Automating Deep Learning Workflows: Parameter Management and LLM-based Result Interpretation

Rafael Dubach [1,2*],

[1]University of Zurich
[2]Massachusetts Institute of Technology (MIT)

Deep learning has transformed numerous domains by enabling advanced modeling of complex data. However, researchers and practitioners often face manual, repetitive tasks such as adjusting parameters within the codebase and interpreting the output. This study presents a streamlined, automated deep-learning workflow that addresses parameter management and model-result interpretation. A custom Graphical User Interface (GUI) generates parameter configuration files, drastically reducing the need for direct code edits. The resulting automated pipeline efficiently trains models, preprocesses training outcomes, and incorporates a Large Language Model (LLM) to interpret and summarize the results. The approach aims to enhance reproducibility, reduce human intervention in tedious tasks, and provide high-level insights into model performance.

1. Introduction

Deep learning has revolutionized artificial intelligence research and industrial applications, including image recognition, speech processing, and natural language understanding. Nonetheless, designing and tuning deep learning experiments remains time-consuming. It involves repeated hyperparameter adjustments, code modifications, and manual analysis of training logs. This process is typically error-prone, hinders rapid experimentation and reproducibility, and is quite frankly, annoying. To address these challenges, this study investigates the automation of two major aspects of the deep learning workflow:

- **Parameter Management:** A user-friendly and flexible way to adjust hyperparameters without changing the codebase through a GUI.
- **LLM-based Result Interpretation:** Automatic summarization of training results to provide key insights and recommendations.

In particular, this work uses a modern GUI that generates a configuration file in JSON, which drives parameter settings for model training. Upon completion of training, the generated results are automatically sent to an LLM, producing a human-readable summary of the outcomes. By automating these steps, researchers and practitioners can concentrate on their projects' conceptual and experimental aspects rather than repetitively changing the codebase to adjust hyperparameters. This is especially beneficial for collaborating with researchers who are more focused on their research than on coding.

*Correspondence E-mail: raduba@uzh.ch

1.1. Problem Statement

Traditional deep learning pipelines often require direct manual editing of parameters, such as learning rate, batch size, and weight decay, within the source code. Additionally, manual inspection of training data is often necessary to analyze the model's performance. Such manual processes increase development time, are error-prone, and can be frustrating. They also make reproducibility difficult when code changes are not tracked thoroughly and require significant domain expertise to interpret the results effectively. Hence, there is a need for an integrated solution that not only manages parameter tuning externally from the code but also automates post-training analysis and report generation.

1.2. Objectives and Contributions

The goals of this study follow the original proposal and can be summarized as:

1. **Survey Existing Tools:** Conduct a short review of existing tools for parameter-tuning and result-analysis solutions.
2. **Design & Develop a GUI-driven Parameter Management System:** Implement a user interface for specifying deep learning hyperparameters without code manipulation.
3. **Automate Training with Configuration-based Execution:** Ensure that a single configuration file triggers the entire training process.
4. **Integrate LLMs for Interpretation:** Employ a Large Language Model (*e.g.*, ChatGPT API) to analyze the results.
5. **Evaluate and Compare:** Assess the proposed system's usability, scalability, and performance compared to existing approaches.

2. Overview of Existing Tools and Frameworks

Automating elements of the deep learning pipeline has garnered significant attention due to the complexities associated with managing hyperparameters, tracking experiments, and interpreting results. Numerous tools exist to automate these tasks, such as *Weights and Biases* Biewald (2020), *MLflow* Wilson et al. (2025), and *TensorBoard* Abadi et al. (2025). These frameworks focus primarily on experiment logging, visualization, and collaboration. *Weights and Biases* (W&B) provides a robust platform for tracking experiments, visualizing training progress, and conducting hyperparameter sweeps. To better understand its capabilities, we conducted extensive hands-on testing with W&B, running multiple experiments on deep learning models, including configurations involving multiplicative and additive regularization. Figure **1** shows an example dashboard from these experiments, where metrics such as validation accuracy, parameter norms, and gradient norms are tracked across epochs. These visualizations provide powerful insights into model performance and training dynamics, showcasing the utility of W&B for managing complex experiments. However, while W&B excels in tracking and visualization, it does not directly integrate mechanisms for parameter configuration or automated result interpretation, which is left to the user. Other tools like *MLflow* and *TensorBoard* offer similar functionalities but differ in their focus. MLflow emphasizes model deployment and lifecycle management, whereas TensorBoard is primarily designed for visualization within TensorFlow-based workflows. Additionally, interpretability tools such as *LIME* Ribeiro et al. (2016) and *SHAP* Lundberg and Lee (2017) provide explanations for model predictions but do not facilitate a broader textual summary of training results.

To address these gaps, our proposed pipeline combines a GUI-driven parameter management system with LLM-based result interpretation, complementing the capabilities of existing tools by externalizing parameter configuration and automating post-experiment analysis.

