Neural Networks Assignment 3

UNM Computer Science

Alexander Bernal, Barath Kurapati, Nicholas Livingstone, Danny Metcalfe, Key Vollers

November 16, 2024

The code for this report and its usage can be found at this LoboGit repository.

1 Xavier Initialization

Xavier initialization is a weight initialization method designed to ensure that the weights are initialized in such a way that the variance of activations remains consistent across layers in a neural network.

1.1 Formula for Uniform Distribution

For a layer with n_{in} inputs (number of neurons in the previous layer) and n_{out} outputs (number of neurons in the current layer), the weights are sampled from a uniform distribution defined as:

$$W \sim U\left[-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right]$$

1.2 Implementation in the Project

Xavier initialization is implemented in the `Layer` class such that each layer automatically initializes its weights using this method during its creation.

1.3 Code for Xavier Initialization

The weights for each layer are initialized using the following code:

```
self.weights = np.random.uniform(
    -np.sqrt(6 / (input_size + output_size)),  # Lower bound of uniform distribution
     np.sqrt(6 / (input_size + output_size)),  # Upper bound of uniform distribution
     (output_size, input_size + 1)            # Size of the weights matrix
)
```

1.4 How It Works

1. **Bounds of Initialization:** The range for the weights is calculated as:

$$\pm\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}$$

   where:

   - n_{in}: Number of input neurons to the layer.
   - n_{out}: Number of output neurons in the layer.

2. **Bias Included:** The input size is incremented by 1 because the weights include the bias term, which is handled as part of the weights matrix.

3. **Weight Matrix Size:** The weights matrix has a shape of $(n_{\text{out}}, n_{\text{in}} + 1)$, accounting for the number of output neurons and the input size, including the bias term.

2 Newtons Method

Newton's Method is a way to update the weights in a neural network by using both the gradient (which shows the slope of the loss function) and the Hessian (which tells us how the slope is changing). It helps find the best weights faster by considering the shape of the loss function, not just the steepest direction.

2.1 Key Concepts

2.2 Gradient

- Represents the first derivative of the loss function.
- Indicates the direction of the steepest ascent/descent.

2.2.1 Hessian

- Represents the second derivative of the loss function.
- Captures curvature information, which helps in scaling and adjusting the gradient more effectively.

2.2.2 Weight Update in Newton's Method

- Incorporates the inverse of the Hessian matrix to scale the gradient before updating weights.
- Provides more accurate steps towards the optimal solution, especially in regions where the curvature varies.

2.3 Steps in the Function

2.3.1 Hessian Approximation

- Each layer computes its Hessian approximation, possibly regularized using λ_{reg} to prevent numerical instability.

2.3.2 Inverse Hessian

- Uses the pseudo-inverse (`np.linalg.pinv`) of the Hessian to handle cases where the Hessian is non-invertible.
- If even the pseudo-inverse fails, it falls back to using a diagonal regularized matrix ($\lambda_{\text{reg}} \times I$, where I is the identity matrix).

2.3.3 Weight Update

The weight update (`update_step`) is computed as:

$$\text{update\_step} = \text{learning\_rate} \times \text{inverse\_Hessian} \times \text{gradient}$$

The weights are updated by adding the computed step.

2.4 Train Function and Method Selection

2.4.1 Gradient Descent (`gd`)

- Adjusts weights using the gradient and a momentum factor (`friction`).

2.4.2 Newton's Method (`newton`)

- Calls `newtons_method` for weight updates, leveraging the curvature information for faster and more precise optimization.

3 Adams Method

The Adam method combines the benefits of momentum and RMSprop. It uses the first and second moments of gradients to adapt the learning rate for each weight.

3.1 Steps in the `Adam` Function

3.1.1 1. Gradient Clipping

- Gradients (`layer.grad_weights`) are clipped to the range $[-1, 1]$ to prevent instability caused by excessively large updates.

