IEMS351 - Final Report

Xiaolong Liu & Kedong Chen

November 2023

1 Introduction

This report extends the work from Phase 1 of the project. Initially, we applied a nonlinear regression model to dataset1, employing both gradient descent and stochastic gradient descent for optimization. The performance of these two methods was then compared. Next, we fitted a logistic regression model to dataset2, using gradient descent with constant/line search and the Newton method with line search, and compared their respective performances. Finally, we trained neural networks on dataset3, experimenting with various batch sizes in gradient descent and different models, and evaluated their performance. For details on the setup of each dataset problem, please refer to the Appendix.

2 Dataset1 (nPentaneIsopentane)

2.1 Experiment Summary

This section focuses on key insights from our experiments, particularly on achieving a minimum loss of approximately 0.136. This loss corresponds to critical points like $[33.52, 2.24, 1.25, 5.23]$, $[32.26, -3.36, -1.96, -7.71]$, and others, indicating vital configurations where the loss function is minimized. A consistent pattern in the x_1 value across these points suggests an area for further exploration.

Setting up. It's important to note the relationship between gradient descent (GD) and stochastic gradient descent (SGD). GD can be considered a specific case of SGD with the batch size equal to the entire dataset. In our discussions, 'gradient descent' refers to this full-dataset GD.
For this dataset, we didn't separate it into training and testing subsets. This decision stems from our focus on a single prediction model and the limited size of our dataset (24 observations). Our evaluations are thus based on the full dataset without the need for performance comparisons typically done with a test set.
We employed the infinity norm for analysis, defined as $||x||_\infty = \max_i |x_i|$, where $|\cdot|$ denotes the absolute value.

2.1.1 Gradient descent (GD)

Result 1. First key finding from our series of experiments is derived from Experiment 3, as documented in our notebook. In this experiment, we explored the effects of using a step size of 0.05, a tolerance level of 0.001, and an initial point of $[1, 1, 1, 1]$. It successfully found a critical point at $[32.26, -3.36, -1.96, -7.71]$ after 42700 iterations, with a loss closely aligning with 0.136. To further analyze the process, we plotted the logarithmic norm of the gradient against the number of epochs. The resulting graph appeared typical and can be seen in the figure below. Similarly, we plotted the logarithmic loss function against the epoch count. This graph also exhibited expected trends, a closer inspection reveals an initial increase in loss, followed by a decline. This initial rise in loss may be attributed to the relatively large step size used in this experiment:

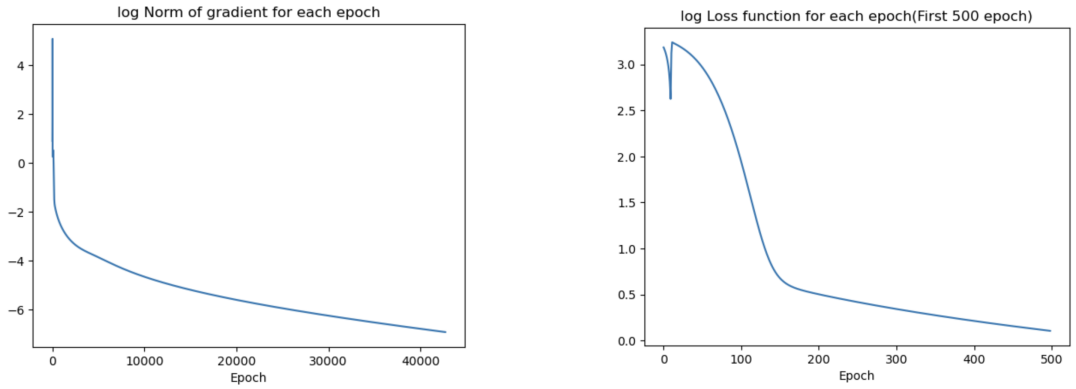


Figure 1: Logarithmic norm of gradient versus batch and logarithmic of loss function for Experiment 3

Result 2. In Experiment 4, with the initial parameters set at $[-1, -1, -1, -1]$ and a step size of 0.1, the process converged quickly, needing only 3074 iterations. However, it reached a stationary point at $[46.87, -2893.27, -646.54, -260.87]$, not the minimum of the loss function (value of 0.91). Remarkably, the parameters, especially x_2, were of very high magnitude. Plotting the gradient norm for the first 100 epochs revealed an exceptionally high value at the 17th epoch (around 28940), indicating a significant shift that likely caused the extreme parameter values. This key moment appears to have substantially altered the trajectory of the parameters from their initial values. The loss function exhibited a similar 'jump' in behavior, aligning with the observed gradient trends.

