Optimization for Machine Learning

Master 2 Artificial Intelligence, System and Data

March 10, 2024

Student:

Quyen Linh TA

Contents

1 Second-order Optimization Methods

1.1 Newton's Method

Implementation Newton's Method

Throughout the project, my implementation of Newton's Method and also other algorithms are encapsulated within a Python class structure, leveraging object-oriented principles for modularity and reusability. The core computation Newton's method is performed in the `_step` method, which calculates the Newton step using the inverse Hessian and the gradient at the current point.

Application of Newton's Method to a Quadratic Problem

Consider applying Newton's method to the following quadratic optimization problem:

$$\min_{\boldsymbol{w}\in\mathbb{R}^3} q(\boldsymbol{w}) := 2(w_1 + w_2 + w_3 - 3)^2 + (w_1 - w_2)^2 + (w_2 - w_3)^2 \tag{1}$$

This function is strongly convex with a unique minimizer $\boldsymbol{w}^* = \begin{pmatrix}1 & 1 & 1\end{pmatrix}^T$.

The gradient and Hessian of q are as follows:

$$\nabla q(\boldsymbol{w}) = \begin{pmatrix} 6w_1 + 2w_2 + 4w_3 - 12 \\ 2w_1 + 8w_2 + 2w_3 - 12 \\ 4w_1 + 2w_2 + 6w_3 - 12 \end{pmatrix}, \tag{2}$$

$$\nabla^2 q(\boldsymbol{w}) = \begin{pmatrix} 6 & 2 & 4 \\ 2 & 8 & 2 \\ 4 & 2 & 6 \end{pmatrix}, \tag{3}$$

with the inverse of the Hessian being:

$$(\nabla^2 q(\boldsymbol{w}))^{-1} = \frac{1}{36} \begin{pmatrix} 11 & -1 & -7 \\ -1 & 5 & -1 \\ -7 & -1 & 11 \end{pmatrix}. \tag{4}$$

The iterative update formula in Newton's method is:

$$\boldsymbol{w}_{\text{new}} = \boldsymbol{w} - (\nabla^2 q(\boldsymbol{w}))^{-1} \nabla q(\boldsymbol{w}). \tag{5}$$

Substituting $\nabla q(\boldsymbol{w})$ and $(\nabla^2 q(\boldsymbol{w}))^{-1}$ into the update formula yields:

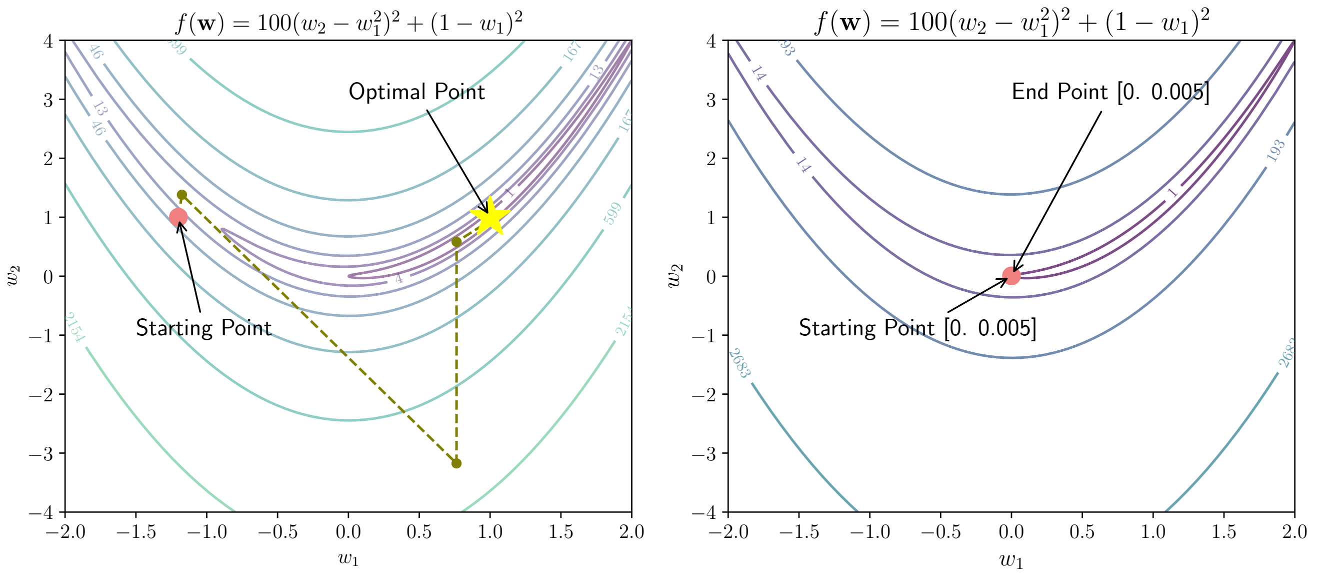
$$\begin{aligned} \boldsymbol{w}_{\text{new}} &= \boldsymbol{w} - \frac{1}{36} \begin{pmatrix} 11 & -1 & -7 \\ -1 & 5 & -1 \\ -7 & -1 & 11 \end{pmatrix} \begin{pmatrix} 6w_1 + 2w_2 + 4w_3 - 12 \\ 2w_1 + 8w_2 + 2w_3 - 12 \\ 4w_1 + 2w_2 + 6w_3 - 12 \end{pmatrix}, && (6) \\ &= \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} - \begin{pmatrix} w_1 - 1 \\ w_2 - 1 \\ w_3 - 1 \end{pmatrix}, && (7) \\ &= \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}. && (8) \end{aligned}$$

Upon a single iteration, the algorithm converges to $\boldsymbol{w} = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}^T$. To verify the optimality, we evaluate $q(\boldsymbol{w})$:

$$q(\boldsymbol{w}) = 2(1 + 1 + 1 - 3)^2 + (1 - 1)^2 + (1 - 1)^2 = 0, \tag{9}$$

which confirms that $\boldsymbol{w} = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}^T$ is indeed the global minimizer, as the objective function attains its minimum value of zero at this point.

Therefore, we have demonstrated that Newton's method, when applied to the given strongly convex quadratic problem, efficiently converges to the global minimum in a single iteration.



■ **Figure 1**: With Newton's method applied from different starting points. Left: The convergence trajectory from the starting point $[-1.2, 1]$ to the optimal point $[1, 1]$. Right: Failure of Newton's method to iterate from the starting point $[0, 0.005]$, illustrating the method's local convergence nature and the importance of the initial starting point.

Rosenbrock function with Newton's method

We now consider the celebrated Rosenbrock function

$$\underset{\boldsymbol{w}\in\mathbb{R}^2}{\text{minimize}} \quad 100\left(w_2 - w_1^2\right)^2 + \left(1 - w_1\right)^2. \tag{10}$$

Newton's method was applied to the Rosenbrock function, a classical test case for optimization algorithms. Starting from two different initial points, the behavior of the algorithm was observed.

Algorithm	Initial Point	Final Point	Iterations
Newton's Method	$\begin{bmatrix}-1.2 & 1\end{bmatrix}$	$\begin{bmatrix}1.0 & 1.0\end{bmatrix}$	5
Newton's Method	$\begin{bmatrix}0 & 5\times10^{-3}\end{bmatrix}$	$\begin{bmatrix}0 & 5\times10^{-3}\end{bmatrix}$	0 (Singular matrix)
Newton's Method	$\begin{bmatrix}0 & \frac{1}{200}+10^{-12}\end{bmatrix}$	$\begin{bmatrix}-5\times10^{9} & 2.5\times10^{19}\end{bmatrix}$	1000

Table 1: Results for the Rosenbrock function optimization using Newton's Method.

From the initial point $\boldsymbol{w}_{01} = [-1.2, 1]$, Newton's method exhibited convergence to a point very close to the known minimum at $[1, 1]$ after 5 iterations, without encountering any numerical issues.

A different behavior was observed when the method was initialized at $\boldsymbol{w}_{02} = [0, \frac{1}{200} + 10^{-12}]$ led to a large divergence, with the point moving away to an order of magnitude of 10^{19} after 1000 iterations. This demonstrates the local nature of Newton's method; while it can converge rapidly when close to the optimum, it can also diverge significantly if the initial point is not well-chosen.

Furthermore, initializing the algorithm at $\boldsymbol{w}_{03} = [0, 0.005]$. Here, the method failed to perform even a single iteration due to the singularity of the Hessian matrix, illustrating the necessity for a well-defined inverse Hessian to proceed with the algorithm.

These points collectively manifest the intrinsic local optimization characteristic of Newton's method. It relies on local curvature (second-order derivatives) to find the minimizer, which is potent near the solution but can lead to failure or divergence if the initial guess is not within a good local neighborhood of any minimizer.