3.1.2 2. Moment Updates

- **First Moment (Mean of Gradients)**: The first moment (`layer.firstm`) is updated using the formula:

$$\text{firstm} \leftarrow \beta_1 \cdot \text{firstm} + (1 - \beta_1) \cdot \text{gradients}$$

- **Second Moment (Mean of Squared Gradients)**: The second moment (`layer.secondm`) is updated as:

$$\text{secondm} \leftarrow \beta_2 \cdot \text{secondm} + (1 - \beta_2) \cdot (\text{gradients})^2$$

  Here, β_1 and β_2 are the decay rates for the moments (hyperparameters).

3.1.3 3. Bias Correction

- To address bias introduced at the start of training, corrected moments are calculated:

$$\text{first\_corrected} = \frac{\text{firstm}}{1 - \beta_1^t}$$

$$\text{second\_corrected} = \frac{\text{secondm}}{1 - \beta_2^t}$$

  where t is the time step (incremented after every update).

3.1.4 4. Weight Update

- The weights are updated using the corrected moments:

$$\text{weights} \leftarrow \text{weights} - \frac{\text{learning\_rate}}{\sqrt{\text{second\_corrected}} + \epsilon} \cdot \text{first\_corrected}$$

  Here, ϵ is a small constant (e.g., 10^{-8}) added for numerical stability.

3.2 Benefits of Adam

- Adaptive learning rates make it effective for noisy gradients or sparse data.
- Combines the strengths of momentum and RMSprop for faster convergence.

4 Case Studies

We conducted four experiments on two datasets to evaluate the performance of various optimization techniques in a neural network with three neurons. The datasets, selected based on instructional requirements, included passenger data from the Titanic taken from the python library scikit-learn, where the objective was to predict survival based on age and sex, and a set of hadwritten digit images, used to test classification accuracy. Across all experiments, the loss function was utilized as the primary metric for accuracy assessment.

Our control experiment employed gradient descent with four distinct batch sizes. In the first experiment, we incorporated Xavier initialization into the gradient descent process to analyze its

effect. The second experiment replaced gradient descent with Nesterov Momentum, exploring the impact of varying friction weights instead of batch sizes. The third experiment utilized the Adam optimization algorithm while revisiting batch size variations. Finally, the fourth experiment implemented Newton's Method to asses its comparative performance.

4.1 Control: Regular Gradient Descent

In our control experiment, we utilized standard gradient descent as the optimization method to establish a baseline for comparison across all subsequent experiments. Four distinct batch sizes were tested to observe their influence on the convergence rate and overall performance of the neural network. By maintaining consistent conditions in terms of initialization and network architecture, this experiment provided a foundational benchmark for evaluating alternative optimization techniques. (Figure 1,2)

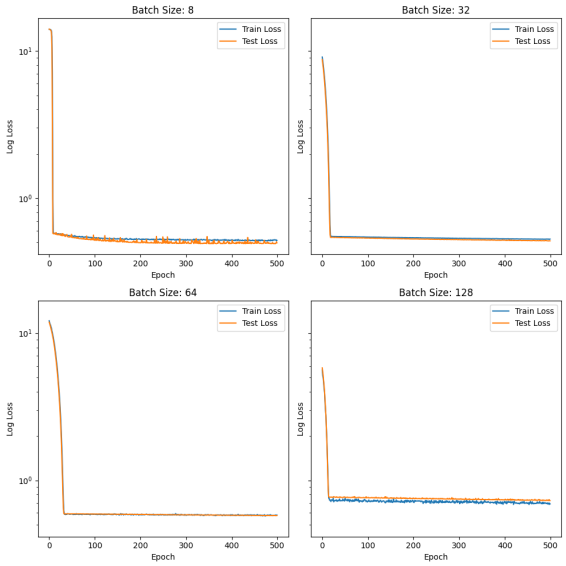


Figure 1: Multi Batch Variance:
Titanic Training vs Test Loss Graph for Gradient Descent