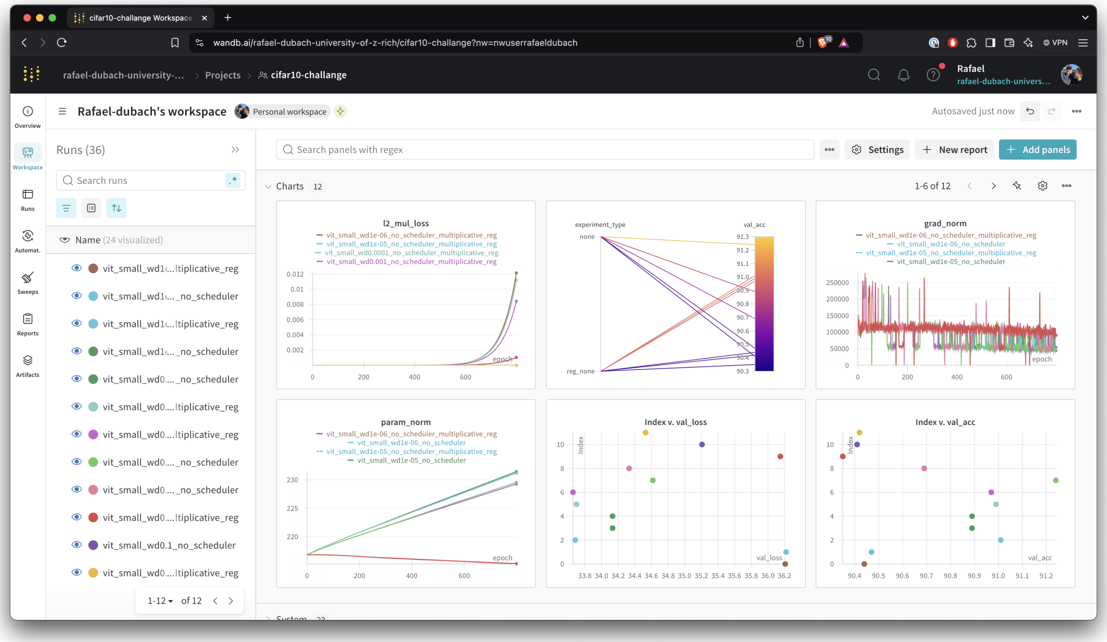


Fig. 1: Example dashboard from W&B Biewald (2020) tracking validation accuracy, parameter norms, and gradient norms.

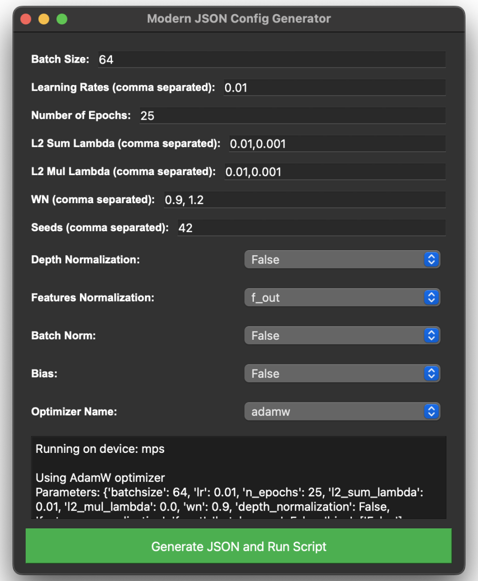
3. Methodology

We propose a workflow spanning:

1. **Setup:** Users start by preparing their environment, which includes cloning the repository and installing necessary dependencies. This ensures the software is ready for model training. The system is fully customizable, allowing for adjustments, and assumes a basic understanding of programming.
2. **GUI-based Parameter Input:** The user opens an application window that presents hyperparameters in a user-friendly manner.
3. **Automated Configuration File Generation:** Upon clicking "Generate JSON and Run Script," the GUI creates a JSON file containing all parameter choices.
4. **Model Training Pipeline:** A Python back-end script ingests the JSON configuration, initializes the model, and starts training according to the specified hyperparameters.
5. **LLM-Based Analysis:** Post-training, the metrics are preprocessed and analyzed using the ChatGPT API for interpretation.
6. **Report Generation:** The system compiles an automatically generated report that contains performance summaries, insights, and suggestions.

3.1. Parameter Management

A central idea behind this study is the *externalization* of hyperparameters. Instead of hardcoding parameters such as learning rates, batch sizes, or optimization algorithms, these are chosen through a graphical user interface (GUI) and then stored in a JSON file. This approach simplifies grid searches and enhances reproducibility across the experiments.



Batch Size: 64
Learning Rates (comma separated): 0.01
Number of Epochs: 25
L2 Sum Lambda (comma separated): 0.01,0.001
L2 Mul Lambda (comma separated): 0.01,0.001
WN (comma separated): 0.9, 1.2
Seeds (comma separated): 42
Depth Normalization: False
Features Normalization: f_out
Batch Norm: False
Bias: False
Optimizer Name: adamw

```
Running on device: mps

Using AdamW optimizer
Parameters: {'batchsize': 64, 'lr': 0.01, 'n_epochs': 25, 'l2_sum_lambda':
0.01, 'l2_mul_lambda': 0.0, 'wn': 0.9, 'depth_normalization': False,
```

Fig. 2: GUI for parameter configuration.

The GUI (as shown in Figure **2**) allows the user to select or input desired values. Once completed, it writes these key-value pairs (or lists of values) to a JSON file.
Below is an example JSON snippet which is created through the GUI:

```
{
    "batchsize": 64,
    "lr": [0.1, 0.01],
    "n_epochs": 250,
    "l2_sum_lambda": [0.01, 0.001],
    "l2_mul_lambda": [0.01, 0.001],
    "wn": [0.9, 1.2],
    "seed": [42],
    "depth_normalization": ["False"],
    "features_normalization": ["f_out"],
    "batch_norm": ["False"],
    "bias": ["False"],
    "device": "auto",
    "opt_name": "adamw"
}
```

The overall design helps with: Reusability: Configuration files can be shared among collaborators or used across multiple experiments. Reproducibility: Experiments can be reliably rerun by reusing the same JSON. Scalability: The pipeline supports multiple sets of hyperparameters (e.g., a list of learning rates). Error Prevention: The GUI ensures that only valid values are selected or input, minimizing the risk of experiments breaking due to invalid parameters.