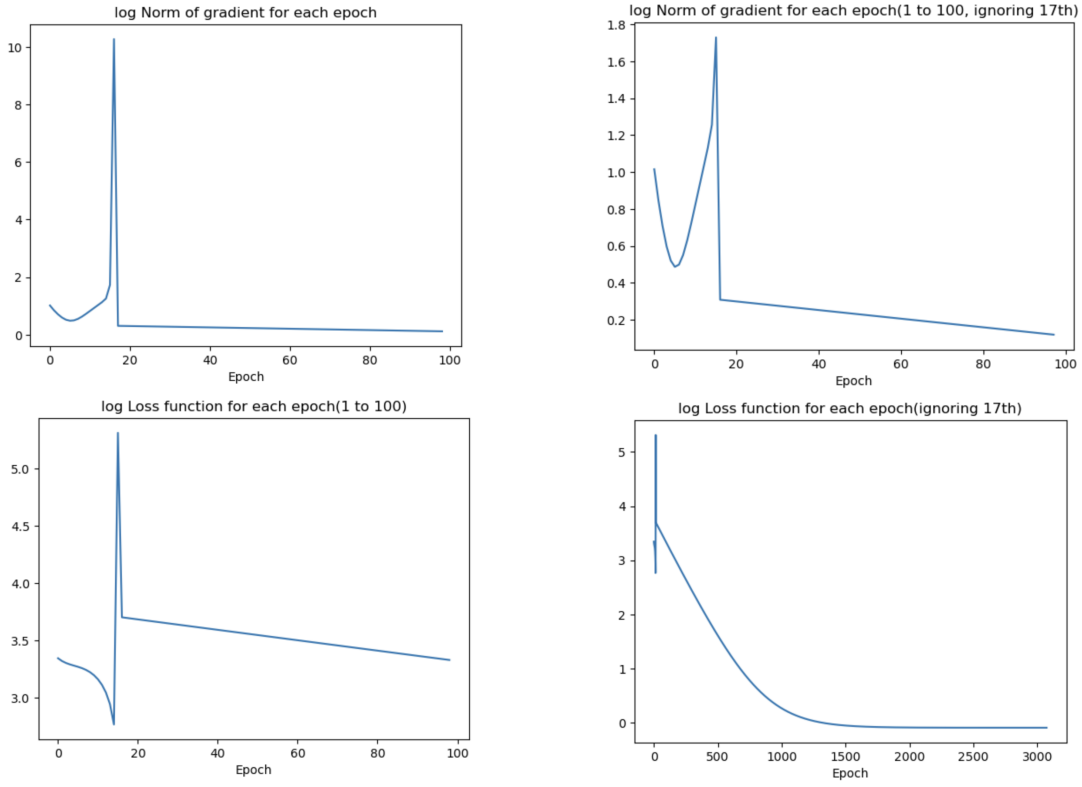


Figure 2: Logarithmic norm of gradient versus epoch and logarithmic loss function for Experiment 4

We hypothesize that the large step size, relative to the chosen starting point, may have caused the algorithm to overshoot a local minimum at the 17^{th} epoch, leading to a steep and unfavorable trajectory in the parameter space.

2.1.2 Stochastic Gradient Descent (SGD)

Result 3. We now turn our attention to Experiment 8 in the notebook. In this experiment, we initiated the process with a starting point of $[-1, -1, -1, -1]$. The algorithm converged after 9800 iterations, successfully identifying the minimum at the critical point $[32.59, 4.95, 2.857, 11.28]$.
The accompanying plots depict the norm of the gradient and the loss function against each batch. These graphs illustrate a characteristic feature of SGD: both the norm and the loss exhibit fluctuations, yet they demonstrate an overall descending trend, indicative of the converging process inherent to SGD.

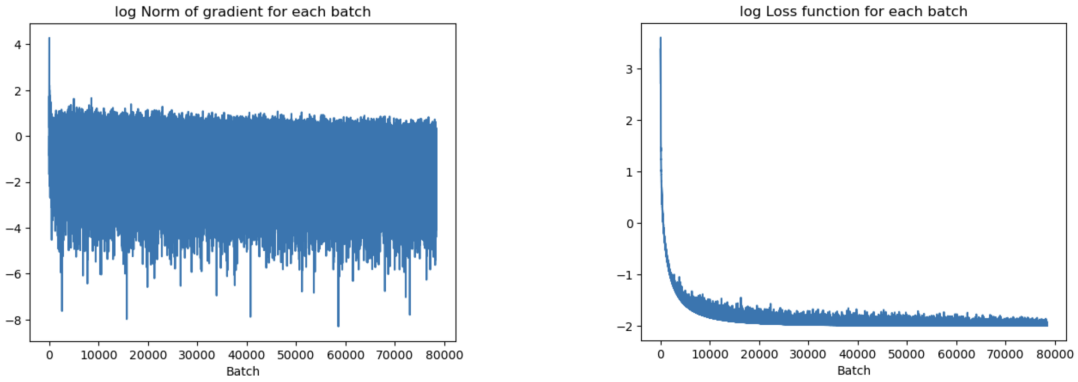


Figure 3: Logarithmic norm of gradient versus batch and logarithmic loss function for Experiment 8

Result 4. In Experiment 9, we adjusted the batch size to 6 and the step size to 0.01, resulting in an extensive training process of 59551 iterations. This experiment converged to a new local minimum at $[32.10, -7.02, -4.13, -16.04]$, with a loss similar to previous experiments. Like Experiment 4, there was a notably large gradient norm. However, the stochastic nature of the Gradient Descent (SGD) method was key here. It helped the algorithm to move past the initial large gradients, guiding it towards a path that led to a significantly low loss. This demonstrates SGD's ability to overcome complex gradients and find optimal or near-optimal solutions.