4.2 Experiment 1: Gradient Descent with Xavier Adjusted Weights

The first experiment aimed to investigate the impact of Xavier initialization on the performance of gradient descent. This initialization technique adjusts the weight distributions to address vanishing or exploding gradients, potentially improving convergence rates. By directly comparing results to the control, this experiment assessed whether Xavier initialization could enhance the network's accuracy and training efficiency under identical conditions.(Figure 3,4)

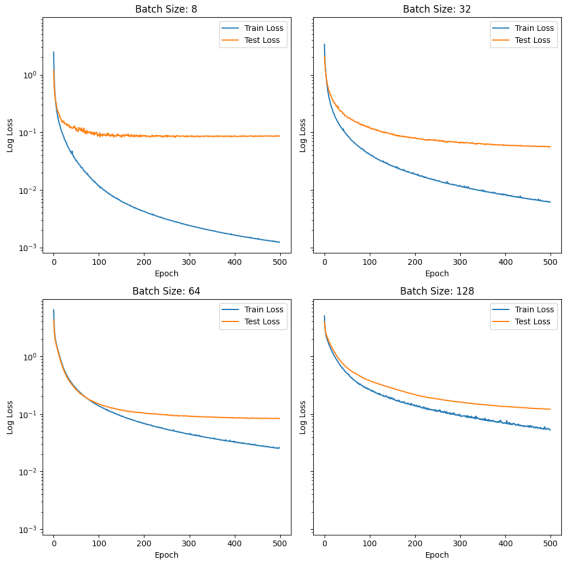


Figure 2: Multi Batch Variance:
Handwriting Training vs Test Loss Graph for Gradient Descent

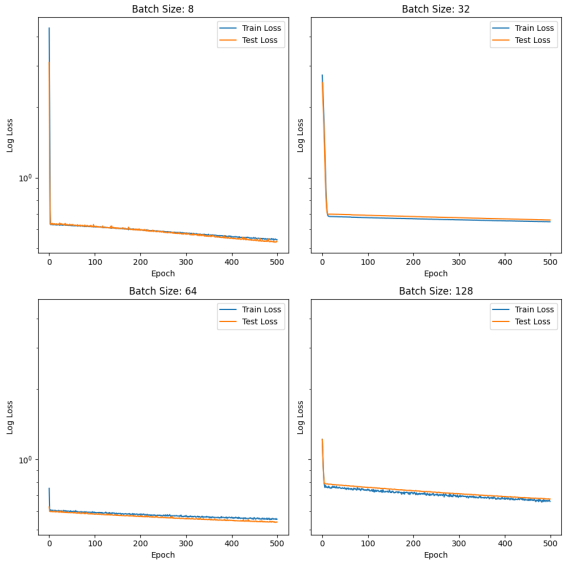


Figure 3: Multi Batch Variance:
Titanic Training vs Test Loss Graph for Xavier Adjustment

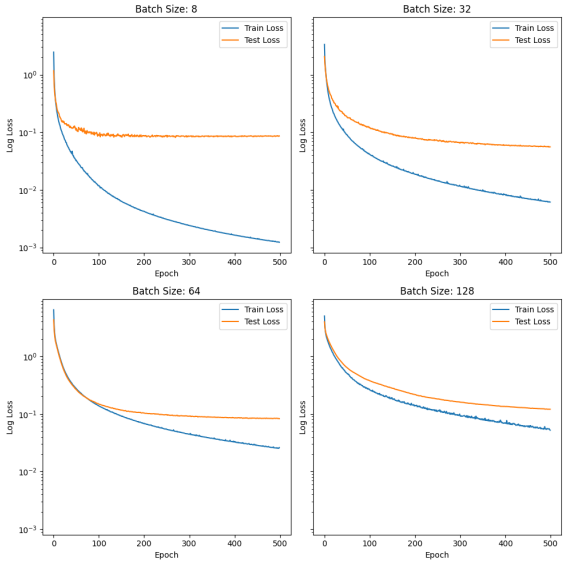


Figure 4: Multi Batch Variance:
Handwriting Training vs Test Loss Graph for Xavier Adjustment