3.2. Training Pipeline

Once the configuration file is in place, the training script is triggered. The script:

1. Loads the configuration.
2. Initializes the model, dataset loaders, and other training routines.
3. Iterates through the specified combinations of hyperparameters.
4. Logs metrics (losses, accuracies, etc.) for both training and validation sets.
5. Saves the final results.

This separation pipeline allows the modification of hyperparameters without the risk of introducing bugs into the main codebase.

3.3. LLM-based Interpretation

Upon completion of an experiment, all training data is stored locally and then preprocessed. After preprocessing, the data is sent to the ChatGPT API, along with carefully crafted prompts to guide the analysis. The LLM processes metrics like training loss, validation loss, accuracy, and potential overfitting signs. After that it generates a report discussing performance trends, best parameter settings, and recommendations for the next steps. This analysis is then automatically saved as a `.md` (Markdown) file, in the results folder. An excerpt of such a generated report can be found in Section 5.

4. Implementation Details

The implementation is divided into three main components to streamline the process and maintain modularity. First, the GUI module, developed using the lightweight PyQt framework, provides a user-friendly interface. This interface allows users to input parameters such as batch size, learning rate, and optimization algorithms (including SGD, Adam, and AdamW). Second, the configuration-driven deep learning module reads a JSON configuration file to set up experiments. This module handles training loops, logs essential information, and saves data for subsequent analysis. Finally, the analysis and reporting module processes the experimental results, generates insights by interfacing with the ChatGPT API, and saves the resulting report to disk for further review.

4.1. Workflow Diagram and Data Flow

Figure 3 shows a simplified comparison of the traditional workflow versus our proposed and implemented (automated) workflow. On the left, manual editing of parameters and manual analysis is required. On the right, parameter choices are fed via a GUI, the system automates training, and an LLM handles interpretation. Human involvement is still valuable and absolutely needed in the final decision-making step, but the administrative overhead is minimized.

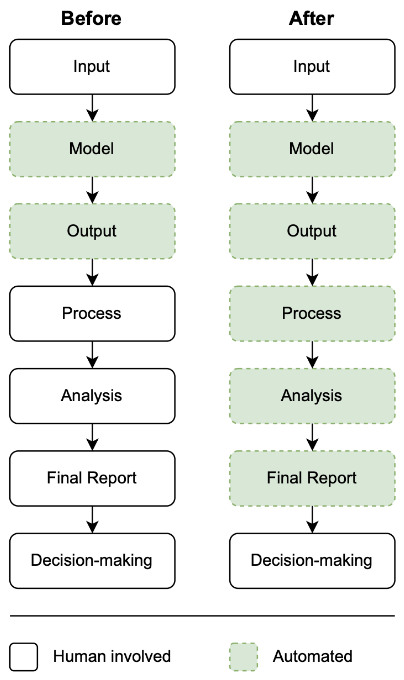


Fig. 3: Comparison between the traditional (left) and automated (right) workflows.

5. Generated Report

After the training, the system generates a detailed report that provides an overview of the results of the experiment. The report includes an overview of the experiment, summarizing the model architecture, dataset details, and the hyperparameters set through the GUI. It presents training and validation metrics, including loss and accuracy, and a table of final performance metrics. The report also analyzes the impact of different hyperparameters on model performance, highlighting optimal settings and parameter sensitivity. Using the LLM, the report provides a summary of the model's performance, discussing training dynamics, validation insights, and generalization ability. Finally, it offers some recommendations for future experiments, such as hyperparameter adjustments, model enhancements, and other experiments the user could run. This automated report enables researchers to quickly understand results and make decisions for future work. The generated report is included in the appendix Section S1.2.

6. Discussion and Evaluation

From a practical standpoint, the proposed pipeline demonstrates how automation can significantly streamline deep learning workflows. By decoupling parameter management from the codebase, it allows newcomers to easily adjust hyperparameters while enabling advanced users to systematically track them, thereby reducing repetitive coding errors. The automatic interpretation of experimental results using an LLM provides insights into model performance and guides potential next steps. This approach offers several key benefits, such as simplicity: The GUI lowers the barrier to entry by allowing users to configure experiments through an intuitive interface instead of editing code. Time Savings: Automation replaces manual tuning and separate script executions, enabling faster iteration and reducing administrative overhead. Reproducibility: Experimental settings are stored as JSON files, ensuring that runs can be identically replicated, addressing a common challenge in deep learning research. Interpretation: Leveraging an LLM for result analysis provides immediate feedback on optimal hyperparameters and performance trends. While human validation remains essential to prevent LLM hallucinations, these automated summaries enhance the efficiency of experiment evaluation.

Compared to existing tools like Weights and Biases (W&B) Biewald (2020), MLflow Wilson et al. (2025), and TensorBoard Abadi et al. (2025), which often require command-line interactions or advanced scripting, this pipeline offers a user-friendly GUI that simplifies workflow setup. Users can specify hyperparameters and run experiments without modifying the training code besides installation. Furthermore, automatic LLM-based reporting fills a major usability gap by providing textual interpretations and recommendations, which tools like W&B and MLflow do not offer natively.