1.2 Globally Convergent of Newton's Method

My implementation of Globalized Newton's extends Newton's Method class by incorporating a globalization strategy. λ adjustment adjusts the Hessian with a λ_k value before computing the search direction, ensuring the matrix is positive definite for enhanced stability. Armijo line search is employed to determine an appropriate step size, improving robustness by ensuring that sufficient decrease conditions are met.

We can observe better result in comparing with figure 1 where the result of Newton's Method without Line Search led to a large divergence with starting point $\boldsymbol{w}_{02} = [0, \frac{1}{200} + 10^{-12}]$. In figure 2, with Line Search, it demonstrates that the inclusion of the line search effectively adjusts the step size, ensuring convergence even when the initial point is not in the immediate vicinity of the minimum. The trajectory in the right plot, originating at $\left(0, \frac{1}{200} + 10^{-12}\right)$, exhibits an convergence, likely due to the flat nature of the Rosenbrock function's valley around the origin. Nonetheless, the trajectory successfully reaches the optimal point. For the left plot with the starting point [-1.2, 1], the trajectory of the Globalized Newton's method with line search converges smoothly to the optimal point. Throughout the first section, all algorithms using same logic implementation of Armijo Line Search.

This behavior underscores the line search's critical role in ensuring convergence for the Globalized Newton's method. While the parameter θ influences the rate of convergence reflected by the trajectory's shape the overall ability to reach the optimal point from varied starting positions is not compromised, emphasizing the method's global convergence capabilities.

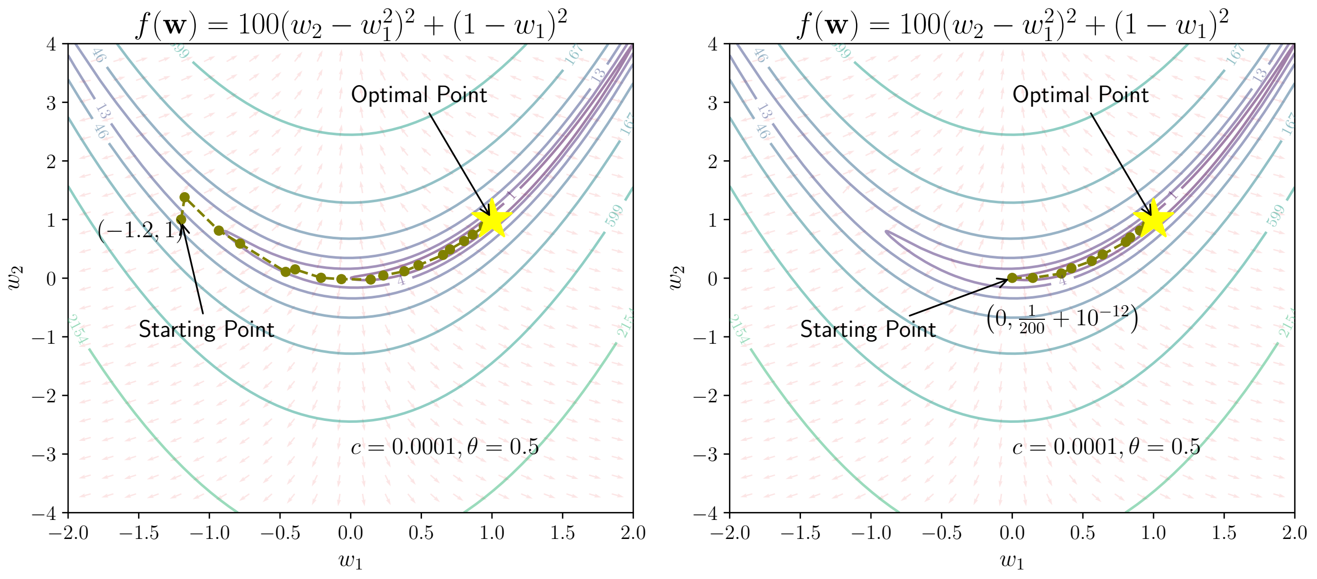


Figure 2: Globalized Newton's Method to Rosenbrock function's values, and the paths depict the progression of the algorithm from the starting point to the optimal point

Tested with different c values while fixed $\theta = 0.5$, c is integral to the Armijo line-search, dictating the necessary decrease in the objective function for an acceptable step size. The convergence process appears relatively insensitive to variations in c within the range of 0.0001 to 0.1, consistently requiring 13 iterations.

However, a slight increase in iteration count to 14 is observed at $c = 0.5$, indicating marginal sensitivity to higher c values.

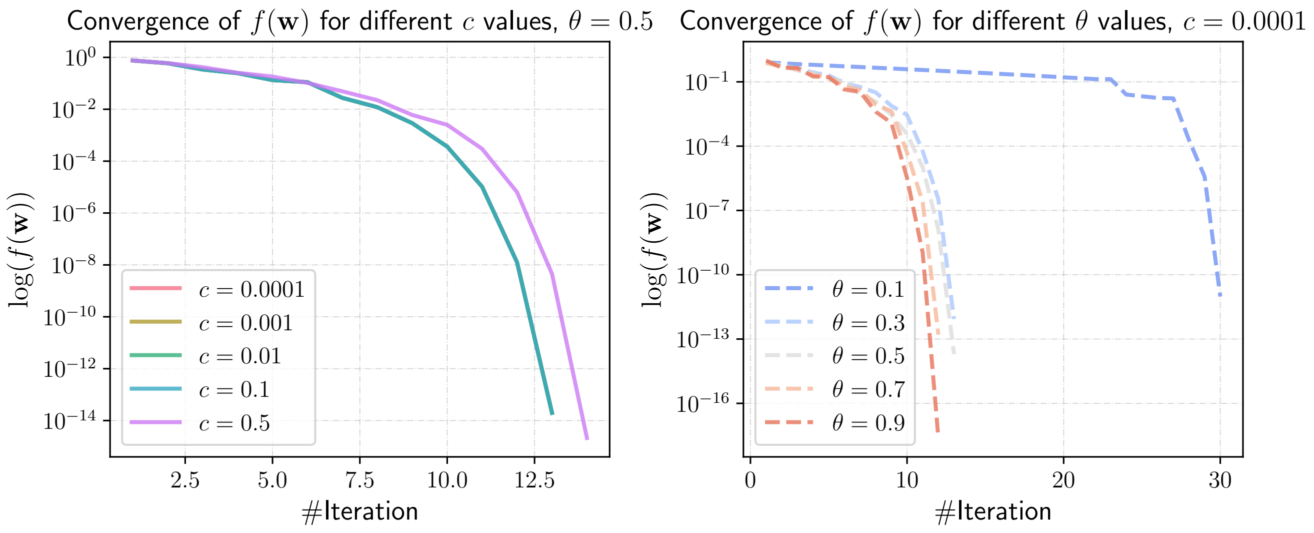


Figure 3: Globalized Newton's method in comparing of differences c and θ

With different θ and $c = 0.0001$, it was affecting the step size reduction during line-search, the algorithm shows more pronounced sensitivity to θ. A gradual reduction in required iterations is observed as θ increases from 0.1 to 0.9, with a notable efficiency and accuracy improvement at higher θ values, particularly at 0.9 which achieves convergence in 12 iterations to the precise final point $[1, 1]$.

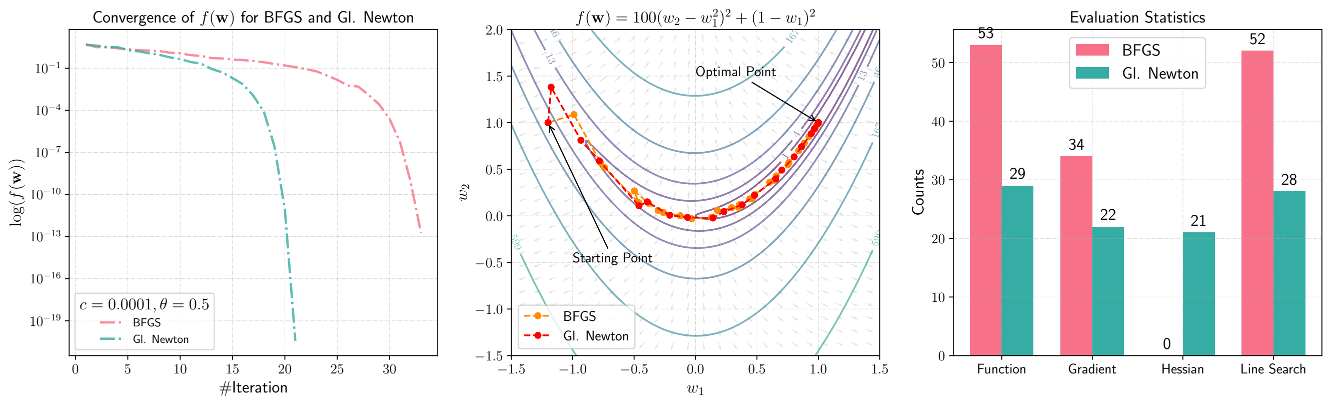
As theorized by [Armijo, 1966] and further discussed by [Nocedal and Wright, 2006], the parameter c in the line search conditions affects the aggressiveness of step size acceptance. The observed results confirm the theoretical expectations where a smaller c does not significantly impact the iteration count within a certain range. However, as c increases, the number of iterations also increases slightly, indicating the sensitivity of the method to higher c values. Similarly, for θ, a larger value implies less aggressive step size reductions, leading to faster convergence, which is in line with our observations and the discussions in optimization literature.

