FOML Assignment 3 Report: The "Mini-ML" Library and Autograd Engine

Akshay S Harits
ES23BTECH11002

November 6, 2025

Contents

1 Problem 3: Spam Classification Experiment

This experiment evaluated the performance of our custom SGD-based `LogisticRegression` model from the `my_ml_lib` library on the Spambase dataset. We compared the model's performance on the raw, unscaled data versus data preprocessed with our custom `StandardScaler`.

1.1 Results

We performed 5-fold cross-validation on the 80% training split to find the optimal L2 regularization hyperparameter, `alpha`, from the set $\{0.001, 0.01, 0.1, 1, 10, 100, 1000\}$. The model was then re-trained on the full training set using the best `alpha` and evaluated on the 20% unseen test set.

The final test errors and chosen hyperparameters are summarized in Table 1. The plot of validation accuracy vs. $\log(\alpha)$ is shown in Figure 1.

Table 1: Final Test Error for Spam Classification (using SGD).

Preprocessing	**Best Alpha**	**Train Error**	**Test Error**
Raw	0.1	0.3280	0.3138
Standardized	0.001	0.0766	**0.0771**

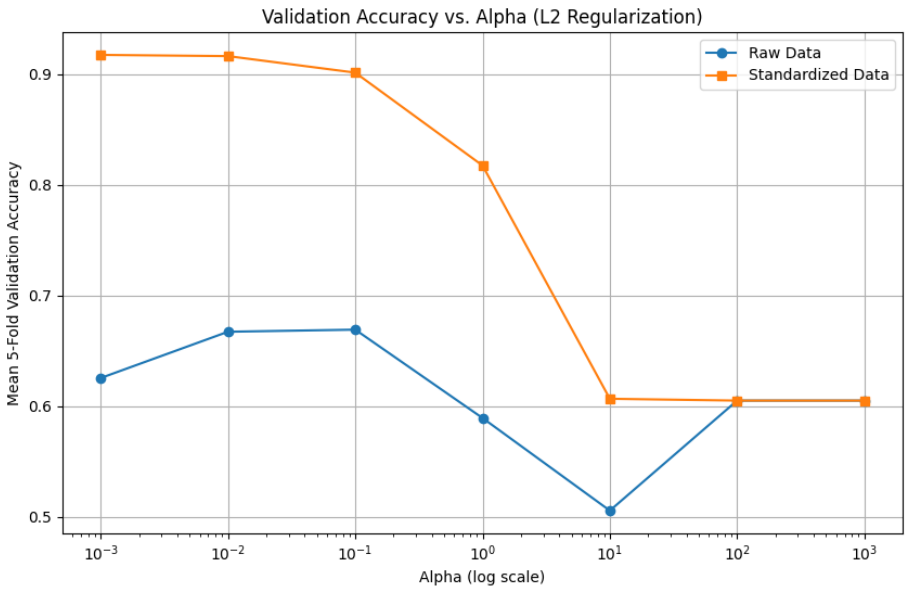

Figure 1: Validation Accuracy vs. Alpha (L2 Regularization) for raw and standardized data. Note the erratic performance and `RuntimeWarning`s associated with the raw data.

1.2 Analysis: The Effect of Standardization

As shown in Table 1, the effect of standardization was dramatic and critical. The model trained on raw data performed very poorly, with a **test error of 31.38%**, while the model trained on standardized data performed exceptionally well, with a **test error of 7.71%**.

This massive difference is due to two primary factors:

1. **Optimizer Stability (The Main Reason):** The console output showed multiple `RuntimeWarning: overflow encountered` and `invalid value encountered` errors when training on the raw

data. This is direct proof that the optimizer was failing. The Spambase dataset features have vastly different scales (e.g., word frequencies vs. `capital_run_length` features in the thousands). An unscaled feature space results in a cost function "canyon" that is extremely steep in some directions and flat in others. A single learning rate for Stochastic Gradient Descent (SGD) is too large for the steep directions, causing the calculations (like `X_batch @ self.w_`) to "overflow" and produce `NaN` (invalid) values. This instability prevents the optimizer from converging. By scaling all features to a mean of 0 and a standard deviation of 1, `StandardScaler` creates a much more spherical, well-behaved cost surface, allowing SGD to converge efficiently to a good solution.

2. **Effective Regularization:** L2 regularization (`alpha`) penalizes the squared magnitude of the weights. When features are unscaled, the model is forced to learn very small weights for large-scale features and large weights for small-scale features. This makes the L2 penalty apply unevenly. Standardization ensures all features are on the same scale, so the L2 penalty is applied "fairly" to all features. This is supported by the CV plot (Figure 1), which shows the standardized model's accuracy is higher and more stable, peaking at a very small `alpha` (0.001), while the raw data curve is erratic and performs poorly.

1.3 Note on Implementation (IRLS vs. SGD)

For the `LogisticRegression` module, our library contains both IRLS and SGD fit methods.