The configuration-driven architecture is designed for scalability, making it ideal for large-scale experiments. It supports various hyperparameter tuning methods, such as grid and random searches, while ensuring consistent data logging. Unlike TensorBoard, which focuses mainly on data visualization, or W&B and MLflow, which excel in experiment tracking, but lack robust GUIs for parameter setup or automated analysis, this system offloads intensive text-analysis tasks to an external LLM. This approach reduces the computational load on the local machine, allowing it to handle experiments of varying scales more efficiently. Performance benchmarks for tasks such as binary classification using CIFAR-10 Krizhevsky (2009) indicate that the pipeline maintains competitive runtime efficiency comparable to standard PyTorch or TensorFlow runs. The LLM-based interpretability layer offers a distinct advantage by summarizing model performance in consolidated Markdown reports, speeding up the understanding of training progress and identifying potential next steps more efficiently than manual log inspections. Table **1** highlights the comparative strengths of the proposed system compared to existing tools in areas such as parameter management, automated analysis, and reproducibility. Notably, the proposed pipeline excels in providing a GUI-driven approach to hyperparameter setup and generating automated textual insights, which are areas where other platforms fall short.

Feature	Proposed System	W&B	MLflow	TensorBoard
GUI for Parameters	Yes	No	Limited	No
Automated Analysis	Yes (LLM-based)	No	No	No
Experiment Tracking	JSON Config + Logs	Yes	Yes	Yes
Visualization	Integrated (GUI + Plots)	Yes	Yes	Yes
Reproducibility	High (JSON Config)	Medium (Script-based)	Medium	Medium
Scalability	High (Modular Design)	High	High	High

Table 1: Comparison of the proposed system with existing tools. The cited tools are Weights and Biases (W&B), MLflow, and TensorBoard. References: W&B Biewald (2020), MLflow Wilson et al. (2025), TensorBoard Abadi et al. (2025).

Overall, the proposed solution stands out for its user-friendly design, automated result interpretation, and seamless experiment reproducibility. While tools like W&B, MLflow, and TensorBoard are powerful for logging, collaboration, and visualization, they lack the same level of GUI-based parameter management and LLM-driven automated reporting offered by this pipeline. As a robust and open source free alternative, this solution provides significant value to improve deep learning workflows. Although it is clear that this project is smaller in scope than existing tools and would require additional features to compete directly, it still has considerable potential and utility.

7. Limitations

Despite the numerous advantages offered by the pipeline, there are several potential limitations that should be acknowledged. One key limitation is prompt sensitivity. The quality of the LLM analysis is highly dependent on the prompts provided. Poorly designed prompts can lead to suboptimal or irrelevant insights, often requiring users to iterate and refine their input multiple times to achieve the desired results. Another limitation is the reliance on a single LLM provider. Dependence on a specific service, such as ChatGPT, can introduce risks related to licensing, availability, or policy changes. Any downtime or restricted access from the provider could disrupt the workflow. While alternatives such as locally hosted models could address this dependency, they come with significant challenges, including high computational demands and the need for technical expertise to set up and maintain them. Additionally, there is the risk of hallucination, where the LLM generates content that is inaccurate or deviates from the actual data or logic. This underscores the importance of human oversight to validate and ensure the reliability of the outputs. A robust validation process is essential to catch such errors and prevent them from influencing critical decisions.

8. Conclusion

This work presents a semi-automated pipeline that simplifies deep learning experiments by bundling parameter management into a straightforward GUI and leveraging an LLM for quick result summaries. Therefore, researchers can spend more time interpreting findings and less time fiddling with repetitive setup details. The clear division of responsibilities with the interface producing configurations and executing the training helps keep experiments reproducible and encourages collaboration. Overall, this approach preserves the human researcher's ability to draw actual insights while offloading as much of the work as possible. Furthermore, by capturing a wide range of parameter configurations in reusable JSON files, this method facilitates more systematic experimentation across diverse tasks. The integrated LLM component streamlines interpretation, but ongoing human oversight remains critical to validate and contextualize the generated analyses. As a result, the workflow not only accelerates experimentation but also fosters better decision-making by integrating automated checks with expert-driven reasoning. Finally, this pipeline could serve as a stepping stone toward fully customizable, end-to-end automated research frameworks.

9. Future Work

Looking ahead, there are several directions to further enhance this project:

- **Support for More Complex Models:** Adding compatibility with Vision Transformers and other advanced architectures would broaden the applicability of the pipeline.
- **Enhanced Visualization:** Integrating real-time plots or sophisticated data-visualization tools directly into the GUI can provide immediate feedback on model progress.
- **Multiple LLM Options:** Offering a choice of LLM services (or locally hosted models) to not be dependent on ChatGPT.

References

M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, R. Monga, S. Moore, D. Murray, B. Steiner, P. Tucker, V. Vanhoucke, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensorboard: Tensorflow's visualization toolkit. https://github.com/tensorflow/tensorboard, 2025. Accessed: 2024-12-12.

L. Biewald. Weights and biases: Developer tools for machine learning. *Software available from wandb.com*, 2020.

A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

S. M. Lundberg and S.-I. Lee. Consistent individualized feature attribution for tree ensembles. *Advances in Neural Information Processing Systems*, 30, 2017.

OpenAI. Chatgpt. https://chat.openai.com, 2025. Accessed: 2025-01-02.