What I could observe that Globalized Newton's method exhibits greater sensitivity to θ than to c, with a higher θ value nearing 1 proving beneficial for both convergence efficiency and solution accuracy. Meanwhile, the algorithm's performance remains robust across a reasonable range of c values, underscoring the importance of carefully selecting θ for optimal algorithm performance.

1.3 Quasi-Newton methods and BFGS

Globalized Newton's method is theoretically favored in terms of local convergence rates, expected to demonstrate quadratic convergence given sufficient smoothness and strong convexity conditions. On the other hand, the BFGS quasi-Newton method is anticipated to show superlinear convergence, sacrificing convergence speed for reduced computational demands per iteration.

With the Rosenbrock function, same starting point, the Globalized Newton's method required 21 iterations to converge, as opposed to the BFGS method, which took 33 iterations. This is consistent with the expected faster convergence of methods leveraging exact Hessian information. From a computational perspective, the Globalized Newton's method necessitated 22 gradient and 21 Hessian evaluations, whereas BFGS required 34 gradient evaluations. Given that each Hessian evaluation equates to d gradient evaluations, the effective computational cost for the Globalized Newton's method can be viewed as



■ **Figure 4**: Comparison Quasi-Newton BFGS vs Globalized Newton on Rosenbrock.

$22 + 21d$ gradient evaluations. This comparison underscores a theoretical compromise between the rapid convergence offered by exact second-order methods and the computational efficiency of quasi-Newton approaches.

The BFGS method relies on an approximation of the inverse Hessian which is iteratively updated using gradient evaluations. As such, the accuracy of the Hessian approximation and, consequently, the search direction, is heavily dependent on gradient information. If the approximated inverse Hessian deviates from the true inverse Hessian, the search direction may be less effective, potentially requiring more function evaluations and line search steps to ensure adequate descent.

Function evaluations in BFGS are driven by the line search algorithm, which repeatedly evaluates the objective function to find a step size that satisfies the Armijo condition. More function evaluations suggest that the BFGS method, in this case, takes smaller steps and thus requires more iterations to confirm the descent condition. This can happen when the search direction is not aligned well with the true descent direction, often due to an inaccurate Hessian approximation. Similarly, the number of line search steps correlates with the effort needed to find a suitable step size that meets the Wolfe conditions, particularly the Armijo rule for sufficient decrease. A higher count indicates the algorithm often adjusts the step size, which implies the initial guesses are often not suitable. The BFGS method may take conservative steps because the approximate inverse Hessian might not capture the local curvature as accurately as the true Hessian does in the Globalized Newton's method. Therefore, it may require more line search iterations to find a step size that results in an acceptable decrease in the function value.

More result comparing on quadratic function (1) and Rosenbrock (10) is available in the notebook.

1.4 Limited-memory BFGS

While BFGS, with its full memory utilization, demonstrates robust performance by converging in 33 iterations, the L-BFGS variant showcases a nuanced dependence on the memory size parameter m, which dictates its approximation quality of the BFGS algorithm.

With $m = 0$, essentially equivalent to employing gradient descent, L-BFGS exhibits a drastic increase in both function and gradient evaluations, necessitating 8058 iterations for convergence. This starkly contrasts with scenarios where $m > 0$, indicating the critical role of leveraging historical gradients and update steps in enhancing convergence rates. Notably, as m increases to 1, the iteration count drops significantly to 42, underscoring the efficiency gain from even a minimal memory footprint. Further increments of m to 5 and beyond result in convergence metrics that not only approximate but, in the case of $m = 10$, slightly surpass the BFGS method's efficiency. This progression highlights L-BFGS's capability to adaptively refine its search direction based on a limited yet recent history, striking a balance between computational overhead and convergence speed.

Optimizer	Function Evaluations	Gradient Evaluations	Line Search Steps	Iterations
BFGS	53	34	52	33
L-BFGS ($m = 0$)	80045	8059	80044	8058
L-BFGS ($m = 1$)	368	43	367	42
L-BFGS ($m = 5$)	65	36	64	35
L-BFGS ($m = 10$)	52	33	51	32

Table 2: Comparison of BFGS and L-BFGS Optimizers for Rosenbrock Function (10).

We can illustrate the effectiveness of the L-BFGS method in optimizing computational resources without compromising on convergence quality. Particularly, it demonstrates the potential of L-BFGS in applications where memory is a constraint, offering a scalable alternative to the traditional BFGS method without significant loss in performance efficacy.

In particular, the sharp decrease in function evaluations for L-BFGS ($m = 0$) to L-BFGS ($m = 1$) and further to L-BFGS ($m = 5$) underscores the method's adaptability and computational interest. The data suggests that by intelligently managing its limited memory, L-BFGS can approximate the beneficial properties of BFGS—such as accurate search directions and efficient convergence—while significantly reducing the computational burden. Therefore, this improvement, as observed in the given scenarios, affirms the theoretical computational interest of employing a limited memory strategy in quasi-Newton methods.

2 Stochastic Second-Order Methods

Toward our forthcoming discussion, we work with the dataset $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ where $\boldsymbol{x}_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$. For each component function f_i, the gradient and Hessian are defined as:

$$\nabla f_i(\boldsymbol{w}) = -\frac{y_i}{1 + \exp\left(y_i \boldsymbol{x}_i^T \boldsymbol{w}\right)} \boldsymbol{x}_i + \lambda \boldsymbol{w}, \tag{11}$$

$$\nabla^2 f_i(\boldsymbol{w}) = \frac{\exp\left(y_i \boldsymbol{w}^T \boldsymbol{x}_i\right)}{\left(1 + \exp\left(y_i \boldsymbol{w}^T \boldsymbol{x}_i\right)\right)^2} \boldsymbol{x}_i \boldsymbol{x}_i^T + \lambda \boldsymbol{I}_d. \tag{12}$$

These expressions will underpin the upcoming discourse on stochastic variants of second-order methods adapted to the logistic regression framework.

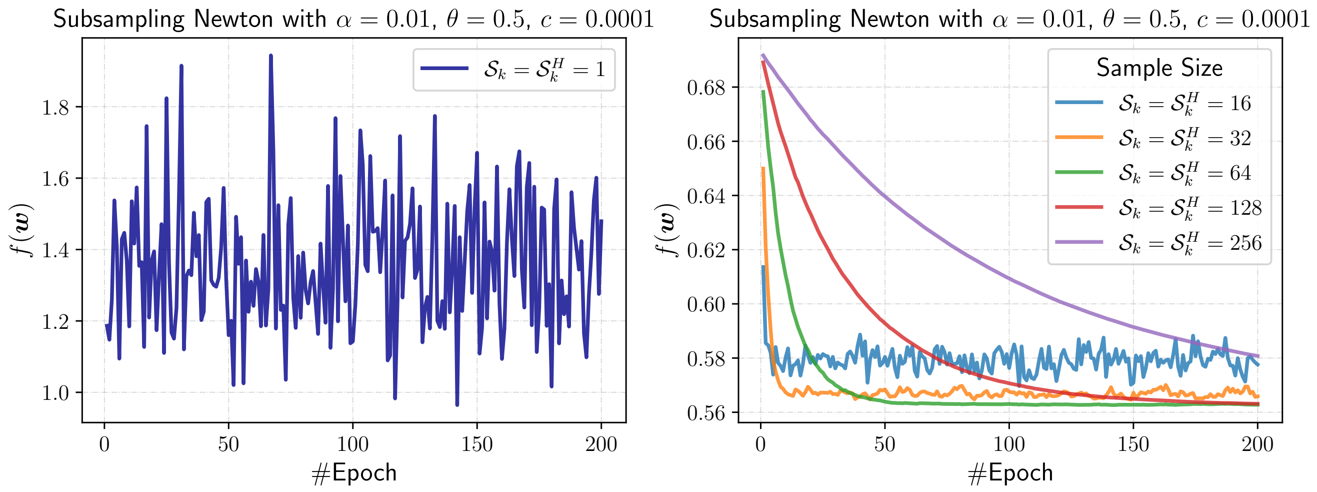
Starting with this section, the implementation transitions to an epoch-based approach, with the sample size determined as per the specific instructions of each question. Additionally, a subsampled Armijo line-search procedure has been incorporated, adhering to the prescribed sample size parameters.

