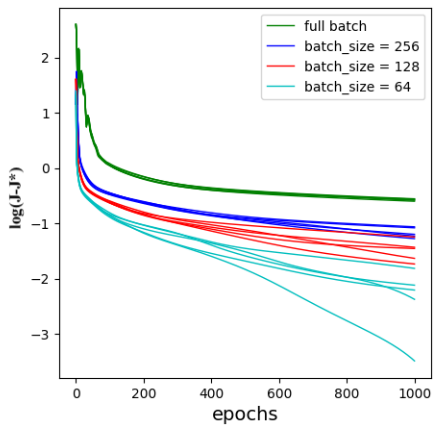

Figure 5: Behaviour of HBG with different batch sizes.

4.5.1 Experiment 3: Observations

In 5 we can appreciate the convergence of HBG method with different batch sizes. Although all variants exhibit a similar behaviour, smaller mini-batches speed up the convergence (in terms of number of epochs) by performing more steps per epoch. Since the number of performed steps is larger, the amount of time required increases when reducing the mini-batch size. However, the time required by a single step is of course larger for bigger mini-batches.

By comparing the last two rows of 6, we can observe that if we continue to diminish the mini-batch size, we fasten the convergence at the cost of worsening the minimum value of J reached. This result suggests that a good compromise for the mini-batch size for HBG in this setting would be 128.

References

[1] Mehiddin Al-Baali. “Descent Property and Global Convergence of the Fletcher–Reeves Method with Inexact Line Search”. In: *IMA Journal of Numerical Analysis* 5 (Jan. 1985). DOI: `10.1093/imanum/5.1.121`.

[2] 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.

[3] Jean Charles Gilbert and Jorge Nocedal. “Global Convergence Properties of Conjugate Gradient Methods for Optimization”. In: *SIAM Journal on Optimization* 2.1 (1992), pp. 21–42. DOI: `10.1137/0802003`. eprint: `https://doi.org/10.1137/0802003`. URL: `https://doi.org/10.1137/0802003`.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[5] Peter Ochs. “Local Convergence of the Heavy-Ball Method and iPiano for Non-convex Optimization”. In: *Journal of Optimization Theory and Applications* 177 (Apr. 2018). DOI: `10.1007/s10957-018-1272-y`.

[6] Boris Polyak. “Some methods of speeding up the convergence of iteration methods”. In: *Ussr Computational Mathematics and Mathematical Physics* 4 (Dec. 1964), pp. 1–17. DOI: `10.1016/0041-5553(64)90137-5`.

[7] M. J. D. Powell. “Nonconvex minimization calculations and the conjugate gradient method”. In: *Numerical Analysis*. Ed. by David F. Griffiths. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 122–141. ISBN: 978-3-540-38881-4.

[8] 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.