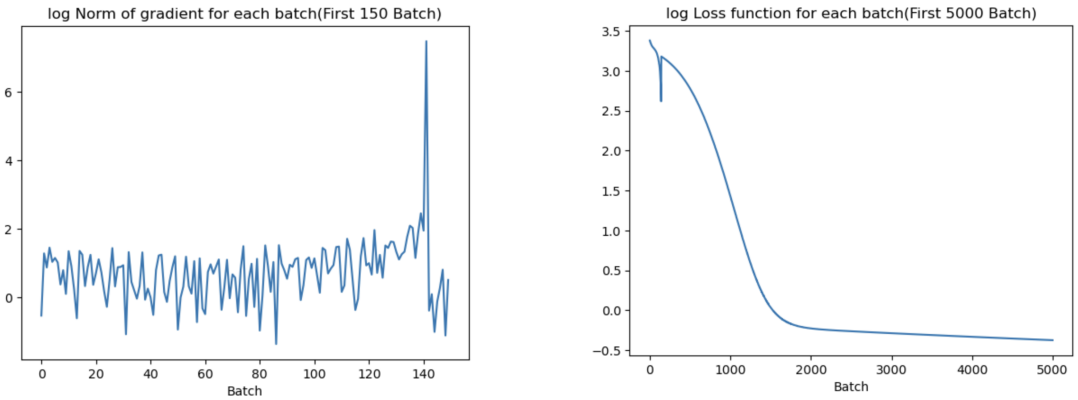


Figure 4: Norm of the gradient and the loss function against batch (first 5000 iterations) for Experiment 9

2.2 Observation and Comparison of Optimization Methods

Both Gradient Descent (GD) and Stochastic Gradient Descent (SGD) achieved comparable levels of loss when appropriately configured. The primary distinction between these methods lies in their convergence rates.

Convergence Rate Analysis. For this comparison, we selected the best-performing experiments for each method—Experiment 3 for GD and Experiment 8 for SGD. Analyzing the convergence rate at the epoch level (over the first 5000 epochs), SGD demonstrated a quicker convergence compared to GD. However, when comparing updates at the batch level, GD showed superior performance, characterized by greater stability and a faster rate of decrease.

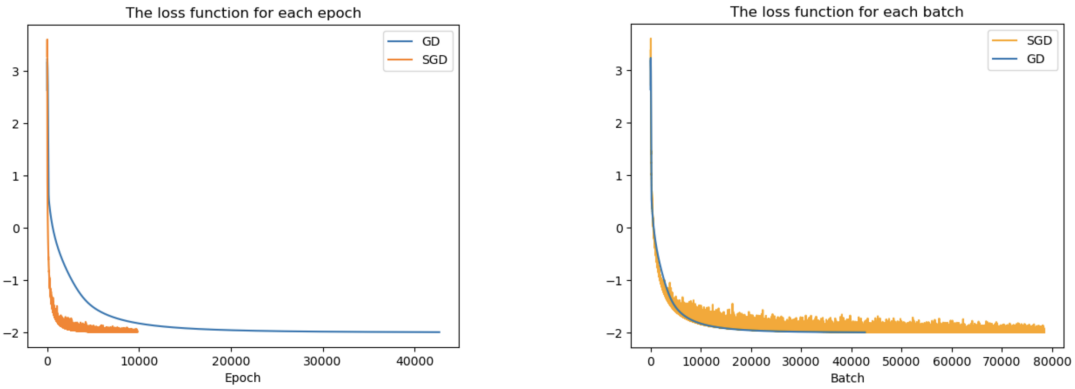


Figure 5: Comparison of loss function against epoch and batch

2.3 Discussion of Algorithm Performance on Dataset1

The comparative analysis reveals that Stochastic Gradient Descent reaches the desired tolerance level more rapidly than Gradient Descent, particularly when evaluated over epochs. However, at the batch level, Gradient Descent exhibits a more consistent and rapid decrease, indicating better stability and efficiency in each update.
This comparison suggests a trade-off: while GD provides more stability and requires higher computational resources, SGD, on the other hand, demonstrates quicker convergence but with less stability. Additionally, from the observations in Experiments 2 and 4, it is evident that SGD has the ability to escape from atypical positions and find critical points, a feat that GD struggles with. This highlights the adaptive nature of SGD in navigating complex optimization landscapes, making it a suitable choice in scenarios where rapid convergence is prioritized over computational efficiency.

3 Dataset2

3.1 Experiment Summary

In these experiments, although all our gradient descent methods surpassed the preset limits, many successfully approached the vicinity of the minimum loss. For Results 1 and 2, we employed starting points where all elements were set to zero.

Result 1. In our notebook, we present Experiments 1, 2, 3, and 4, where we selected a step size of 0.01. For gradient descent with a constant step size, we observed fluctuations in the gradient norm and corresponding loss, indicating that this step size is excessively large. Conversely, employing the same step size in gradient

descent with backtracking successfully established a trajectory for decreasing loss. Notably, the number of objective function evaluations in the backtracking approach was about seven times greater than that in the constant step size method, primarily due to the extensive evaluations required during the line search process.