4.3 Experiment 2: Nesterov Momentum

The second experiment explored the use of Nesterov Momentum as an alternative optimization method. This approach introduces a look ahead mechanism, which anticipates future gradient directions for more efficient convergence. Instead of varying batch sizes as in the control, this experiment focused on testing friction weights to determine their effect on stability and accuracy. The results provided insights into the balance between acceleration and overshooting during training. (Figure 5,6)

4.4 Experiment 3: Adam Algorithm

In the third experiment, we replaced standard gradient descent with the Adam optimization algorithm, known for its adaptive learning rates and efficient handling of sparse gradients. This experiment revisited the batch size variations from the control to assess the sensitivity of Adam's performance to different data processing scales. By comparing results with both the control and other experiments, this study evaluated Adam's adaptability and effectiveness for the given data sets. (Figure 7,8)

4.5 Experiment 4: Newtons Method

The final experiment implemented Newton's Algorithm as the optimization technique, leveraging second-order derivatives to achieve potentially faster convergence. This method, while computationally intensive, can provide superior accuracy in certain scenarios. By analyzing its performance under conditions consistent with prior experiments, this study assessed whether the benefits of

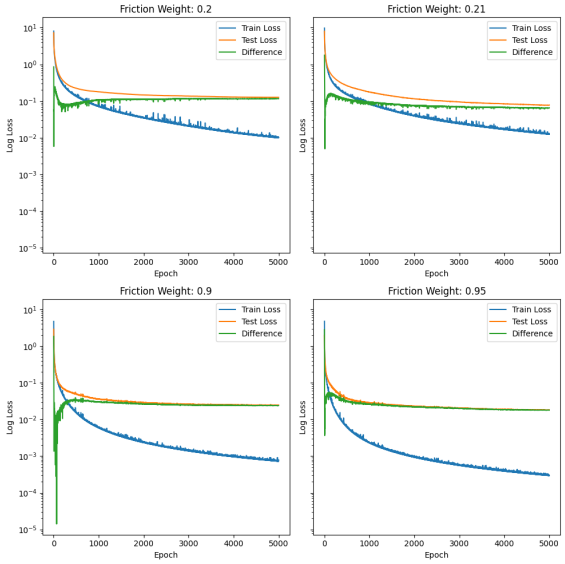


Figure 5: Friction Weight Variance:
Titanic Training vs Test Loss Graph for Nesterov Momentum

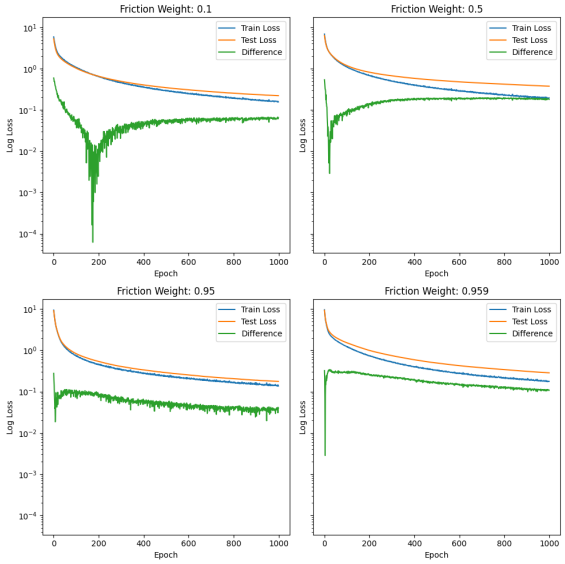


Figure 6: Friction Weight Variance:
Handwriting Training vs Test Loss Graph for Nesterov Momentum

Newton's Algorithm outweighed its complexity and resource demands.

