Experiments on Gradient Caching in Gauss-Southwell BCGD Compared to Batch Gradient Descent for Semi-Supervised Learning

Mikael Poli[a] **and Hazeezat Adebimpe Adebayo**[b]

[a]*mikael.poli@studenti.unipd.it*
[b]*hazeezatadebimpe.adebayo@studenti.unipd.it*

Abstract—These experiments explore how different optimization strategies affect both accuracy and computation time in a semi-supervised learning task. We compared gradient descent with several versions of Gauss-Southwell Block Coordinate Gradient Descent (BCGD-GS) on two simple 3D datasets. Our results show how increasing block size improves accuracy, and that using gradient caching can cut per-iteration CPU time roughly in half. These small implementation changes can make a big difference in performance.

Keywords—*Gradient Descent, Gauss-Southwell BCGD, Gradient Caching, Semi-Supervised Learning*

1. Problem Definition: The Semi-Supervised Setting

We are working on a semi-supervised learning problem, where we have access to a mix of labeled and unlabeled data:

- l labeled examples $(\bar{x}^i, \bar{y}^i)$ for $i = 1, ..., l$
- u unlabeled examples (x^j, y^j) for $j = 1, ..., u$

Under the smoothness assumption that nearby points should share similar labels, we define feature-based similarity weights. Using inverse Euclidean distance with $\epsilon = 1 \times 10^{-6}$ to avoid division by zero as our similarity metric:

- $W_{ij} = 1/(\|\bar{x}^i - x^j\| + \epsilon)$ between labeled $\bar{x}^i$ and unlabeled x^j
- $\bar{W}_{ij} = \bar{W}_{ji} = 1/(\|x^i - x^j\| + \epsilon)$ between unlabeled x^i and x^j

We want to find the set of labels $y = (y^1, ..., y^u) \in \mathbb{R}^u$ for the unlabeled data that solves:

$$\min_{y \in \mathbb{R}^u} f(y) \tag{1}$$

$$f(y) = \sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij}(y^j - \bar{y}^i)^2 + \frac{1}{2}\sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij}(y^i - y^j)^2 \tag{2}$$

The first term (the "fit" term) ensures that each unlabeled point gets a label similar to nearby labeled examples, while the second term (the "smoothness" term) enforces label smoothness, meaning that similar unlabeled points get similar predicted labels. The gradient of $f(y)$ w.r.t. each unlabeled example y^j is:

$$\nabla_{y^j} f(y) = 2\sum_{i=1}^{l} w_{ij}(y^j - \bar{y}^i) + 2\sum_{i=1}^{u} \bar{w}_{ij}(y^j - y^i) \tag{3}$$

2. Implementation Details

2.1. Algorithms

We solve the problem using Gradient Descent (GD) and Gauss-Southwell Block Coordinate Gradient Descent (BCGD-GS) using the general schemes:

Algorithm 1 Gradient Descent

Choose a point $x_1 \in \mathbb{R}^n$
for $k = 1, ...$ **do**
  if x_k satisfies some specific condition **then**
    STOP
  end if
  Set $x_{k+1} = x_k - \alpha \nabla f(x_k)$
end for

Algorithm 2 BCGD with Gauss-Southwell Rule, Block Size s_b, and Optional Gradient Caching

Choose a point $x_1 \in \mathbb{R}^n$, block size s_b, and set `use_cache` $\in$ {`True`, `False`}
if `use_cache` **then**
  Compute full gradient: $g = \nabla f(x_1)$
end if
for $k = 1, ...$ **do**
  if x_k satisfies some specific stopping condition **then**
    STOP
  end if
  if `use_cache` is `False` **then**
    Compute full gradient: $g = \nabla f(x_k)$
  end if
  Set $y_0 = x_k$
  Select block $B \subseteq \{1, ..., n\}$ of size s_b corresponding to top-s_b entries of $|g|$
  Update:
$$y_1 = y_0 - \alpha \mathbf{U}_b \nabla_B f(y_0)$$
  Set $x_{k+1} = y_1$
  if `use_cache` **then**
    Update cached gradient: $g_B = \nabla_B f(x_{k+1})$
  end if
end for

We begin by implementing BCGD-GS with 1D blocks, both with and without gradient caching, to appreciate the impact of caching on CPU time. We then extend the cached version to 10D and 100D blocks to evaluate the effect of larger block sizes.

2.1.1. Learning Rate

We use a Lipschitz-constant-based fixed stepsize for both algorithms:

$$\alpha = \frac{1}{L} = \frac{1}{\lambda_{\max}(2D + 2\bar{L})} \tag{4}$$

where $2D + 2\bar{L}$ is the Hessian of the objective function $f(y)$, $H \in \mathbb{R}^{u \times u}$, with:

- $D_{jj} = \sum_{i=1}^{l} w_{ij}$ a degree matrix
- $\bar{L} = \bar{D} - \bar{W}$, with $\bar{D}ii = \sum j = 1^u \bar{w}_{ij}$ a graph Laplacian

2.1.2. Stopping Conditions

We implement both gradient-based and accuracy-based early stopping. Early stopping occurs when one of these conditions is met:

- Accuracy stagnates or decreases for 20 consecutive iterations or reaches 99%, or
- Gradient norm $< 1 \times 10^{-6}$

We set the maximum number of iterations to 1000.