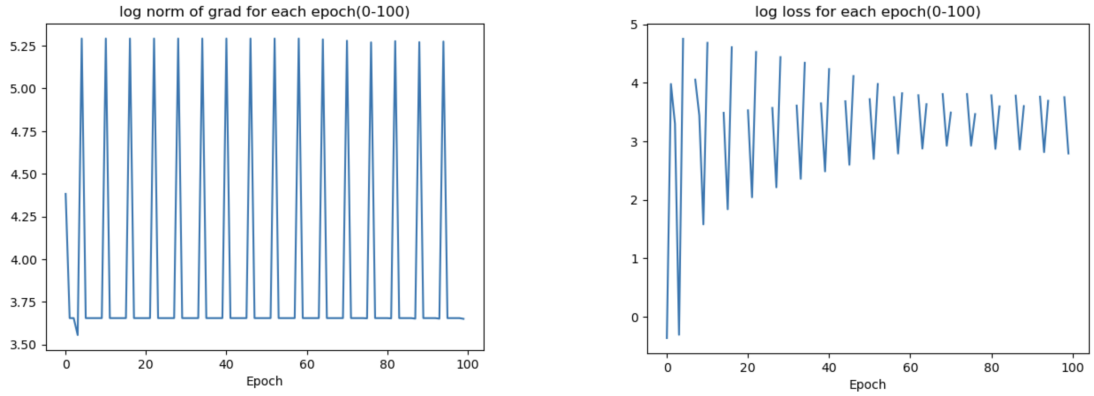


Figure 6: Gradient descent with constant, step size = 0.01

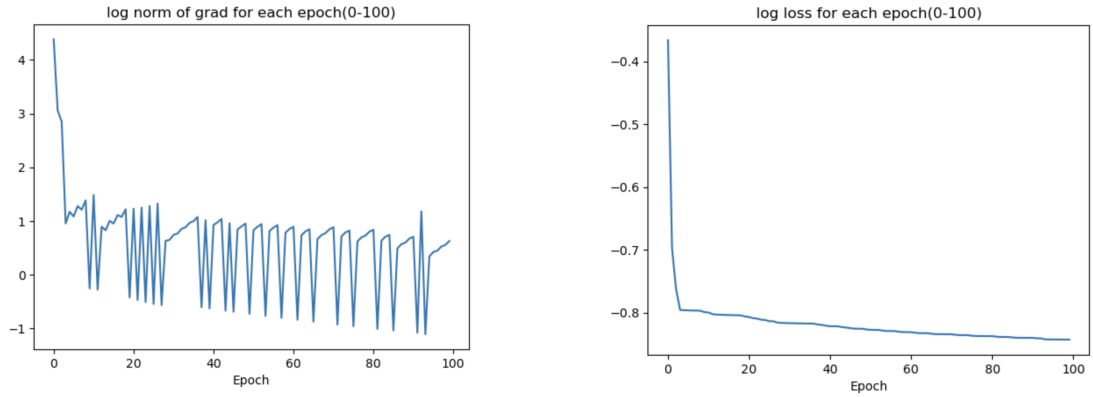


Figure 7: Gradient descent with backtracking, step size = 0.01

It was only when we reduced the step size to 0.0001 that the gradient descent with a constant step size began to decrease the loss as anticipated.

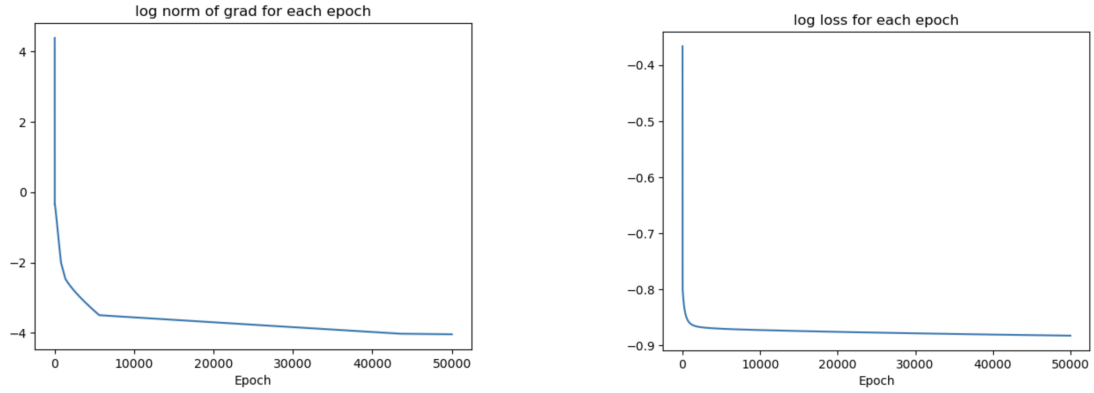


Figure 8: Gradient descent with constant, step size = 0.0001

Result 2. Next, we present Experiment 6 in the notebook, showcasing the Newton method. This approach demonstrates rapid convergence, requiring only 6 iterations. The swift decline in loss highlights the method's effectiveness and its quadratic convergence rate.

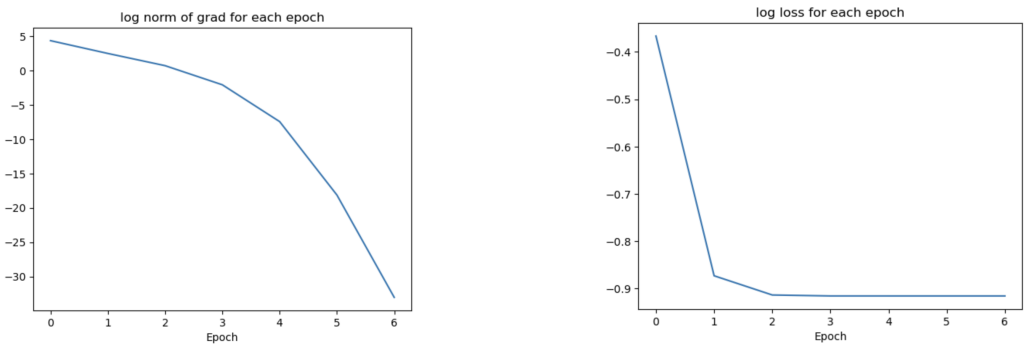


Figure 9: Newton method with backtracking

Result 3. In Experiments 7-13 of our notebook, we examine the impact of initial starting points, specifically set at starting point with all elements equals to 0.01 or 100. The Newton method often showed rapid divergence in these scenarios, particularly within the first 1 or 2 iterations. Even when adjusting alpha to 0.01, it only gradually minimized loss($27,044$ iterations and around $10,000$ total evaluations, we do not show its graphs which is very normal), highlighting its sensitivity to initial conditions. Gradient Descent (GD) with a constant step size could find the correct path, albeit requiring a much smaller step size. In contrast, GD with backtracking, while slower, consistently identified a trajectory towards minimizing loss.