2.1 Subsampling Newton methods

In Lab 4, we have a class `RegPb` encapsulating the instance problem with a synthetic dataset designed for testing optimization algorithms. The dataset, created using a linear model, is characterized by features of dimension $d = 50$ and comprises 1000 samples. The loss function used is the L2 loss, with a regularization term parameterized by $\lambda = 1/n^{0.5}$, ensuring the problem's strong convexity. The implementation provides methods for evaluating the function value (`fun`), the gradient (`grad`), as well as their stochastic counterparts that operate on individual samples (`f_i`, `grad_i`). The Hessian computation (`hess`) has been explicitly implemented myself while it doesn't exist.

Fixed α with varied $|\mathcal{S}_k|$ & $|\mathcal{S}_k^H|$

Fixed parameters for our experiment were set to $\alpha = 0.01$, $\theta = 0.5$, and $c = 0.0001$ across all runs. In theory, we would anticipate that as $|\mathcal{S}_k| = |\mathcal{S}_k^H|$ increases, the variance in the stochastic estimations of the gradient and Hessian should decrease, yielding a smoother and more consistent convergence path toward the minimum of the objective function. This expectation is grounded in the understanding that larger samples better approximate the full dataset, reducing the noise in the direction and magnitude of the update steps.

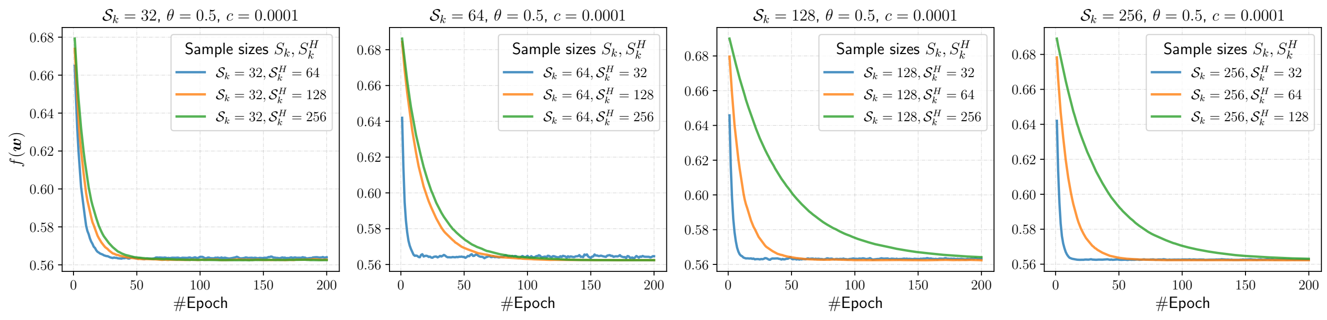


🟨 **Figure 5**: Subsampling Newton with $|\mathcal{S}_k| = |\mathcal{S}_k^H|$

When $|\mathcal{S}_k| = |\mathcal{S}_k^H| = 1$, the updates are highly stochastic, leading to significant oscillations in the function value across epochs, as evidenced in the first plot. As the sample size grows, the convergence trajectory becomes less erratic and more stable, with the variance in function value reductions diminishing with each increment in sample size. This trend continues up to $|\mathcal{S}_k| = |\mathcal{S}_k^H| = 256$, where the method almost mirrors the behavior of a deterministic Newton method, showcasing a steady decrease in function value and a smoother convergence curve.

In subsampling optimization methods, if the gradient subsample size $|\mathcal{S}_k|$ differs from the Hessian subsample size $|\mathcal{S}_k^H|$, we might expect a trade-off between the accuracy of the descent direction and the curvature estimation. Theoretical insights suggest that larger subsamples provide more precise estimates, potentially yielding smoother and quicker convergence.

Turning to the results where $|\mathcal{S}_k| \neq |\mathcal{S}_k^H|$, we observe a complex interplay between the sizes of gradient and Hessian subsamples. It appears that having a larger subsample for either the gradient or the Hessian does not uniformly lead to faster convergence; instead, the rate and stability of convergence are influenced by how these sample sizes are paired. Specifically, if the Hessian's subsample size is too small compared to the gradient's, we may not gain much from more accurate gradient estimates due to poor curvature approximation. Conversely, a very accurate Hessian estimate with a noisy gradient can also lead to erratic updates. The series of plots provided, corresponding to various combinations of $|\mathcal{S}_k|$ and $|\mathcal{S}_k^H|$, show a nuanced picture: certain combinations like $|\mathcal{S}_k| = 256$ and $|\mathcal{S}_k^H| = 64$ or $|\mathcal{S}_k| = 64$ and $|\mathcal{S}_k^H| = 256$ do not necessarily result in a convergence path that is better than when both sample sizes are equal but smaller.

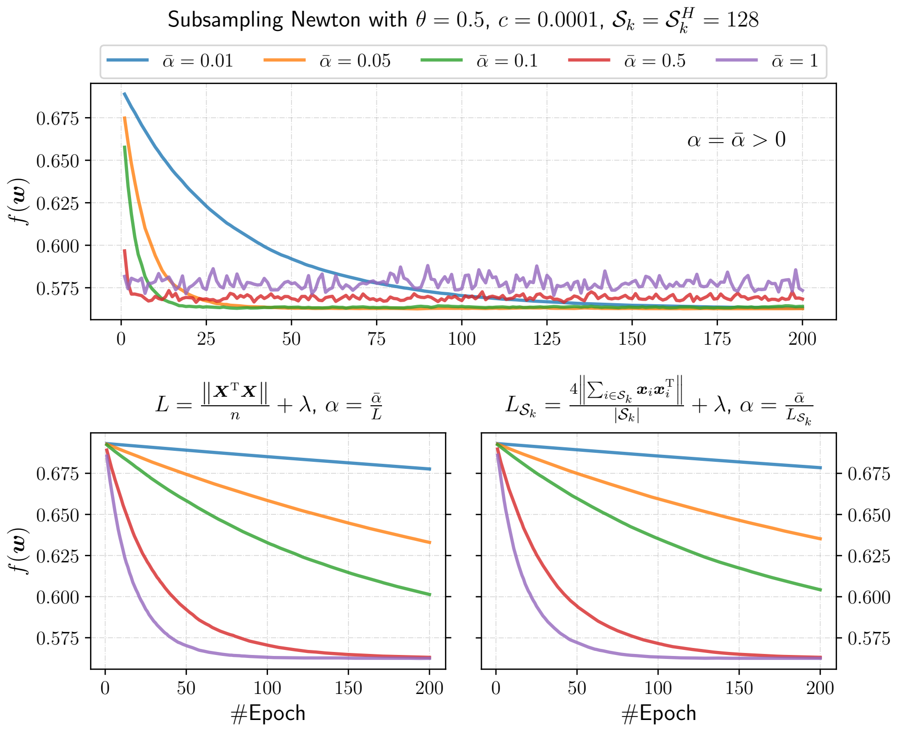


🟨 **Figure 6**: Subsampling optimization methods with $|\mathcal{S}_k| \neq |\mathcal{S}_k^H|$

Fixed $|\mathcal{S}_k|$ and $|\mathcal{S}_k^H|$

The experiment's setup is designed to evaluate the impact of different step sizes on the convergence of the subsampling Newton method, using a fixed batch size of $|\mathcal{S}_k| = |\mathcal{S}_k^H| = 128$. Three step size strategies were compared a constant $\alpha_k = \bar{\alpha} > 0$, $\alpha_k = \frac{\bar{\alpha}}{L}$ across all batches, with L being the global Lipschitz constant of ∇f. On the other ahnd, a batch-specific step size $\alpha_k = \frac{\bar{\alpha}}{L_{\mathcal{S}k}}$, where $L_{\mathcal{S}_k}$ represents the Lipschitz constant of $\nabla f \mathcal{S}_k$.

My expectation is that the use of $L_{\mathcal{S}_k}$ in the step size adjustment accounts for the local curvature of the loss surface, potentially providing a more tailored and efficient update at each iteration. Conversely, using the global L could be less responsive to the problem's local structure but ensures consistent step sizes across epochs. The global Lipschitz constant L was computed over the entire dataset, while $L_{\mathcal{S}_k}$ was calculated batch-wise. Different values of $\bar{\alpha}$ were tested to observe their influence under each step size rule.