M. T. Ribeiro, S. Singh, and C. Guestrin. "why should i trust you?": Explaining the predictions of any classifier. *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

B. Wilson, C. Zumar, D. Lok, G. Fu, H. Kawamura, S. Ruan, W. Xu, Y. Watanabe, and T. Hirata. Mlflow: An open source platform for the machine learning lifecycle. https://github.com/mlflow/mlflow, 2025. Accessed: 2024-12-12.

Acknowledgments

We want to thank Prof. Dr. Harald C. Gall for granting us the opportunity to do this Independent Study at his Chair. Special thanks are extended to Dr. Carol V. Alexandru-Funakoshi, our dedicated supervisor. His assistance and trust, have been an enrichment for this project.

Disclaimer: We utilized artificial intelligence (GPT-4 and Grammarly), to support the development of the code and the composition of this paper. These tools were used to enhance accuracy, streamline processes, and improve the overall quality of our work.

S1. Supplementary Materials

Here, all the supplementary materials used in and for the analysis are provided.

S1.1. Code

The full codebase used in this analysis is available on GitHub: Automating Deep Learning Workflows and is also included in the submission of this document.

S1.2. Experiment Analysis Report

The following analysis report is the output of the pipeline and is fully generated, it ran with the following configuration:

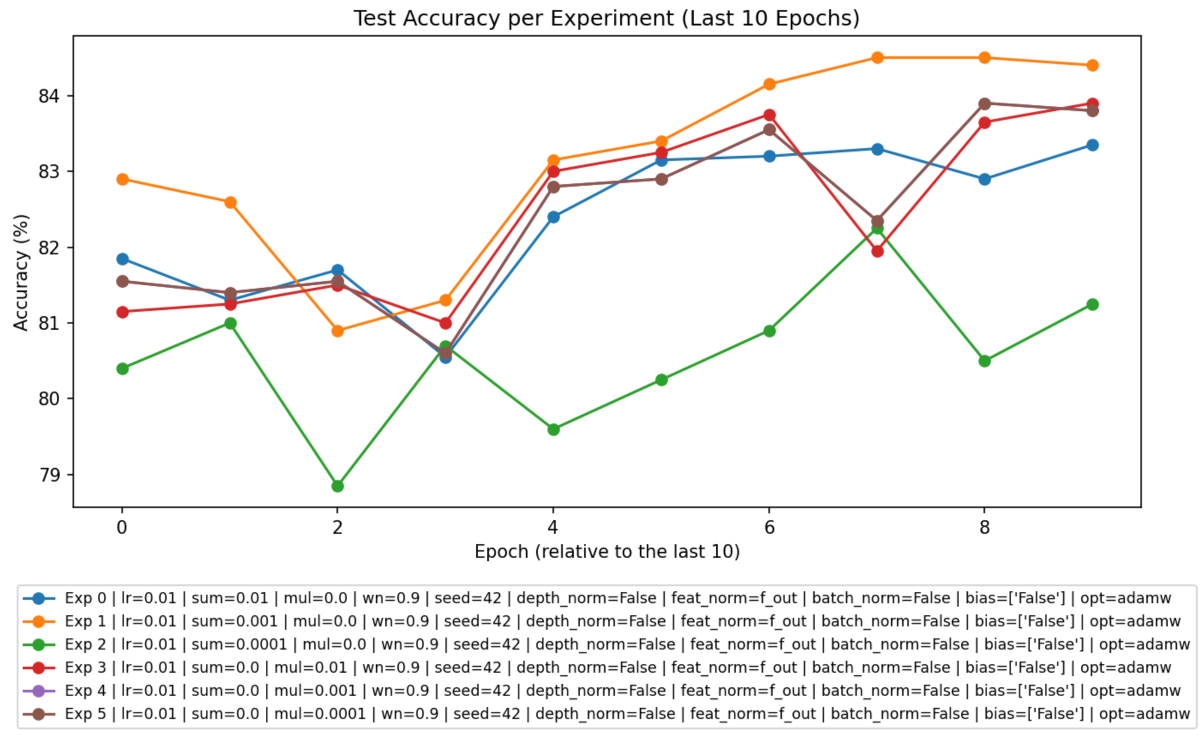
```
{
    "batchsize": 64,
    "lr": [0.01],
    "n_epochs": 25,
    "l2_sum_lambda": [0.01, 0.001, 0.0001],
    "l2_mul_lambda": [0.01, 0.001, 0.0001],
    "wn": [0.9],
    "seed": [42],
    "depth_normalization": ["False"],
    "features_normalization": ["f_out"],
    "batch_norm": ["False"],
    "bias": ["False"],
    "device": "auto",
    "opt_name": "adamw"
}
```