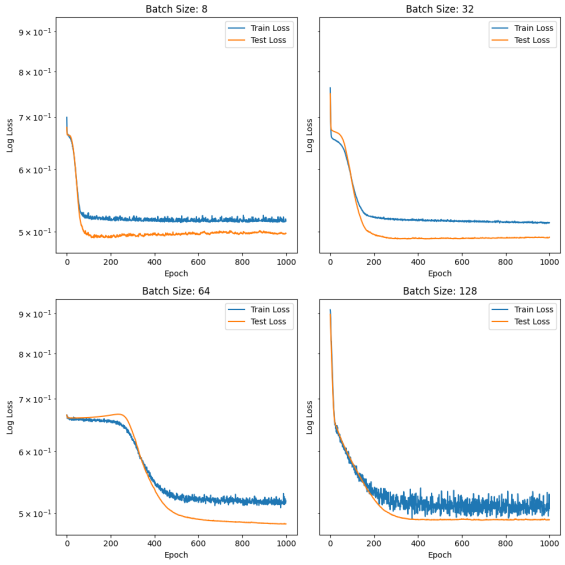


Figure 7: Batch Size Variance:
Titanic Training vs Test Loss Graph for Adam Descent

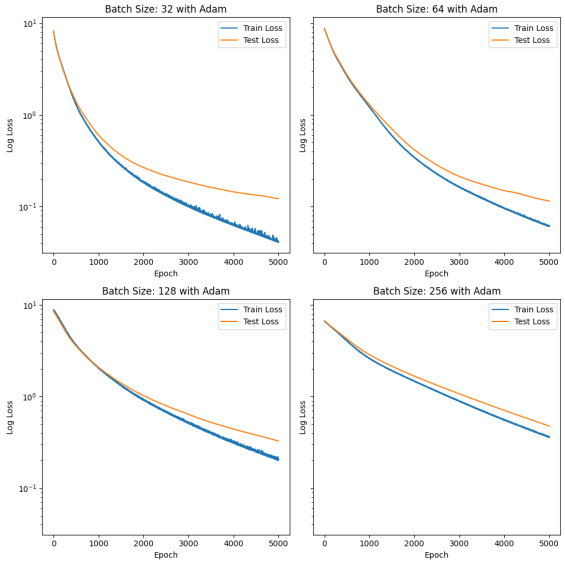


Figure 8: Batch Size Variance:
Handwriting Training vs Test Loss Graph for Adam Descent

4.6 Computational Analysis

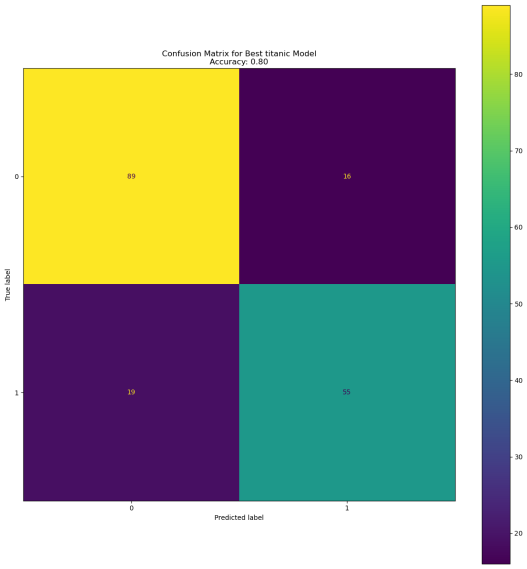
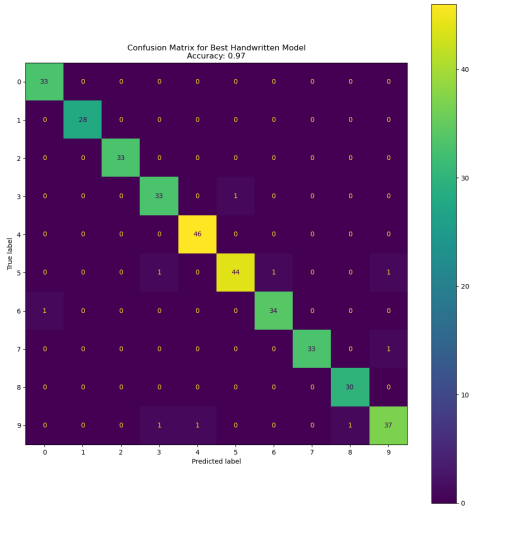