🟨 **Figure 7**: Subsampling optimization methods with $\alpha_k = \bar{\alpha} > 0$, $\alpha_k = \frac{\bar{\alpha}}{L}$, $\alpha_k = \frac{\bar{\alpha}}{L_{\mathcal{S}k}}$

The observed outcomes, as depicted in figure 7, suggest that the choice of α_k significantly affects the convergence profile. For the global step size adjustment (upper plot), we notice convergence behavior varies with $\bar{\alpha}$, with smaller values leading to more stable but potentially slower convergence. On the other hand, the batch-specific adjustment or global L-Lipchitz (lower plot) demonstrates a more nuanced behavior, where larger $\bar{\alpha}$ does not necessarily accelerate convergence, my attention is about the trade-off between step size and the accuracy of the local curvature approximation.

It implies that employing $L_{\mathcal{S}_k}$ offers a flexible approach to handle varying local geometries of the loss surface across different batches, which could be advantageous for complex landscapes. However, it also introduces variability in step sizes that may lead to non-monotonic convergence patterns, especially for larger $\bar{\alpha}$ values. This can be contrasted with the global Lipschitz constant strategy, which seems to provide a more predictable, albeit potentially conservative, convergence trajectory. Conversely, a step size determined by the global Lipschitz constant L might result in a less dynamic but more stable convergence across epochs. While L reflects the average smoothness of the entire dataset, $L_{\mathcal{S}_k}$ is recalculated for each batch to account for local smoothness.

The adaptive approach using $L_{\mathcal{S}_k}$ did not consistently outperform the global Lipschitz strategy, as seen in the lower plot. Larger values of $\bar{\alpha}$ under $L_{\mathcal{S}_k}$ did not significant expedite convergence, indicating that the theoretical benefit of adapting to local curvature may be offset by increased variability and noise in the subsampled gradients and Hessians. The global Lipschitz strategy, depicted in the upper plot, exhibited a more stable and predictable path toward convergence, albeit potentially more conservative due to constant step sizes not reacting to local curvature changes. These observations suggest that the practical efficacy of employing an adaptive step size is contingent on the interplay between step size, batch variability, and the precision of curvature approximation, which may not align straightforwardly with theoretical expectations.

2.2 Stochastic Quasi-Newton methods

In stochastic optimization, sample size critically influences the trade-off between computational efficiency and convergence reliability. Theoretically, larger batch sizes reduce gradient and Hessian estimate variances, promoting smoother convergence trajectories. This trend is evident in our results [Figure 8] for both Stochastic BFGS and L-BFGS methods, where increased sample sizes led to lower variability in function values across epochs.

Stochastic L-BFGS, with its limited memory approach, is expected to be particularly efficient for large-scale problems. Our empirical observations support this, as L-BFGS consistently exhibited stable behavior even with smaller batch sizes. This indicates the method's adeptness at leveraging curvature information from recent iterations to accelerate convergence, without necessitating large memory requirements.

However, when evaluating the performance trade-offs, it's essential to consider the computational cost against potential gains in convergence speed. Our results suggest that while both methods benefit from larger samples, the advantage diminishes beyond a certain threshold, highlighting the importance of selecting appropriate batch sizes in practice. This balance is a crucial determinant of algorithm performance, alongside problem-specific characteristics such as dimensionality and convexity.

Comparatively, the Stochastic BFGS and L-BFGS methods introduce variance in each epoch due to the subsampling process, which can lead to less smooth convergence curves compared to their deterministic counterparts. This variance can be observed in the function value plots across epochs, where stochastic methods exhibit more fluctuations. Conversely, deterministic BFGS and L-BFGS tend to have more predictable descent paths due to the use of full dataset information in each iteration.

The classical BFGS method incurs a time complexity of $\mathcal{O}(nd)$ for gradient computations and $\mathcal{O}(d^2)$ for

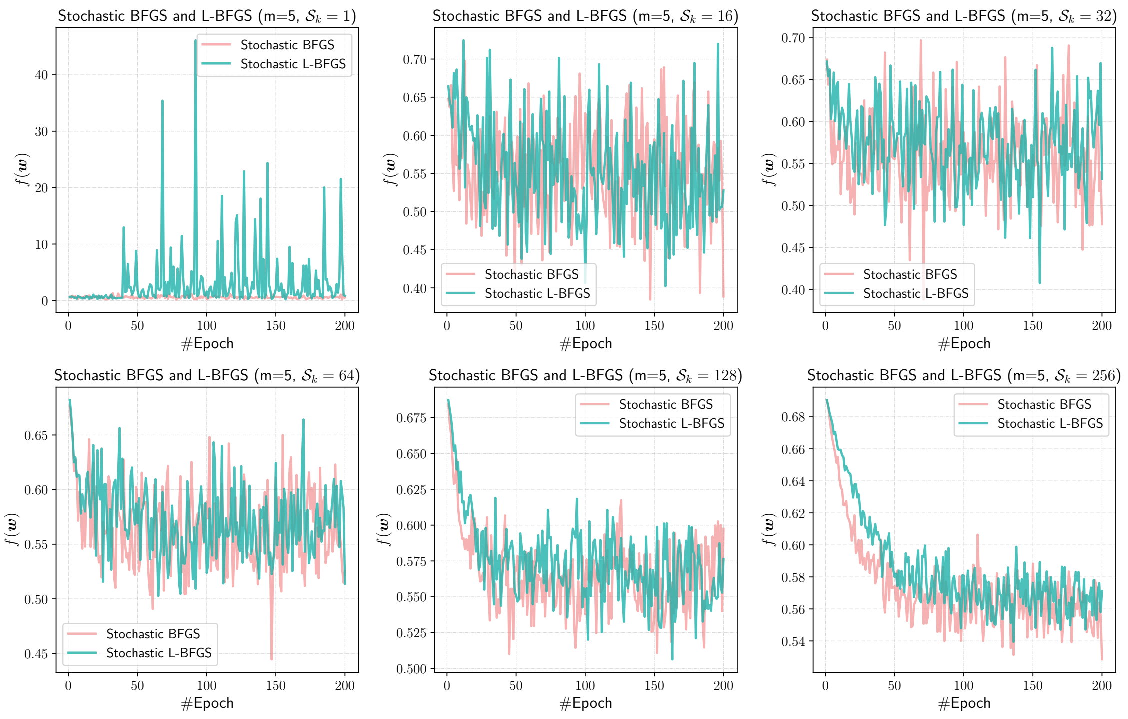


Figure 8: Stochastic BFGS & L-BFGS with several batch sizes

updating the inverse Hessian approximation, with a space complexity of $\mathcal{O}(d^2)$. L-BFGS, which avoids explicit Hessian calculations, has a time complexity of $\mathcal{O}(md)$ where m is the number of corrections stored, leading to a space complexity of $\mathcal{O}(md)$. Stochastic versions of BFGS and L-BFGS reduce the per-iteration time complexity to $\mathcal{O}(m'd)$ by using subsamples of the dataset, but they require more iterations to converge. Despite potential increases in the number of iterations, stochastic methods are especially beneficial for large datasets due to their lower computational burden per iteration, thus offering a better trade-off between computational efficiency and convergence quality.

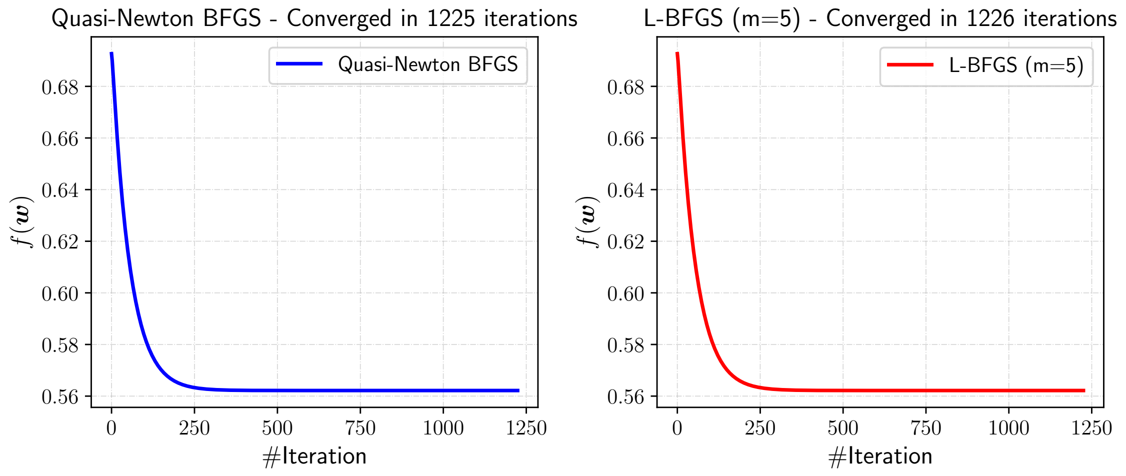


Figure 9: Classical BFGS & L-BFGS with pre-defined $\alpha = 0.01$

An iteration in classical methods encapsulates an update with the full dataset, while an epoch in stochastic methods represents a full pass through the dataset across multiple mini-batch updates. Classical methods typically showcase a smoother convergence trajectory due to the utilization of comprehensive data, whereas stochastic methods may exhibit a higher variance in the objective function across epochs.