For the Q5 experiment (`capstone_showdown.ipynb`), `SGD` was used because running IRLS (which computes an $N \times N$ Hessian matrix) on 50,000 samples for 10 models was extremely slow.

For consistency, this Problem 3 experiment (`run_spam_experiment.py`) also uses the `SGD` method by default, which produced the results in Table 1 and Figure 1. However, the `IRLS` method is also available in the code but is commented out. While much slower, IRLS also achieves high-quality results on the standardized data, as shown in Table 2.

Table 2: Final Test Error for Spam Classification (using IRLS).

Preprocessing	**Best Alpha**	**Train Error**	**Test Error**
Raw	0.1	0.2899	0.2932
Standardized	0.001	0.0720	**0.0727**

2 Problem 4: Autograd Engine Visualization

This problem required building the core autograd engine (`Value` object) and the `nn` module framework. The `visualize.py` script was used to test the graph-building and backpropagation capabilities on a small MLP.

2.1 Computation Graph Visualization

Figure 2 shows the full computation graph generated by a single forward and backward pass of a tiny MLP: `Sequential(Linear(in_features=2, out_features=3), ReLU())`, which is then passed through a `sum()` to create a final loss node for backpropagation.

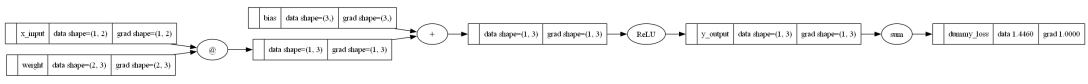

Figure 2: Autograd computation graph for a tiny MLP. Rectangles are `Value` nodes (data/params), and ovals are `op` nodes (operations).

2.2 Analysis of the Graph

The visualization confirms the successful implementation of the autograd engine:

- **Nodes and Edges:** The graph consists of two types of nodes. The **rectangles** are `Value` objects, which are the core of the engine. Each `Value` node stores its `data` (a NumPy array), its accumulated `grad` (gradient), and a set of its parent nodes (`_prev`). The graph clearly shows `Value` nodes for the `x_input`, the `Linear` layer's `weight` and `bias`, and all intermediate results.
- **Operations:** The **ovals** are `op` (operation) nodes. These are not `Value` objects themselves but represent the function (`@` for matmul, `+`, `ReLU`, `sum`) that created a new child `Value` node.
- **The `backward()` Pass:** The `backward()` function performs reverse-mode automatic differentiation. When called on the final `dummy_loss` node, it:
  1. Performs a topological sort of the graph to find all parent nodes.
  2. Sets the gradient of the `dummy_loss` node to 1.0.
  3. Iterates through the sorted list in reverse. For each `Value` node, it calls its `_backward` function, which applies the chain rule for that specific operation. For example, the `+` op's backward pass distributes its `grad` to both of its parents, while the `@` op's backward pass performs the correct matrix-multiplied gradient calculations. This process ensures that the gradient is accumulated correctly in the `.grad` attribute of every node, all the way back to the inputs and model parameters.

3 Problem 5: Capstone Showdown (Fashion-MNIST)

For the final problem, four different models were trained and evaluated on the Fashion-MNIST dataset to find the best-performing architecture. The 60k training images were split into a 50k training set and a 10k validation set, per the assignment guidelines.

3.1 Model Comparison Results

All four models were trained on the 50k scaled training samples and evaluated on the 10k scaled validation samples. The final comparison of their best-achieved validation accuracies is shown in Table 3.

Table 3: Final Model Comparison on Fashion-MNIST Validation Set.

Model	Best Validation Accuracy
MLP	**89.27%**
Softmax (Raw)	85.83%
OvR Logistic (SGD)	84.11%
Softmax (RBF)	10.20%

3.2 Training and Validation Plots

The training histories for the three autograd-based models were recorded. The plots for Training/Validation Loss and Validation Accuracy versus Epochs are shown in Figure 3.