2.2. Evaluation

We run the algorithms on two datasets, both with $n = 10,000$ examples, balanced classes, z-score normalized features, and 10% of the examples labeled. We have:

- Toy dataset: 3D Gaussian blobs centered at $(-3, 3, 2)$ and $(3, -3, -2)$ with labels ± 1. We primarily used it as a sanity check for our code.

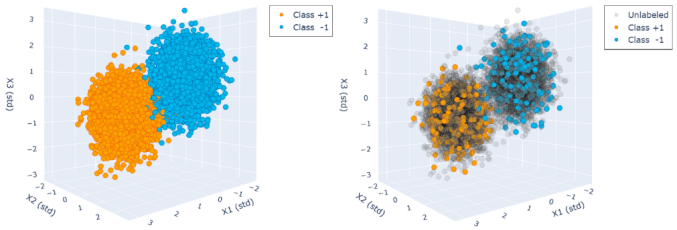


Figure 1. Toy dataset before and after labeled-unlabeled split.

- Rice Type Classification Dataset on Kaggle: Real-world data with *Eccentricity*, *Area*, and *MinorAxisLength* as features, and binary class labels in *Class* mapped to ± 1

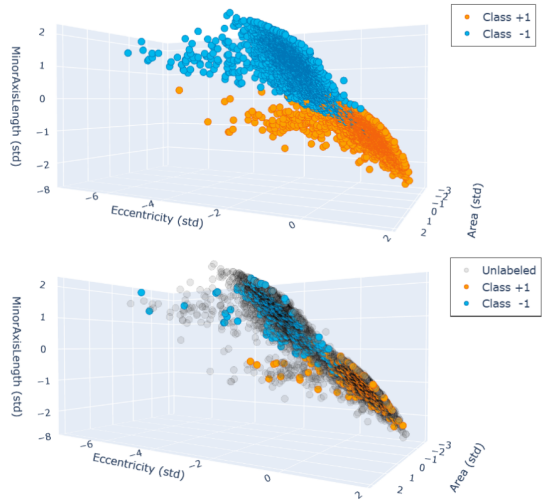


Figure 2. Rice Type Classification dataset after labeled-unlabeled split.

We compare methods based on accuracy, total and per-iteration CPU time, number of iterations, and objective function progression.

We implement all experiments in Python 3.11.12 with consistent settings across algorithms to ensure that any performance differences are attributable to the algorithms themselves.

3. Results

For each dataset, we report a table of results including total number of iterations, total CPU time, accuracy, and reason for stopping. Fig. 3 and fig. 4 include plots showing each algorithm's behavior in terms of accuracy and loss value progression. For both datasets, the algorithm that takes the longest to run is BCGD-GS 1D without gradient caching, which takes $\approx 15 - 20$ minutes to converge on Google Colab's CPU setup (Intel(R) Xeon(R) CPU @ 2.20GHz with 2 vCPUs).

Interactive plots showing each algorithm's results against a decision boundary interpolated by a k-NN classifier with $k = 1$ can be found in the 'main.ipynb' notebook on GitHub. Their animated versions can be found in the README page of the project on GitHub.

Table 1. Results for the toy dataset.

Algorithm	Iters	CPU	Acc	Stop Reason
GD	3	3.01s	99.156%	Acc > 99%
BCGD 1D NC	1000	949.32s	61.32%	Max iter
BCGD 1D C	1000	514.15s	56.989%	Max iter

Note: NC = without gradient caching, C = with gradient caching.

Table 2. Results for the Rice Type Classification dataset.

Algorithm	Iters	CPU	Acc	Stop Reason
GD	35	39.32s	98.378%	Acc plateau
BCGD-1D NC	1000	994.05s	58.711%	Max iter
BCGD-1D C	1000	502.13s	55.156%	Max iter
BCGD-10D C	1000	500.93s	90.411%	Max iter
BCGD-100D C	460	238.06s	98.111%	Acc plateau