3 Binary Classification

In our binary classification setup using the `mushroom` dataset, the objective function's gradient and Hessian are key to our optimization problem. The problem is defined over the dataset $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, with $\boldsymbol{x}_i \in \mathbb{R}^d$ and $y_i \in \{0,1\}$, aimed at minimizing the function:

$$g(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} g_i(\boldsymbol{w}), \quad \text{where} \quad g_i(\boldsymbol{w}) = \left(y_i - \frac{1}{1+\exp(-\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w})}\right)^2.$$

The gradient $\nabla g_i(\boldsymbol{w})$ and Hessian $\nabla^2 g_i(\boldsymbol{w})$ of each component g_i defined as:

$$\nabla g_i(\boldsymbol{w}) = -\frac{2\exp(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w})\left(\exp(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w})(y_i-1)+y_i\right)}{(1+\exp(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w}))^3}\boldsymbol{x}_i, \tag{13}$$

$$\nabla^2 g_i(\boldsymbol{w}) = \frac{2\exp(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w})\left(\exp(2\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w})(y_i-1)+2\exp(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w})-y_i\right)}{(1+\exp(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w}))^4}\boldsymbol{x}_i\boldsymbol{x}_i^{\mathrm{T}}. \tag{14}$$

I implemented a `Instance` class to manage setup tasks, including gradient and Hessian computations. During my experiments, I noticed that certain datasets, such as `mushrooms`, resulted in singular matrices. To address this issue, I added a regularization term in the Hessian (14) enhances the numerical stability of the optimization process. We have new Hessian $\nabla^2 g_i(\boldsymbol{w})$:

$$\nabla^2 g_i(\boldsymbol{w}) = \frac{2\exp(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w})\left(\exp(2\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w})(y_i-1)+2\exp(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w})-y_i\right)}{(1+\exp(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w}))^4}\boldsymbol{x}_i\boldsymbol{x}_i^{\mathrm{T}} + 2\lambda\mathbf{I}, \tag{15}$$

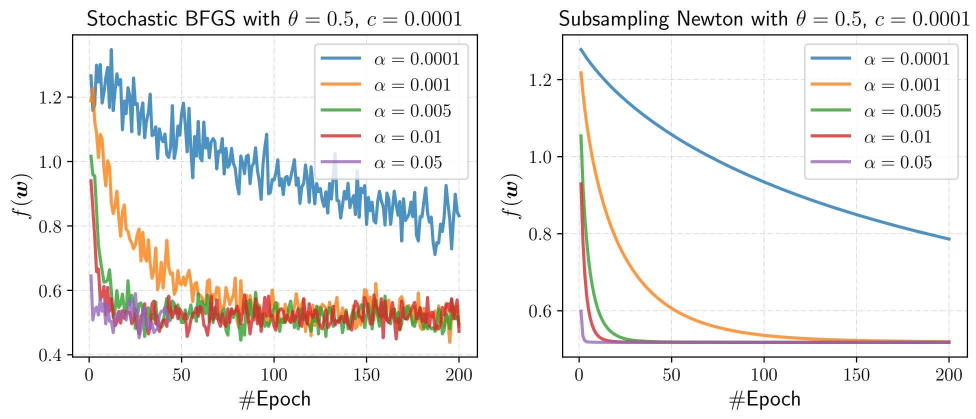


Figure 10: Convergence of the objective function $f(\boldsymbol{w})$ over 200 epochs for multiple learning rates using Subsampling Newton and Stochastic BFGS methods.

Upon scrutinizing the learning rates, $\alpha = 0.001$ was identified as optimally representative for juxtaposing the Subsampling Newton and Stochastic BFGS algorithms. While both algorithms demonstrated similar trajectories in minimizing the objective function, it is by its nature the Subsampling Newton method provided more stability throughout the 200 epochs. In terms of execution time, however, Stochastic BFGS showed a clear advantage, completing the process significantly faster than its counterpart. This positions Stochastic BFGS as the preferable algorithm in environments where time constraints are critical, despite the Subsampling Newton method's edge in maintaining stability during convergence. Ultimately, the choice of algorithm hinges on whether the application demands rapid execution or favors stability in the optimization process.

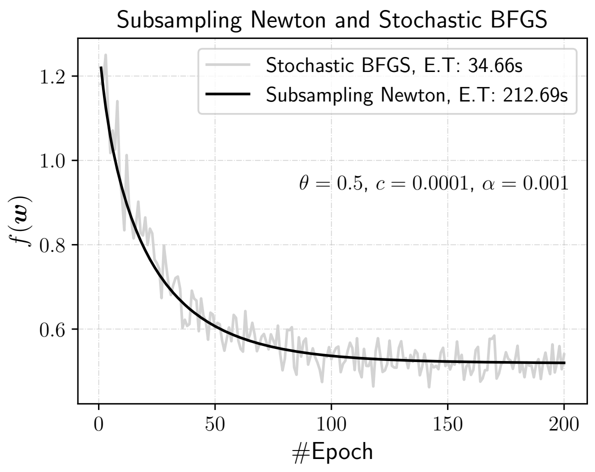


Figure 11: Comparison of between Subsampling Newton and Stochastic BFGS

4 Testing Loss vs. Training Loss

For this task, we evaluate the performance of the algorithms on the A9A dataset, we first split the data into training and test sets using a 70% to 30% ratio.

For both training and testing phases, `Instance` class were created to encapsulate the datasets, providing a structure for calculating individual losses and facilitating batch operations. The models' coefficients were initialized as zero vectors and updated using Stochastic BFGS and Subsampling Newton methods across 100 epochs, with a sample size of 256 and a learning rate of 0.001.

For the A9A dataset, after running the optimization algorithms, we assess the models' generalization using the average loss on both training and testing data. Mathematically, the training loss is computed as:

$$\text{Train Loss} = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} L(y_i, f(\boldsymbol{x}_i; \boldsymbol{w})) \quad \text{Test Loss} = \frac{1}{N_{\text{test}}} \sum_{j=1}^{N_{\text{test}}} L(y_j, f(\boldsymbol{x}_j; \boldsymbol{w}))$$

A9A Dataset	**Subsampling Newton**	**Stochastic BFGS**
Train Loss	0.9535	0.9475
Test Loss	0.9551	0.9523

These results, alongside the plotted convergence in Figure 12, suggest that while both algorithms effectively minimized the loss function, Stochastic BFGS achieved slightly better performance with less computational time, making it a preferable choice for efficiency without a significant compromise in model accuracy.

Metric	Subsampling Newton	Stochastic BFGS
Accuracy	0.81	0.82
F1 Score	0.37	0.45
Precision	0.88	0.86
Recall	0.24	0.31

Table 3: Performance Metrics for the A9A Dataset

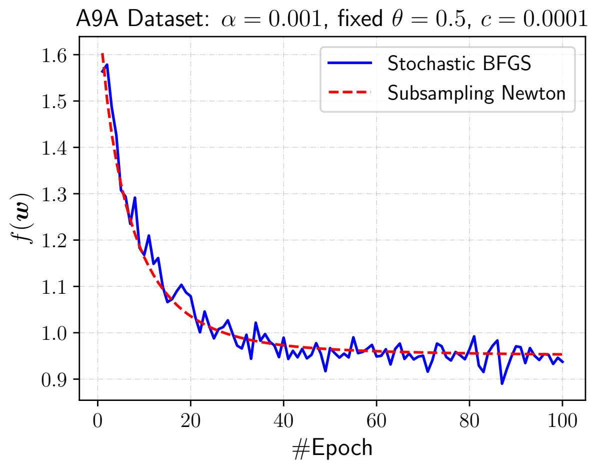


Figure 12: Training loss over 100 epochs for the A9A dataset using the Subsampling Newton and Stochastic BFGS methods with a learning rate of $\alpha = 0.001$, fixed $\theta = 0.5$, and $c = 0.0001$.