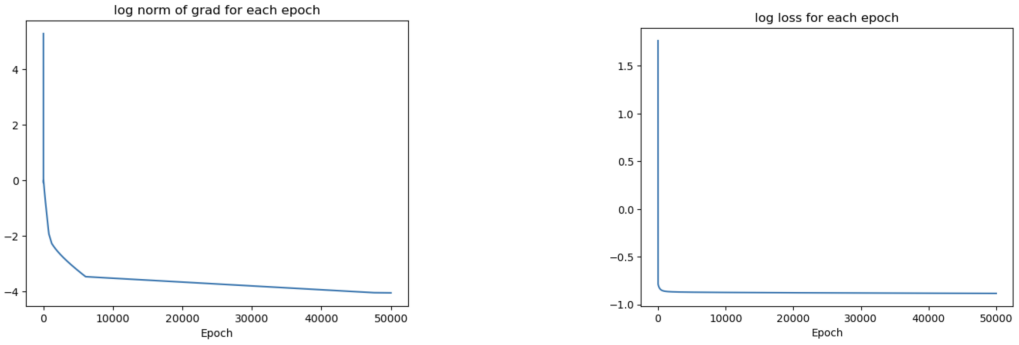


Figure 10: GD with constant, step size = 0.0001, starting point with all 0.01

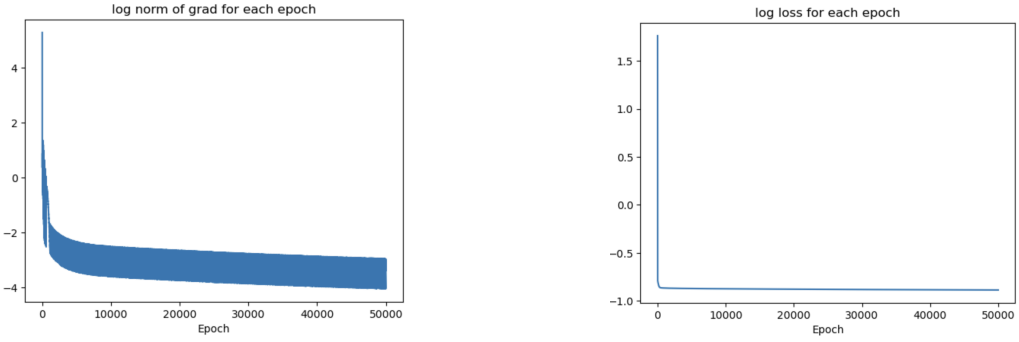
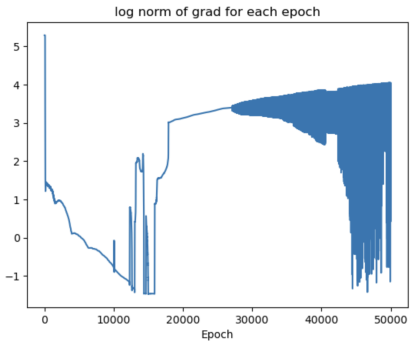


Figure 11: GD with backtracking, step size = 0.01, starting point with all 0.01



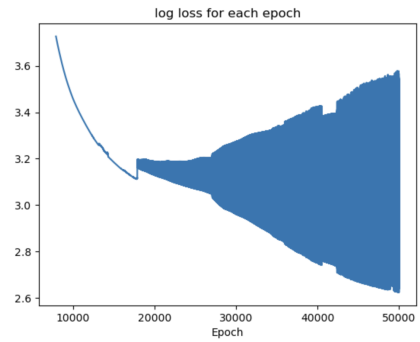
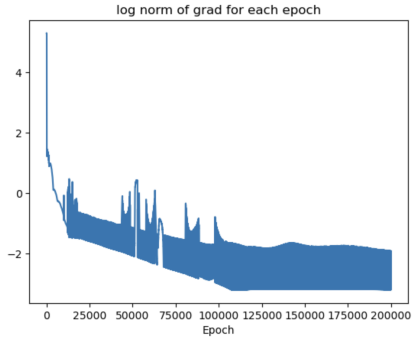