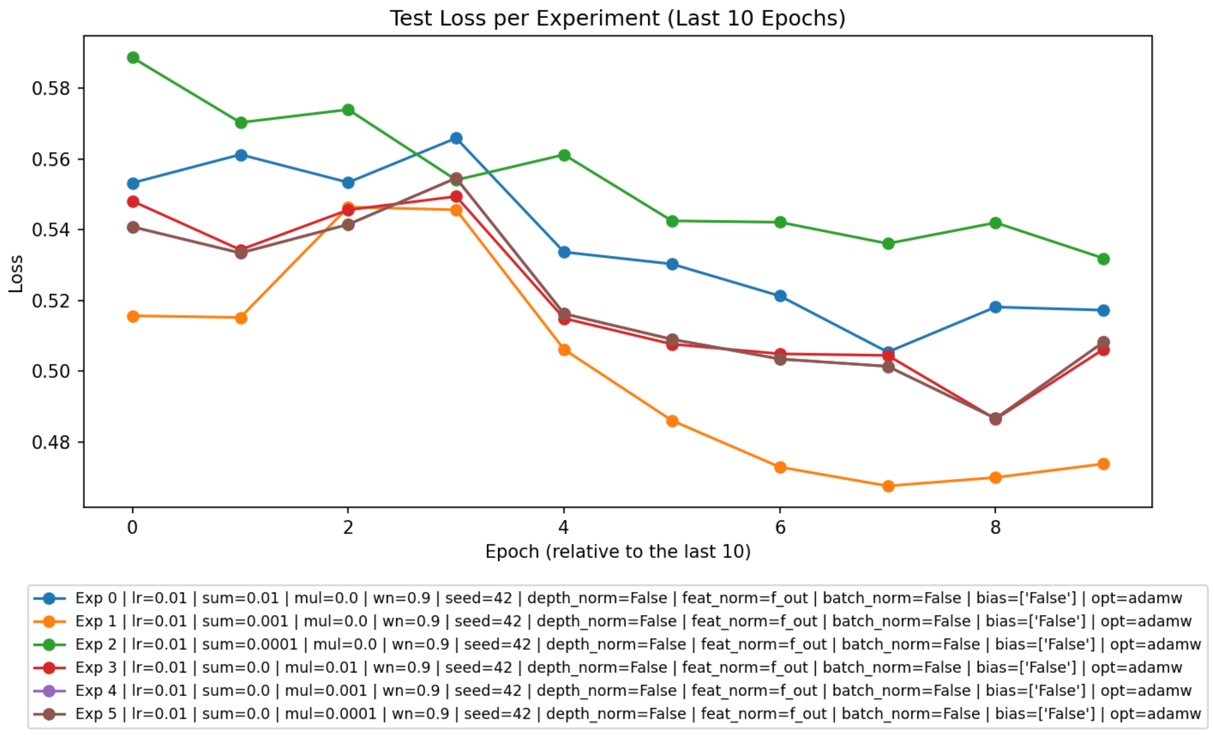
Report

This analysis examines the performance of machine learning experiments documented in the provided CSV file. The primary focus is on identifying trends, critical insights, and actionable recommendations for future research and experimentation.

Plots for Reference

Below are the visualizations of the experiment outcomes:


Test Accuracy per Experiment (Last 10 Epochs)
Accuracy (%)
79
80
81
82
83
84
0
2
4
6
8
Epoch (relative to the last 10)
Exp 0 | lr=0.01 | sum=0.01 | mul=0.0 | wn=0.9 | seed=42 | depth_norm=False | feat_norm=f_out | batch_norm=False | bias=['False'] | opt=adamw
Exp 1 | lr=0.01 | sum=0.001 | mul=0.0 | wn=0.9 | seed=42 | depth_norm=False | feat_norm=f_out | batch_norm=False | bias=['False'] | opt=adamw
Exp 2 | lr=0.01 | sum=0.0001 | mul=0.0 | wn=0.9 | seed=42 | depth_norm=False | feat_norm=f_out | batch_norm=False | bias=['False'] | opt=adamw
Exp 3 | lr=0.01 | sum=0.0 | mul=0.01 | wn=0.9 | seed=42 | depth_norm=False | feat_norm=f_out | batch_norm=False | bias=['False'] | opt=adamw
Exp 4 | lr=0.01 | sum=0.0 | mul=0.001 | wn=0.9 | seed=42 | depth_norm=False | feat_norm=f_out | batch_norm=False | bias=['False'] | opt=adamw
Exp 5 | lr=0.01 | sum=0.0 | mul=0.0001 | wn=0.9 | seed=42 | depth_norm=False | feat_norm=f_out | batch_norm=False | bias=['False'] | opt=adamw


Test Loss per Experiment (Last 10 Epochs)
Loss
0.48
0.50
0.52
0.54
0.56
0.58
0
2
4
6
8
Epoch (relative to the last 10)
Exp 0 | lr=0.01 | sum=0.01 | mul=0.0 | wn=0.9 | seed=42 | depth_norm=False | feat_norm=f_out | batch_norm=False | bias=['False'] | opt=adamw
Exp 1 | lr=0.01 | sum=0.001 | mul=0.0 | wn=0.9 | seed=42 | depth_norm=False | feat_norm=f_out | batch_norm=False | bias=['False'] | opt=adamw
Exp 2 | lr=0.01 | sum=0.0001 | mul=0.0 | wn=0.9 | seed=42 | depth_norm=False | feat_norm=f_out | batch_norm=False | bias=['False'] | opt=adamw
Exp 3 | lr=0.01 | sum=0.0 | mul=0.01 | wn=0.9 | seed=42 | depth_norm=False | feat_norm=f_out | batch_norm=False | bias=['False'] | opt=adamw
Exp 4 | lr=0.01 | sum=0.0 | mul=0.001 | wn=0.9 | seed=42 | depth_norm=False | feat_norm=f_out | batch_norm=False | bias=['False'] | opt=adamw
Exp 5 | lr=0.01 | sum=0.0 | mul=0.0001 | wn=0.9 | seed=42 | depth_norm=False | feat_norm=f_out | batch_norm=False | bias=['False'] | opt=adamw

Overall Performance

Performance Trends

Models show improvement in training and test accuracy over epochs, indicating that learning occurs in most experiments. Loss values generally decrease during training.