A Pseudo-code

Algorithm 1: Newton's Method

```
Input: $f : \mathbb{R}^d \to \mathbb{R}$, $\nabla f$, $\nabla^2 f$, $\boldsymbol{w}_0 \in \mathbb{R}^d$, tolerance $\epsilon$, maximum iterations $M$
Output: Approximate solution $\boldsymbol{w}^*$
1  $\boldsymbol{w} \leftarrow \boldsymbol{w}_0$                      // Initialize parameter vector
2  $k \leftarrow 0$                                  // Initialize iteration counter
3  while $k < M$ do
4      $g \leftarrow \nabla f(\boldsymbol{w})$                   // Compute gradient at $\boldsymbol{w}$
5      $H \leftarrow \nabla^2 f(\boldsymbol{w})$                 // Compute Hessian at $\boldsymbol{w}$
6      if $\|g\| < \epsilon$ then
7          break          // Terminate if gradient norm is below threshold
8      $\Delta \boldsymbol{w} \leftarrow -H^{-1} g$              // Calculate Newton step direction
9      $\boldsymbol{w} \leftarrow \boldsymbol{w} + \Delta \boldsymbol{w}$                // Update parameter vector
10     $k \leftarrow k + 1$                          // Increment iteration counter
```

Algorithm 2: Armijo Line Search

```
Input: $\boldsymbol{w}$, search direction $\boldsymbol{d}$, objective function $f$, gradient $\nabla f$, control parameter $c \in (0, \frac{1}{2})$,
       reduction factor $\theta \in (0, 1)$
Output: Step size $\alpha$
$\alpha \leftarrow 1$                                              // Initialize step size
$g \leftarrow \nabla f(\boldsymbol{w})$                            // Compute gradient at $\boldsymbol{w}$
while True do
    $\boldsymbol{w}_{\text{next}} \leftarrow \boldsymbol{w} + \alpha \boldsymbol{d}$     // Calculate potential next point
    if $f(\boldsymbol{w}_{\texttt{next}}) < f(\boldsymbol{w}) + c\alpha g^T \boldsymbol{d}$ then
        break                                                      // Check Armijo condition
    $\alpha \leftarrow \theta\alpha$                               // Reduce step size
return $\alpha$                                                    // Return step size
```

Algorithm 3: Globalized Newton's Method with Armijo Line Search

```
Input: $f : \mathbb{R}^d \to \mathbb{R}$, $\nabla f$, $\nabla^2 f$, initial point $\boldsymbol{w}_0 \in \mathbb{R}^d$, control parameter $c \in (0, \frac{1}{2})$, reduction factor
       $\theta \in (0, 1)$, tolerance $\epsilon$, maximum iterations $M$
Output: Approximate solution $\boldsymbol{w}^*$
Initialize $\boldsymbol{w} \leftarrow \boldsymbol{w}_0$, $k \leftarrow 0$, $\lambda_k \leftarrow 0$     // Setup initial values and counter
while $k < M$ do
    $g \leftarrow \nabla f(\boldsymbol{w})$                        // Compute gradient at current point
    $H \leftarrow \nabla^2 f(\boldsymbol{w})$                      // Compute Hessian at current point
    $\lambda_k \leftarrow 2 \max\{-\min(\text{eig}(H)), 10^{-10}\}$   // Adjust for positive definiteness
    $H \leftarrow H + \lambda_k I$                                 // Make Hessian positive definite
    $\boldsymbol{d}_k \leftarrow -H^{-1} g$                        // Determine Newton-type direction
    $\alpha_k \leftarrow 1$                                        // Initialize step size
    $j_k \leftarrow 0$                                             // Initialize line search counter
    while $f(\boldsymbol{w} + \theta^{j_k} \boldsymbol{d}_k) \geq f(\boldsymbol{w}) + c\theta^{j_k} g^T \boldsymbol{d}_k$ do
        $j_k \leftarrow j_k + 1$                                   // Increment line search counter
    $\alpha_k \leftarrow \theta^{j_k}$                             // Determine step size
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_k \boldsymbol{d}_k$   // Update parameter vector
    $k \leftarrow k + 1$                                           // Increment iteration counter
```

Algorithm 4: Quasi-Newton BFGS Optimizer with Armijo Line Search

```
Input: $f : \mathbb{R}^d \to \mathbb{R}$, $\nabla f$, initial point $\boldsymbol{w}_0 \in \mathbb{R}^d$, control parameter $c \in (0, \frac{1}{2})$, reduction factor
       $\theta \in (0, 1)$, tolerance $\epsilon$, maximum iterations $M$
Output: Approximate solution $\boldsymbol{w}^*$
1  $\boldsymbol{w} \leftarrow \boldsymbol{w}_0$, $H \leftarrow I_d$                 // Initialize $\boldsymbol{w}$ and Hessian approximation
2  $k \leftarrow 0$                                      // Initialize iteration counter
3  while $k < M$ do
4      $g \leftarrow \nabla f(\boldsymbol{w})$                         // Compute gradient at current point
5      $\boldsymbol{d}_k \leftarrow -Hg$                              // Determine search direction
6      Choose $\alpha_k$ using line search          // Determine step size via Armijo rule
7      $\boldsymbol{s}_k \leftarrow \alpha_k \boldsymbol{d}_k$                                        // Step vector
8      $\boldsymbol{w}_{\text{next}} \leftarrow \boldsymbol{w} + \boldsymbol{s}_k$                         // Update parameter vector
9      $\boldsymbol{v}_k \leftarrow \nabla f(\boldsymbol{w}_{\text{next}}) - g$                  // Gradient difference vector
10     if $\boldsymbol{s}_k^T \boldsymbol{v}_k > 0$ then
11         Update $H$ using BFGS update     // Update Hessian approximation if s and v are
            positive correlated
12     $\boldsymbol{w} \leftarrow \boldsymbol{w}_{\text{next}}$                                // Update parameter vector
13     $k \leftarrow k + 1$                                // Increment iteration counter
```

Algorithm 5: Limited-memory BFGS (L-BFGS) Optimizer

```
Input: Objective function $f : \mathbb{R}^d \to \mathbb{R}$, gradient $\nabla f$, initial point $\boldsymbol{w}_0 \in \mathbb{R}^d$, memory size $m \in \mathbb{N}$,
       Armijo line search parameters $c \in (0, \frac{1}{2})$, $\theta \in (0, 1)$, maximum iterations $M$
Output: Approximate solution $\boldsymbol{w}^*$
1  Initialize $\boldsymbol{w} \leftarrow \boldsymbol{w}_0$, $k \leftarrow 0$
2  Initialize memory structures $\{\boldsymbol{s}_i\}$, $\{\boldsymbol{v}_i\}$ as empty lists
3  while $k < M$ do
4      $\boldsymbol{g}_k \leftarrow \nabla f(\boldsymbol{w})$                         // Compute gradient at current point
5      $\boldsymbol{d}_k \leftarrow$ ComputeSearchDirection($\boldsymbol{g}_k, \{\boldsymbol{s}_i\}, \{\boldsymbol{v}_i\}, m$)      // Determine L-BFGS direction
6      Choose $\alpha_k$ using line search          // Determine step size via Armijo rule
7      $\boldsymbol{s}_k \leftarrow \alpha_k \boldsymbol{d}_k$                                        // Step vector
8      $\boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{s}_k$                                // Update parameter vector
9      $\boldsymbol{g}_{\text{new}} \leftarrow \nabla f(\boldsymbol{w})$                                    // New gradient
10     $\boldsymbol{v}_k \leftarrow \boldsymbol{g}_{\text{new}} - \boldsymbol{g}_k$                            // Gradient difference
11     UpdateMemory($\{\boldsymbol{s}_i\}, \{\boldsymbol{v}_i\}, \boldsymbol{s}_k, \boldsymbol{v}_k, m$)                  // Update history
12     $k \leftarrow k + 1$
13 Function UpdateMemory($\{\boldsymbol{s}_i\}, \{\boldsymbol{v}_i\}, \boldsymbol{s}_k, \boldsymbol{v}_k, m$)
14     if length($\{\boldsymbol{s}_i\}$) = m then
15         Remove oldest $\boldsymbol{s}_i$ and $\boldsymbol{v}_i$
16     Add $\boldsymbol{s}_k$ and $\boldsymbol{v}_k$ to the memories
```