Figure 12: GD with constant, step size = 0.01, starting point with all 100



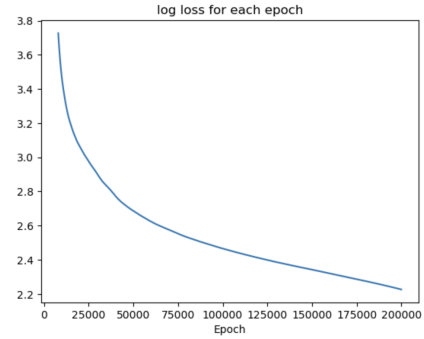


Figure 13: GD with backtracking, step size = 0.01, starting point with all 100

3.2 Discussion of Algorithm Performance on Dataset2

• In gradient descent with a constant step size, the function's sensitivity necessitates a small step size to avoid unstable oscillations in loss and gradient norm.
• While gradient descent with backtracking line search ensures a steady loss reduction, arbitrary step sizes can inflate computational requirements, as demonstrated by the varying function evaluations needed in different experiments.
• Both constant step size gradient descent and backtracking line search find satisfactory solutions but don't converge within set limits. In contrast, the Newton method with line search converges quickly, needing only 6 iterations.
• The Newton method faces challenges, particularly in computational cost due to the need for Hessian matrix calculations and the complexities of handling non-positive definite matrices.
• The Newton method demonstrates notable sensitivity to initial conditions, often resulting in divergence. On the other hand, gradient descent methods show greater adaptability to varied starting points, though their convergence is typically slower. Specifically, for Gradient Descent (GD) with a constant step size, careful selection of a smaller step size is crucial. In contrast, GD with backtracking doesn't necessitate such precise step size selection, but this flexibility may lead to an increased number of evaluations during line search.

4 Dataset3

4.1 Experiment Summary

We conducted experiments using four distinct neural network architectures, each tested with various batch sizes. Due to computational constraints, each experiment was limited to 20 epochs. The network models em-

ployed are as follows: Model 1: Comprises three fully connected layers. Model 2: Features two convolutional layers with corresponding pooling layers, followed by three fully connected layers. Model 3: Similar to Model 2 but with the ReLU activation function replaced by Tanh. Model 4: Consists of two convolutional layers, each paired with a pooling layer, then augmented with two batch normalization layers, and finally two fully connected layers. For comprehensive details on the architecture, parameters, and specific configurations of each model, please refer to the appendix.

Result 1. We experimented with batch sizes of 10, 100, and 1000, resulting in 5000, 500, and 50 updates per epoch, respectively. The training and testing accuracies for each model, corresponding to these batch sizes, are detailed in Tables 1 and 2.

Our results indicate that Model 4 outperforms the others in terms of accuracy on both training and testing datasets. This superior performance aligns with its significantly higher parameter count (2,122,378), suggesting a more complex model capacity. Across all models, training accuracies generally exceed testing accuracies, with the exception of Model 3 at a batch size of 1000. Notably, Model 4 achieved the highest testing accuracy of approximately 0.774, but its training accuracy was a remarkable 0.9997, raising concerns about potential overfitting.

While extending the number of epochs or employing more sophisticated network architectures might improve testing accuracy, computational limitations preclude such explorations in this study. Nevertheless, our findings reveal some intriguing patterns and insights, despite these constraints.

The results reveal a notable trend: as the batch size increases, there's a significant decrease in accuracy within the fixed epoch constraint. This observation is logical considering the update frequency of the model. For instance, with a batch size of 10, the model undergoes 5000 updates per epoch, totaling 100,000 updates across 20 epochs. Conversely, at a batch size of 1000, the update frequency drops to just 50 updates per epoch, or 1000 updates in total for 20 epochs. This variance in updates suggests that the data points are not highly similar to each other; otherwise, fewer updates might still have sufficed to achieve higher accuracy.

Comparing Model 3 with Model 2 reveals no significant benefit in substituting Tanh for ReLU as the activation function. Additionally, despite Model 1 having a substantial number of parameters (1,640,330), its improvement in accuracy is not particularly pronounced when compared to the other models.

Model	Parameters	Train Acc (BS=10)	Test Acc (BS=10)
Model 1	1640330	0.8758	0.5402
Model 2	62006	0.81096	0.6447
Model 3	62006	0.798	0.6233
Model 4	2122378	0.99974	0.774

Table 1: Model Parameters and Accuracies for Batch Size 10

Model	Train Acc (BS=100)	Test Acc (BS=100)	Train Acc (BS=1000)	Test Acc (BS=1000)
Model 1	0.61768	0.5352	0.37632	0.3758
Model 2	0.5714	0.5533	0.18814	0.1881
Model 3	0.576	0.5595	0.2932	0.2987
Model 4	0.94042	0.7487	0.67052	0.6441

Table 2: Model Parameters and Accuracies for Batch Sizes 100 and 1000

Result 2. We documented the training loss at half-epoch intervals and plotted the logarithmic values of these losses for each model(See Figure14). This analysis led to a consistent observation: smaller batch sizes correlate with lower losses given the same number of epochs. Notably, with a batch size of 10, the loss exhibited a zigzag pattern, unlike the smoother trends observed with larger batch sizes. This pattern suggests that using smaller batch sizes introduces more randomness into the training process.

The logarithmic losses for each model, standardized to the same batch size, are depicted in Figure 15. As anticipated, Model 4 registers the lowest and most rapidly decreasing loss among all models. Intriguingly, when the batch size is set to 1000, using Tanh as an activation function seems to result in a faster reduction in loss compared to ReLU. Model 1 outperforms Models 2 and 3, likely attributable to its higher parameter count. This trend underscores the general principle that models with more parameters tend to achieve lower training losses, though they also carry a higher risk of overfitting.