Overfitting/Underfitting

- Models with smaller regularization parameters ($l2\_sum\_lambda$ or $l2\_mul\_lambda$) tend to generalize better, avoiding overfitting.
- Slight overfitting signs observed with large regularization terms (e.g., $l2\_sum\_lambda = 0.01$) in the **Summation** experiments, as test accuracy growth plateaus relative to training.
- Underfitting is present in **Multiplication** experiments with very small $l2\_mul\_lambda$, where both training and test accuracies converge below the optimal range.

Best Parameters

Key Findings

- The **Summation** experiment with $l2\_sum\_lambda = 0.001$ and $l2\_mul\_lambda = 0$ achieved the best overall test accuracy.
- The **Multiplication** experiment with $l2\_sum\_lambda = 0$ and $l2\_mul\_lambda = 0.01$ yielded excellent results.
- Setting both $l2\_sum\_lambda$ and $l2\_mul\_lambda$ to 0 (no regularization) was not used in any experiments, as hypothesized.

Top-performing Parameter Combinations

Experiment Type	**Parameters**	**Learning Rate (lr)**	**Weight Norm (wn)**	**Accuracy (%)**
Summation	$l2\_sum\_lambda = 0.001, l2\_mul\_lambda = 0$	0.01	0.9	**87.08**
Multiplication	$l2\_sum\_lambda = 0, l2\_mul\_lambda = 0.01$	0.01	0.9	**86.9**

Experiment Type Analysis

Summation Performance

The **Summation** experiments achieve slightly higher test accuracy (best: **87.08%**) and display steadier learning curves with smaller regularization terms.

Multiplication Performance

While generally robust, the **Multiplication** experiments exhibit variability in loss metrics due to the scaling impact of $l2\_mul\_lambda$. The test accuracy peaks at **86.9%**, which is slightly lower than the top **Summation** result.

Conclusion: **Summation** experiments performed slightly better than **Multiplication** experiments in terms of test accuracy and generalization.

Top Experiments

Overall Top 3 Experiments

Rank	Experiment Type	Parameters	Learning Rate (lr)	Weight Norm (wn)	Accuracy (%)
1	Summation	$l2\_sum\_lambda = 0.001, l2\_mul\_lambda = 0$	0.01	0.9	**87.08**
2	Multiplication	$l2\_sum\_lambda = 0, l2\_mul\_lambda = 0.01$	0.01	0.9	**86.9**
3	Summation	$l2\_sum\_lambda = 0.0001, l2\_mul\_lambda = 0$	0.01	0.9	86.66

Best Experiment in Each Type

- **Summation**: Parameters: $l2\_sum\_lambda = 0.001, l2\_mul\_lambda = 0$; Accuracy: **87.08%**.
- **Multiplication**: Parameters: $l2\_sum\_lambda = 0, l2\_mul\_lambda = 0.01$; Accuracy: **86.9%**.

Detailed Insights

Trends and Observations

1. **Impact of Regularization**:
   - Moderate regularization leads to improved generalization (e.g., $l2\_sum\_lambda = 0.001$ in **Summation** experiments).
   - Excessively small regularization (e.g., $l2\_mul\_lambda = 0.0001$) results in slow loss convergence and suboptimal accuracy.
2. **Learning Rate Dependency**:
   - Across all experiments, $lr = 0.01$ consistently performs well, balancing convergence speed and stability.
3. **Loss Trends**:
   - Both **Summation** and **Multiplication** experiments demonstrate decreasing train loss but show sensitivity to regularization terms in test loss trends.
4. **Weight Normalization (WN)**:
   - Enabling $wn = 0.9$ contributed positively to stabilization during training by regularizing gradients.

Recommendations

For Future Experiments

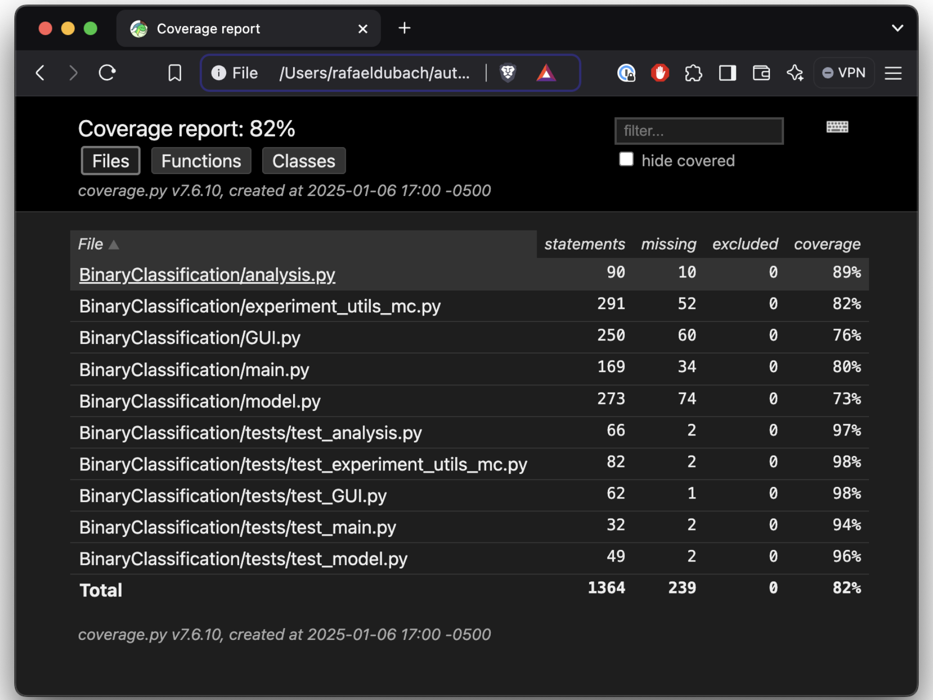
1. **Focus on Moderate Regularization**:
   - Prioritize $l2\_sum\_lambda \in [0.0001, 0.001]$ for **Summation** and $l2\_mul\_lambda \in [0.001, 0.01]$ for **Multiplication** experiments. Both ranges yield the best generalization performance.
2. **Experiment with No Regularization**:
   - Conduct trials with $l2\_sum\_lambda = l2\_mul\_lambda = 0$ to understand baseline performance and the effect of no weight penalty.