Figure 9: Confusion Matrix for Best Titanic Model



Best Params: {'batch_size': 32, 'lambda_reg': 1e-05, 'learning_rate': 0.01, 'max_epochs': 50, 'optimizer': 'adam', 'optimizer_kw_args': {'p1': 0.9, 'p2': 0.999}}

Figure 10: Confusion Matrix for Best Handwritten Digits Model

4.7 Experiment Results

Table 1: Handwritten Digits Results

optimizer	$\max_{e}pochs$	Fit Time (s) (mean)	Fit Time (s) (std)	Accuracy (mean)	Accuracy (std)
Adam	10	0.57	0.06	0.57	0.38
	50	2.64	0.21	0.62	0.33
	100	5.00	1.05	0.70	0.27
GD	10	0.53	0.05	0.45	0.32
	50	2.61	0.08	0.53	0.38
	100	5.11	0.23	0.54	0.39
Nesterov	10	0.53	0.04	0.54	0.35
	50	2.60	0.11	0.61	0.32
	100	5.23	0.25	0.65	0.31
Newton	10	0.55	0.08	0.73	0.18
	50	2.63	0.26	0.76	0.20
	100	5.17	0.47	0.73	0.25

Table 2: Titanic Results

optimizer	$\max_epochs$	Fit Time (s) (mean)	Fit Time (s) (std)	Accuracy (mean)	Accuracy (std)
Adam	10	0.06	0.02	0.56	0.25
	50	0.27	0.10	0.61	0.22
	100	0.54	0.21	0.65	0.18
	200	1.02	0.44	0.69	0.13
GD	10	0.05	0.02	0.43	0.13
	50	0.23	0.08	0.46	0.15
	100	0.48	0.15	0.46	0.16
	200	0.93	0.34	0.49	0.19
Nesterov	10	0.05	0.02	0.46	0.15
	50	0.24	0.08	0.52	0.16
	100	0.49	0.15	0.55	0.17
	200	0.96	0.33	0.58	0.17
Newton	10	0.05	0.02	0.67	0.07
	50	0.24	0.09	0.71	0.10
	100	0.47	0.17	0.71	0.10
	200	0.94	0.32	0.71	0.11

5 Contributions

- **Alexander Bernal** Wrote the sections on Newton's method and Adam optimizer. Implemented newtons method. The function handles Hessian approximation, bias correction through regularization, and precise weight updates using the gradient and inverse Hessian. It includes practical considerations, such as numerical stability using pseudo-inverses and regularized fallbacks

- **Barath Kurapati** Implemented Xavier initialization. This included researching the method, understanding its mathematical foundation, and integrating it into the neural network structure. The implementation ensures that weights are initialized appropriately for efficient training and consistent activation variances across layers. He also created his own tests to ensure the accuracy of his implementation.

- **Nicholas Livingstone** Implemented custom sklearn classifier wrapper for Neural Network implementation. Provided general guidance and code review throughout project. Contributed to analysis and experiment results.

- **Danny Metcalfe** Implemented Nesterov momentum learning and Adam adaptive learning rates.

- **Key Vollers** Implemented the code for the first three experiments and the control optimizing the FNN networks with the data that was assigned. Wrote this portion of the essay to extrapolate the experimental results. Performed code review for the first three implementations performed by Alexander, Danny and Barath.