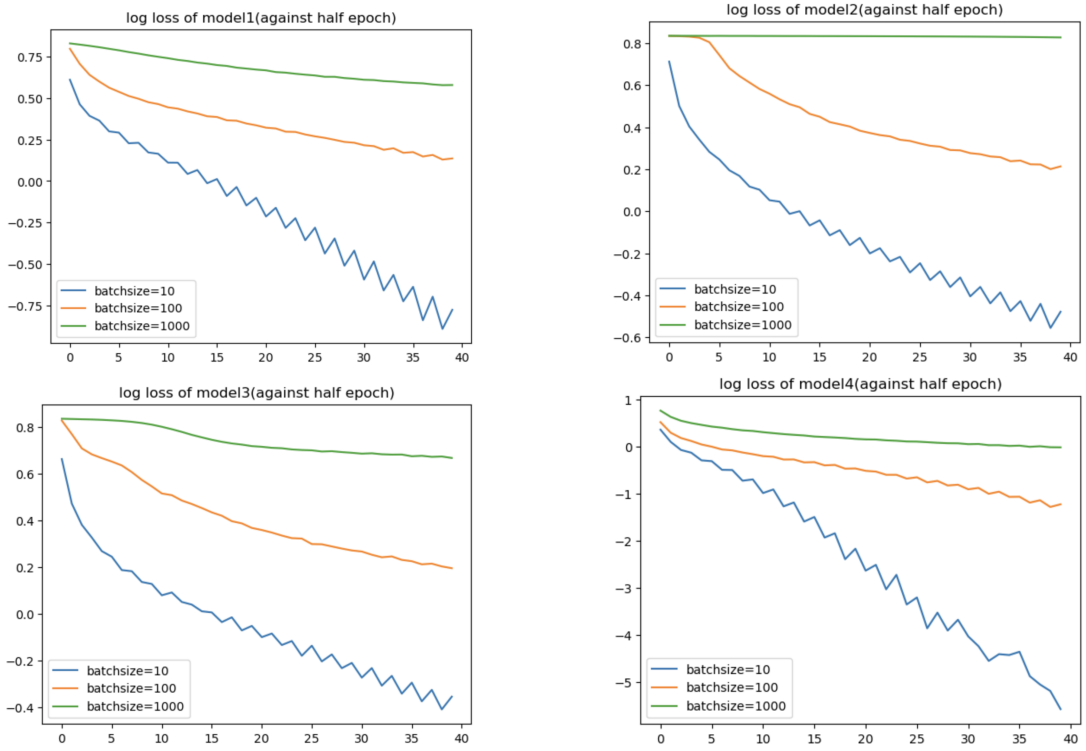


Figure 14: log loss for each model(against half epoch)

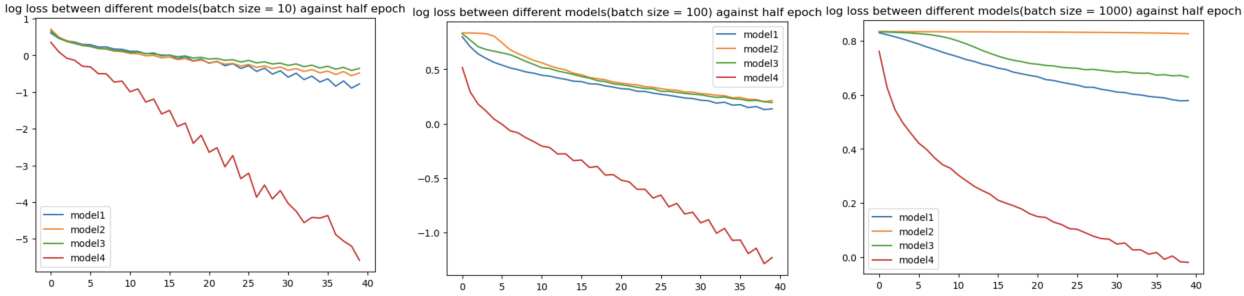


Figure 15: log loss comparison for each model

Result 3. The improved training outcomes observed with models having more parameters come at a cost of increased computational time. We examined this trade-off by plotting the logarithmic values of both the total and average per-update running times for each model configuration (see Figure 16). As expected, models with more parameters tend to have longer total running times and higher average times per update. Additionally, larger batch sizes generally result in shorter running times. However, an anomaly was observed

with Model 2, where the running time unexpectedly increased with larger batch sizes; this could potentially be attributed to memory constraints on my computer during training.
Specifically, when the batch size is set to 10, the use of Tanh as an activation function incurs more time than ReLU, which aligns with the computational simplicity of ReLU. Interestingly, for Model 2, an increase in batch size from 100 to 1000 led to longer running times, further suggesting memory limitations during training.
Additionally, there is a clear inverse relationship between batch size and total running time: as the batch size increases, the total running time, or equivalently, the running time per epoch, decreases.