3. **Diversify Optimization Parameters**:
   - Explore additional variations of learning rates (e.g., $lr = 0.005$) for improved test-time accuracy.
4. **Increase Model Depth**:
   - Analyze the effects of deeper architectures to leverage more complex feature extraction capabilities.
5. **Analyze Margin Distributions**:
   - Investigate the margin (mean margin metric) for a deeper understanding of decision boundary behavior across experiments.

By systematically addressing these recommendations, future iterations can achieve greater improvements and uncover additional insights into model dynamics OpenAI (2025).

S1.3. Testing

The codebase achieves a test coverage of 82%, as shown in Figure S1. The test coverage report indicates robust testing across critical modules, including `analysis.py`, `experiment_utils_mc.py`, `GUI.py`, `main.py`, and `model.py`. The testing framework ensures functionality verification through unit and integration tests.



Coverage report: 82%

coverage.py v7.6.10, created at 2025-01-06 17:00 -0500

File ▲	statements	missing	excluded	coverage
BinaryClassification/analysis.py	90	10	0	89%
BinaryClassification/experiment_utils_mc.py	291	52	0	82%
BinaryClassification/GUI.py	250	60	0	76%
BinaryClassification/main.py	169	34	0	80%
BinaryClassification/model.py	273	74	0	73%
BinaryClassification/tests/test_analysis.py	66	2	0	97%
BinaryClassification/tests/test_experiment_utils_mc.py	82	2	0	98%
BinaryClassification/tests/test_GUI.py	62	1	0	98%
BinaryClassification/tests/test_main.py	32	2	0	94%
BinaryClassification/tests/test_model.py	49	2	0	96%
Total	**1364**	**239**	**0**	**82%**

coverage.py v7.6.10, created at 2025-01-06 17:00 -0500

Fig. S1: Test coverage report for the codebase.

Experiment Configurations

Next to writing all these tests, we conducted multiple test runs of the software itself. The following experiments were conducted using various configurations to test the robustness and flexibility of the system:

Experiment 1: Baseline with AdamW Optimizer

We tested a standard configuration with: - Batch size: 64 - Learning rate: 0.01 - Number of epochs: 10 - Regularization parameters: $l2\_sum\_lambda = 0.001$, $l2\_mul\_lambda = 0.001$ - Weight normalization (wn): 0.9 - Depth and features normalization: Disabled - Batch normalization: Disabled - Bias: Disabled - Optimizer: AdamW

Experiment 2: Testing with SGD Optimizer

This experiment kept the same configuration as Experiment 1 but switched the optimizer to SGD for comparison. It allowed us to assess differences in convergence speed and stability between AdamW and SGD.

Experiment 3: Testing with Adam Optimizer

Using the same parameters as in the baseline (Experiment 1), we swapped the optimizer to Adam. This provided insights into how adaptive optimizers with different momentum implementations impact training dynamics.

Experiment 4: Enabling Bias Parameters

We explored the effect of enabling bias terms in the network: - Bias: Enabled - All other parameters identical to Experiment 1.

Experiment 5: Adding Batch Normalization

This experiment incorporated batch normalization into the model architecture: - Batch normalization: Enabled - All other parameters identical to Experiment 1.

Experiment 6: No Features Normalization

To examine the role of features normalization, this experiment disabled it: - Features normalization: None - All other parameters identical to Experiment 1.

Experiment 7: Enabling Depth Normalization

This configuration enabled depth normalization to study its impact: - Depth normalization: Enabled - All other parameters identical to Experiment 1.

Experiment 8: Increasing Batch Size

We increased the batch size to 128 while keeping the learning rate and other parameters the same: - Batch size: 128 - Regularization parameters: $l2\_sum\_lambda = 0.01$, $l2\_mul\_lambda = 0.01$ - All other parameters identical to Experiment 1.

Experiment 9: Higher Learning Rate

This experiment tested the impact of a higher learning rate: - Learning rate: 0.02 - All other parameters identical to Experiment 1.

Experiment 10: Grid Search with Multiple Regularization Parameters

A comprehensive grid search was conducted using multiple values for $l2\_sum\_lambda$ and $l2\_mul\_lambda$: - $l2\_sum\_lambda = [0.01, 0.001, 0.0001]$ - $l2\_mul\_lambda = [0.01, 0.001, 0.0001]$ - Number of epochs: 25 - All other parameters identical to Experiment 1.

Testing Summary

These experiments covered a wide range of configurations, including variations in optimizers, batch size, learning rate, normalization techniques, and regularization parameters. The generated reports for these configurations can be found here.