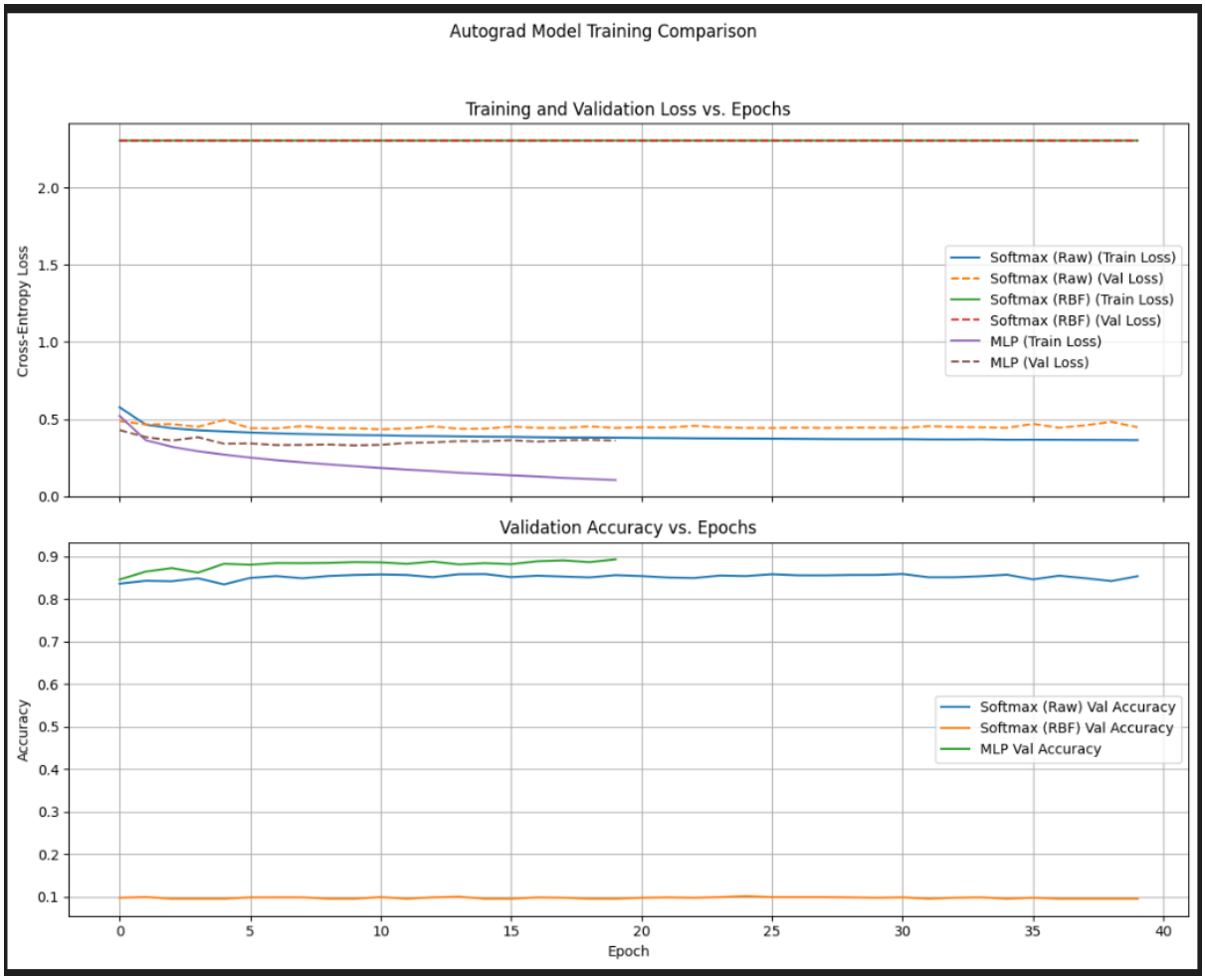

Figure 3: Training/Validation Loss and Validation Accuracy vs. Epochs for the three autograd models.

3.3 Analysis: Model Performance (The Winner)

The **Multi-Layer Perceptron (MLP) was the clear winner**, achieving **89.27%** validation accuracy.

This result is expected. The Fashion-MNIST dataset is a complex image classification task that is not linearly separable. The `MLP`'s success is due to its architecture:

- The `ReLU` activation functions introduce **non-linearity** into the model.
- This non-linearity allows the stacked `Linear` layers to learn complex, hierarchical patterns from the 784 pixel features, rather than just a single linear decision boundary.

The other two linear models, `Softmax (Raw)` (85.83%) and `OvR Logistic (SGD)` (84.11%), performed respectably but were fundamentally limited by their linearity. They could only find a single hyperplane to separate each class, which is not powerful enough to capture the intricate differences between items like "T-shirt/top," "Shirt," and "Pullover."

3.4 Analysis: Hyperparameter Tuning and Surprises

The tuning process and results revealed several key insights, as required by the assignment.

- **Important Hyperparameters:** For all autograd models (Softmax, MLP), the `learning_rate` was the most critical parameter. A value of `0.05` provided stable convergence. The `batch_size` of `128` offered a good balance between gradient stability and training speed. For the `OvR Logistic (SGD)` model, the `alpha` (L2 regularization) parameter was also tuned, with `0.001` providing the best result for the standardized data.

- **Number of Epochs:** The plots (Figure 3) show that the number of epochs was key for the autograd models. The `Softmax (Raw)` model's accuracy fluctuated but did not strongly overfit within 40 epochs. The `MLP`'s validation accuracy, however, peaked at **Epoch 20 (89.27%)**. This peak represents the optimal trade-off between learning the training data and generalizing to the validation data.

- **The "Surprise" (Model 3 Failure):** The most significant surprise was the complete failure of the `Softmax (RBF)` model, which scored only **10.20%** (no better than random guessing). This approach reduced the 784-dimensional data to 200 features based on Gaussian similarity to 200 random training samples. This failure indicates that 200 random centers are not nearly enough to create a useful, low-dimensional representation for such high-dimensional image data, as also noted in the `Q5_Note.pdf`. The model could not find any meaningful patterns in this new feature space, resulting in a flat loss and accuracy curve.