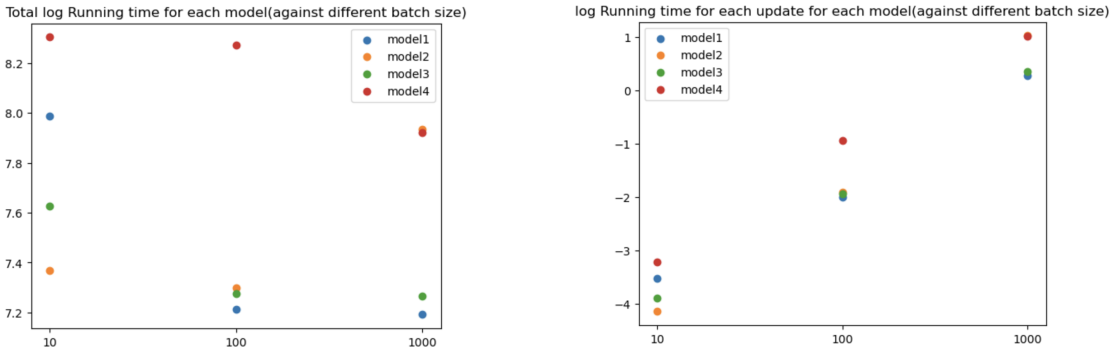


Figure 16: Total runing time and average runing time for each model

4.2 Discussion for Dataset3

• Using SGD for network training, smaller batch sizes result in more frequent updates per epoch. Consequently, while completing an epoch may take longer with smaller batches, each update is quicker. Importantly, with a fixed number of epochs, smaller batch sizes tend to yield lower training losses.
• In network design, the judicious selection of parameters plays a crucial role. A model with a well-considered number of parameters is more likely to achieve lower training losses and potentially higher testing accuracy. However, it is important to balance this with vigilance against overfitting and consideration of computational constraints.
• Generally, substituting Tanh for ReLU does not offer significant benefits and may increase computational costs. However, this additional cost tends to be less substantial with larger batch sizes.
• While our current testing accuracy falls short of expectations, there are promising avenues for improvement. Developing a more sophisticated model, potentially inspired by contemporary research in the field, could yield better results. Additionally, extending the number of training epochs may further enhance performance. However, these advancements hinge on access to more powerful computing resources, underscoring the need for an upgraded server to facilitate these enhancements.

5 Conclusion

Through extensive experimentation across various models and the development of numerous code iterations, we have gained profound insights into each algorithm, uncovering a range of intriguing findings. This report presents the culmination of our experiments and discussions on three datasets. We appreciate your attention and thank you for reading.

6 Appendix

6.1 Reference of Source

Dataset1:

- Programming Assignment 4: pa4.py(class: CsvDataSet())
- gradient_descent.py (function: get_options(), minimize())
- Programming Assignment 5: MNIST_template.ipynb(function: get_train_test_loader())

Dataset2:

- Programming Assignment 6: minimize_objective.py
- Programming assignment 6: pa6.py

Dataset3:

- Programming Assignment 5

6.2 Problem Setting for Dataset1

Prediction model. The goal for this dataset is to find the parameters x_1, x_2, x_3, x_4 of the following nonlinear model that predicts the reaction rate(denote it as y) as a function of z_1 (partial pressure of hydrogen), z_2 (partial pressure of isopentane), z_3 (partial pressure of n-pentane):

$$m(z, x) = \frac{x_1 x_3 (z_2 - z_3/1.632)}{1 + x_2 z_1 + x_3 z_2 + x_4 z_3},$$

here, $z = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix}$, $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$.

Nonlinear regression. First we define the loss function as

$$L(x) = \frac{1}{n} ||Y - M(x, Z)||_2^2,$$

here, $Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$ is a column vector including all labels, , $M(x, Z) = \begin{bmatrix} m(z^{(1)}, x) \\ m(z^{(2)}, x) \\ \vdots \\ m(z^{(n)}, x) \end{bmatrix}$, $z^{(i)} = \begin{bmatrix} z_1^{(i)} \\ z_2^{(i)} \\ z_3^{(i)} \end{bmatrix}$ is the data z for the ith observation. So we also can write $L(X) = \frac{1}{n} ||Y - M(Z, x)||_2^2 = \frac{1}{n} \sum_{i=1}^{n} [y_i - m(z^{(i)}, x)]^2$, but treat all these as matrices(vectors) will make the calculation much more easier. We can easily calculate the gradient of L:

$$\frac{\partial L}{\partial x_i} = -\frac{2}{n} (Y - M(Z, x))^T \frac{\partial M(Z, x)}{\partial x_i}, \quad \frac{\partial M(Z, x)}{\partial x_i} = \begin{bmatrix} \frac{\partial m(z^{(1)}, x)}{\partial x_i} \\ \frac{\partial m(z^{(2)}, x)}{\partial x_i} \\ \vdots \\ \frac{\partial m(z^{(n)}, x)}{\partial x_i} \end{bmatrix} \quad \text{and } \nabla L = \begin{bmatrix} \frac{\partial L}{\partial x_1} \\ \frac{\partial L}{\partial x_2} \\ \frac{\partial L}{\partial x_3} \\ \frac{\partial L}{\partial x_4} \end{bmatrix}$$

6.3 Problem Setting for Dataset2

The objective is to forecast the probability of a patient developing coronary heart disease (CHD) within the next 10 years, using various predictive factors. In this section, we employ a logistic regression model for CHD prediction and assess the effectiveness of different optimization algorithms. These include gradient descent with a constant step size, gradient descent with line search, and the Newton method with line search, enabling a comparison of their performance in this predictive task.

6.4 Problem Setting for Dataset3

The objective is to develop neural network classifiers using the CIFAR10 dataset, aimed at identifying the class of a given image.
The models we used:
•Model1: nn.Linear(3072, 512), nn.Linear(512, 128), nn.Linear(128, 10).
•Model2: nn.Conv2d(3, 6, 5), nn.MaxPool2d(2, 2), nn.Conv2d(6, 16, 5), nn.Linear(16 * 5 * 5, 120), nn.Linear(120, 84), nn.Linear(84, 10).
•Model3: Only change activation function to tanh() based on Model2.
•Model4: nn.Conv2d(3, 32, 3, padding=1), nn.BatchNorm2d(32), nn.Conv2d(32, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.Linear(64 * 8 * 8, 512), nn.Linear(512, 10). To see more details, please refer to Experiment for Dataset3.ipynb.