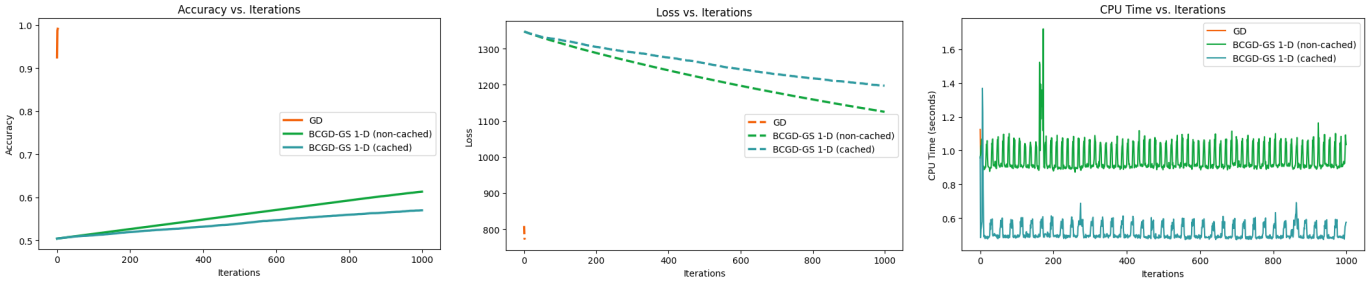
Note: NC = without gradient caching, C = with gradient caching.

4. Discussion

We observe very similar performance across both datasets, likely due to their inherent similarity, particularly in terms of linear separability. On both datasets, gradient descent (GD) achieved the desired accuracy threshold in significantly fewer iterations compared to all versions of Gauss-Southwell Block Coordinate Gradient Descent (BCGD-GS). The 1D BCGD-GS versions produced nearly identical results in terms of both loss function value and accuracy, with both reaching the maximum number of iterations. Although the accuracy of both versions increased very slowly and the loss continued to decrease, performance remained unsatisfactory (approximately 50%, i.e., chance level).

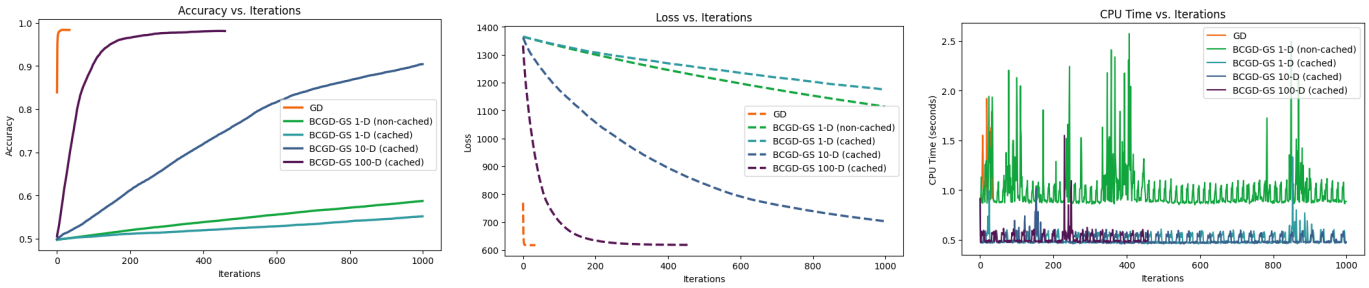
The 10D version also reached the maximum number of iterations but achieved significantly higher accuracy than the 1D versions (approximately 90%). On the real-world dataset, the 100D version of BCGD-GS reached an accuracy comparable to that of GD (approximately 98%). On that dataset, both algorithms stopped early due to the accuracy plateauing.

Total CPU usage was higher for all BCGD-GS versions compared to GD, due to both the significantly higher number of iterations and, in the case of the 1D version, the lack of gradient caching. However, gradient caching halved the per-iteration CPU time, which remained consistently around ≈ 0.5s for the BCGD-GS versions that used it, compared to ≈ 1.0s for the 1D BCGD-GS and GD.



(a) Accuracy across iterations. **(b)** Loss across iterations. **(c)** CPU time per iteration.

Figure 3. Toy dataset: accuracy, loss, and per-iteration CPU time.



(a) Accuracy across iterations. **(b)** Loss across iterations. **(c)** CPU time per iteration.

Figure 4. Rice type classification dataset: accuracy, loss, and per-iteration CPU time.

Overall, these results align with our expectations: larger block sizes in BCGD-GS lead to higher accuracy and faster convergence, and gradient caching significantly reduces BCGD-GS's CPU time.

5. Conclusion

We compared batch gradient descent (GD) with four versions of Gauss-Southwell Block Coordinate Gradient Descent (BCGD-GS): three using gradient caching (with 1D, 10D, and 100D blocks), and one without caching (with 1D blocks), across two datasets. GD achieved higher accuracy much faster than all BCGD-GS versions, while both 1D BCGD-GS versions struggled to surpass chance-level accuracy for binary classification (50%). Increasing the block size in BCGD-GS substantially improved accuracy, with the 100D version matching the performance of GD.

In terms of total CPU usage, all BCGD-GS versions were more computationally expensive than GD, with the 1D non-cached version being the most costly. However, per-iteration CPU usage was halved in cached BCGD-GS versions compared to both the non-cached version and GD.

Of course, the choice, implementation, and practical feasibility of these algorithms should depend on the dataset's characteristics and size — for extremely large datasets, 100D blocks may not be feasible, and Gauss-Southwell block selection may not be the most efficient.

6. Software

The code, data, figures, and documentation used in the experiments can be found on GitHub.