Algorithm 6: Subsampling Armijo Line Search

```
Input: $f_{\mathcal{S}_k} : \mathbb{R}^d \to \mathbb{R}$, gradient $\nabla f_{\mathcal{S}_k}$, iterate $\boldsymbol{w}_k \in \mathbb{R}^d$, $\boldsymbol{d}_k \in \mathbb{R}^d$, $\alpha \in \mathbb{R}_+$, $\theta \in (0,1)$, $c \in (0, \frac{1}{2})$
Output: Step size $\alpha_k$
1 $f_{\mathcal{S}_k} \leftarrow \frac{1}{|\mathcal{S}_k|} \sum_{i \in \mathcal{S}_k} f_i(\boldsymbol{w}_k)$        // Compute the subsampled function value at $\boldsymbol{w}_k$
2 $g_{\mathcal{S}_k} \leftarrow \nabla f_{\mathcal{S}_k}(\boldsymbol{w}_k)$        // Compute the subsampled gradient at $\boldsymbol{w}_k$
3 for $j_k = 0$ to 100 do
4     $\boldsymbol{w}_{\text{next}} \leftarrow \boldsymbol{w}_k + \theta^{j_k} \boldsymbol{d}_k$        // Update the iterate with the current step size
5     $f^{\text{new}}_{\mathcal{S}_k} \leftarrow \frac{1}{|\mathcal{S}_k|} \sum_{i \in \mathcal{S}_k} f_i(\boldsymbol{w}_{\text{next}})$        // Compute the subsampled function value at $\boldsymbol{w}_{\text{next}}$
6     if $f^{\texttt{new}}_{\mathcal{S}_k} < f_{\mathcal{S}_k} + c\theta^{j_k} g^T_{\mathcal{S}_k} \boldsymbol{d}_k$ then
7         break        // Armijo condition met
8 return $\theta^{j_k}$        // Return the step size
```

Algorithm 7: Subsampling Newton Method

```
Input: $f : \mathbb{R}^d \to \mathbb{R}$, $|\mathcal{S}_k|$, $|\mathcal{S}^H_k|$, $\boldsymbol{w}_0 \in \mathbb{R}^d$, regularization $\lambda$, epochs $E$, $\theta \in (0,1)$, $c \in (0, \frac{1}{2})$
Output: $\boldsymbol{w}^*$
1 $\boldsymbol{w} \leftarrow \boldsymbol{w}_0$        // Initialize iterate
2 for epoch ← 1 to $E$ do
3     for batch ← 1 to $\lceil n / \min(|\mathcal{S}_k|, |\mathcal{S}^H_k|) \rceil$ do
4         $\mathcal{S}_k \leftarrow \text{RandomSample}(\{1, \dots, n\}, |\mathcal{S}_k|)$        // Sample for gradient
5         $\mathcal{S}^H_k \leftarrow \text{RandomSample}(\{1, \dots, n\}, |\mathcal{S}^H_k|)$        // Sample for Hessian
6         $\nabla f_{\mathcal{S}_k} \leftarrow \frac{1}{|\mathcal{S}_k|} \sum_{i \in \mathcal{S}_k} \nabla f_i(\boldsymbol{w})$        // Compute subsampled gradient
7         $\nabla^2 f_{\mathcal{S}^H_k} \leftarrow \frac{1}{|\mathcal{S}^H_k|} \sum_{i \in \mathcal{S}^H_k} \nabla^2 f_i(\boldsymbol{w}) + \lambda \boldsymbol{I}_d$        // Compute subsampled Hessian
8         $\boldsymbol{d}_k \leftarrow -\nabla^2 f^{-1}_{\mathcal{S}^H_k} \nabla f_{\mathcal{S}_k}$        // Compute search direction
9         if use line search then
10            $\alpha_k \leftarrow \text{SubsamplingArmijoLineSearch}(\boldsymbol{w}, \boldsymbol{d}_k, f_{\mathcal{S}_k}, \nabla f_{\mathcal{S}_k}, \theta, c, \mathcal{S}_k)$
11        else
12            $\alpha_k \leftarrow \alpha$
13        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_k \boldsymbol{d}_k$        // Update iterate
14    Record epoch loss and update statistics
15 return $\boldsymbol{w}$        // Return the optimized weights
```

Algorithm 8: Stochastic BFGS Method

```
Input: $f : \mathbb{R}^d \to \mathbb{R}$, $|\mathcal{S}|$, $\boldsymbol{w}_0 \in \mathbb{R}^d$, epochs $E$, $\theta \in (0,1)$, $c \in (0, \frac{1}{2})$, step size $\alpha$, use line search
Output: $\boldsymbol{w}^*$
1  $\boldsymbol{w} \leftarrow \boldsymbol{w}_0$                                   // Initialize iterate
2  $\boldsymbol{H} \leftarrow \boldsymbol{I}_d$                                   // Initialize Hessian approximation
3  for epoch ← 1 to $E$ do
4      for batch ← 1 to $\lceil n/|\mathcal{S}| \rceil$ do
5          $\mathcal{S} \leftarrow \text{RandomSample}(\{1, \ldots, n\}, |\mathcal{S}|)$      // Sample indices for gradients
6          $\boldsymbol{g} \leftarrow \frac{1}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} \nabla f_i(\boldsymbol{w})$          // Compute subsampled gradient
7          $\boldsymbol{d} \leftarrow -\boldsymbol{H}\boldsymbol{g}$                          // Compute search direction
8          if use line search then
9              $\alpha \leftarrow \text{SubsamplingArmijoLineSearch}(\boldsymbol{w}, \boldsymbol{d}, \boldsymbol{g}, \mathcal{S}, \theta, c)$
10         $\boldsymbol{s} \leftarrow \alpha \boldsymbol{d}$                                   // Step taken
11         $\boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{s}$                              // Update iterate
12         $\boldsymbol{g}_{\text{new}} \leftarrow \frac{1}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} \nabla f_i(\boldsymbol{w})$     // Compute new subsampled gradient
13         $\boldsymbol{y} \leftarrow \boldsymbol{g}_{\text{new}} - \boldsymbol{g}$             // Compute difference in gradients
14         $\rho \leftarrow \frac{1}{\boldsymbol{y}^\top \boldsymbol{s}}$                         // Compute scaling factor
15         if $\boldsymbol{s}^\top \boldsymbol{y} > 0$ then
16             $\boldsymbol{H} \leftarrow (\boldsymbol{I}_d - \rho \boldsymbol{s}\boldsymbol{y}^\top)\boldsymbol{H}(\boldsymbol{I}_d - \rho \boldsymbol{y}\boldsymbol{s}^\top) + \rho \boldsymbol{s}\boldsymbol{s}^\top$   // Update Hessian approximation
17         Record epoch loss and update statistics
18 return $\boldsymbol{w}$                                   // Return the optimized weights
```

Algorithm 9: Stochastic L-BFGS Method

```
Input: $f : \mathbb{R}^d \to \mathbb{R}$, $|\mathcal{S}|$, $\boldsymbol{w}_0 \in \mathbb{R}^d$, memory size $m$, epochs $E$, $\theta \in (0,1)$, $c \in (0, \frac{1}{2})$, step size $\alpha$, use
       line search
Output: $\boldsymbol{w}^*$
1  $\boldsymbol{w} \leftarrow \boldsymbol{w}_0$                                   // Initialize iterate
2  Initialize L-BFGS history $\boldsymbol{S}_m$, $\boldsymbol{Y}_m$, $\boldsymbol{\rho}_m$             // Empty matrices
3  $k \leftarrow 0$                                        // Initialize iteration counter
4  for epoch ← 1 to $E$ do
5      for batch ← 1 to $\lceil n/|\mathcal{S}| \rceil$ do
6          $\mathcal{S} \leftarrow \text{RandomSample}(\{1, \ldots, n\}, |\mathcal{S}|)$      // Sample indices for gradients
7          $\boldsymbol{g} \leftarrow \frac{1}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} \nabla f_i(\boldsymbol{w})$          // Compute subsampled gradient
8          Compute L-BFGS direction $\boldsymbol{d}$ using $\boldsymbol{g}$, $\boldsymbol{S}_m$, $\boldsymbol{Y}_m$, $\boldsymbol{\rho}_m$, and $\boldsymbol{H}_0$   // Apply L-BFGS formula
9          if use line search then
10             $\alpha \leftarrow \text{SubsamplingArmijoLineSearch}(\boldsymbol{w}, \boldsymbol{d}, \boldsymbol{g}, \mathcal{S}, \theta, c)$
11         $\boldsymbol{s} \leftarrow \alpha \boldsymbol{d}$                                   // Step taken
12         $\boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{s}$                              // Update iterate
13         Update L-BFGS history $\boldsymbol{S}_m$, $\boldsymbol{Y}_m$, $\boldsymbol{\rho}_m$ with $\boldsymbol{s}$ and $\boldsymbol{y}$     // Store latest vectors
14         Record epoch loss and update statistics
15 return $\boldsymbol{w}$                                   // Return the optimized weights
```

References

[Armijo, 1966] Armijo, L. (1966). Minimization of functions having lipschitz continuous first partial derivatives. *Pacific Journal of Mathematics*, 16(1):1–3.

[Nocedal and Wright, 2006] Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer.