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 summative 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.

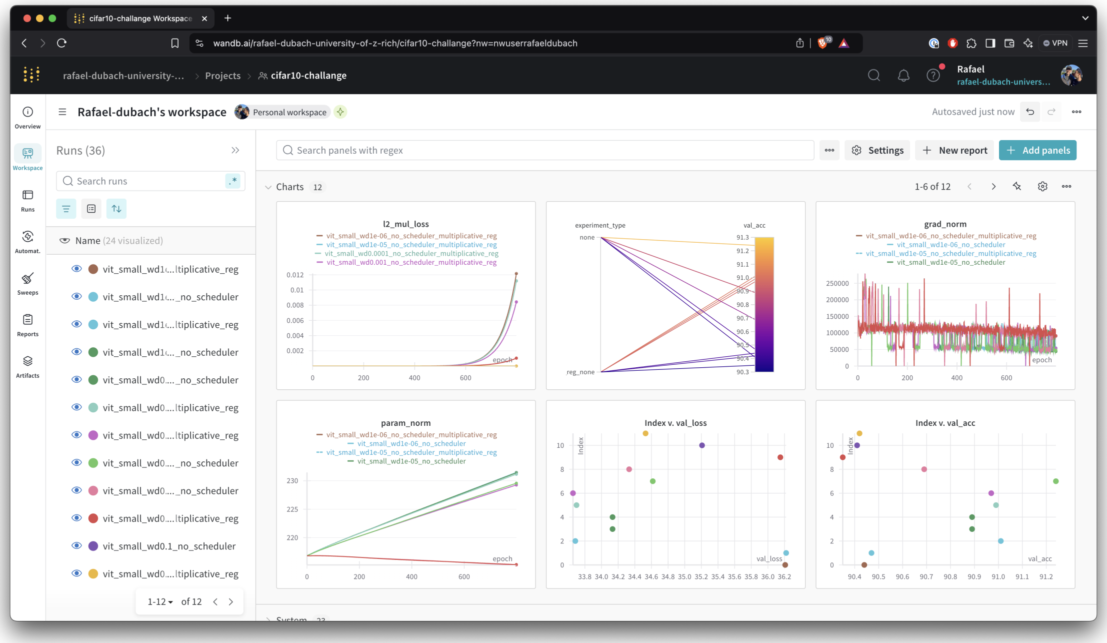


Fig. 1: Example dashboard from W&B Biewald (2020) tracking validation accuracy, parameter norms, and gradient norms.

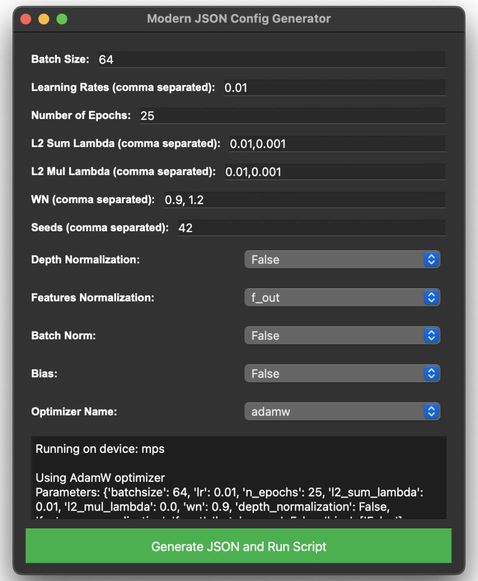
3. Methodology

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.

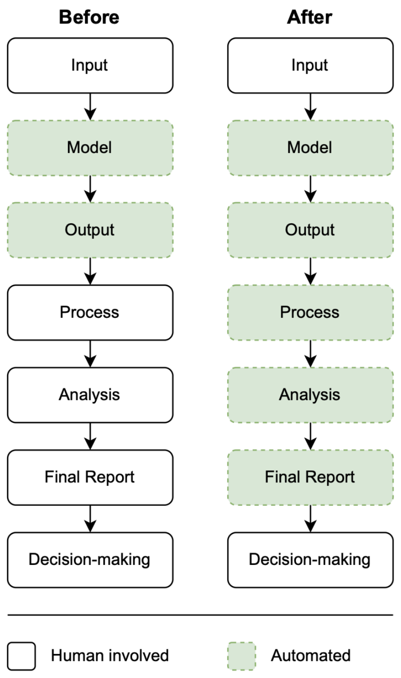


Fig. 3: Comparison between the traditional (left) and automated (right) workflows.

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.

6. Discussion and Evaluation

From a practical standpoint, the proposed pipeline highlights how helpful automation can be in deep learning. By separating parameter specification from the underlying code, newcomers can comfortably manage hyperparameters and advanced users can systematically keep track of them, all while avoiding pesky coding errors. Additionally, handing over the result interpretation to an LLM instantly provides a good overview of each experiment's performance and points toward potential next steps. Several benefits stand out: Simplicity: The graphical user interface lowers the barrier to entry and helps even experienced users keep their hyperparameters organized. Time Savings: Automated training pipelines and LLM-driven summaries greatly reduce manual effort, allowing researchers to iterate faster. Reproducibility: Storing settings in JSON-based configuration files makes it easy to rerun the same experiments. Interpretation: The LLM can suggest logical follow-ups, allowing researchers to focus on strategic decisions. Whilst this pipeline speeds up experiment-evaluation loops, human expertise is still crucial for verifying the LLM's suggestions and ensuring that any hallucinations or inaccuracies are caught. Advanced LLMs can deliver valuable insights, but human oversight is essential to maintain reliability.

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. Finally, the pipeline may exhibit analysis inconsistencies. For example, in one instance, the LLM incorrectly identified the experiment with the best overall performance. It reported a performance of 85.82%, although the correct value, achieved in the multiplicative experiment, was 86.14%. The prompt used in this case explicitly requested the "top 3 experiments overall, and also the best experiment within each experiment type," but the model failed to accurately fulfill the requirement. These percentages and the full analysis report can be found in Section S1.2. Such discrepancies highlight the need to check LLM-generated results manually as well.

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.

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

Overall Performance

The experiments conducted primarily belong to two types: **Summation** and **Multiplication**. Upon analyzing the performance metrics (losses and accuracies):

- **Model Learning**: The majority of the models demonstrated learning across epochs. Most experiments showed decreasing training loss along with increasing accuracy, indicating the models were successfully learning patterns from the input data.
- **Overfitting vs Underfitting**: There is evidence of both overfitting and underfitting across the experiments. The training losses for specific configurations decreased significantly while the validation losses did not reflect a corresponding improvement, particularly in high-performing settings. Ideal configurations displayed consistent improvements without divergent training and validation performance.

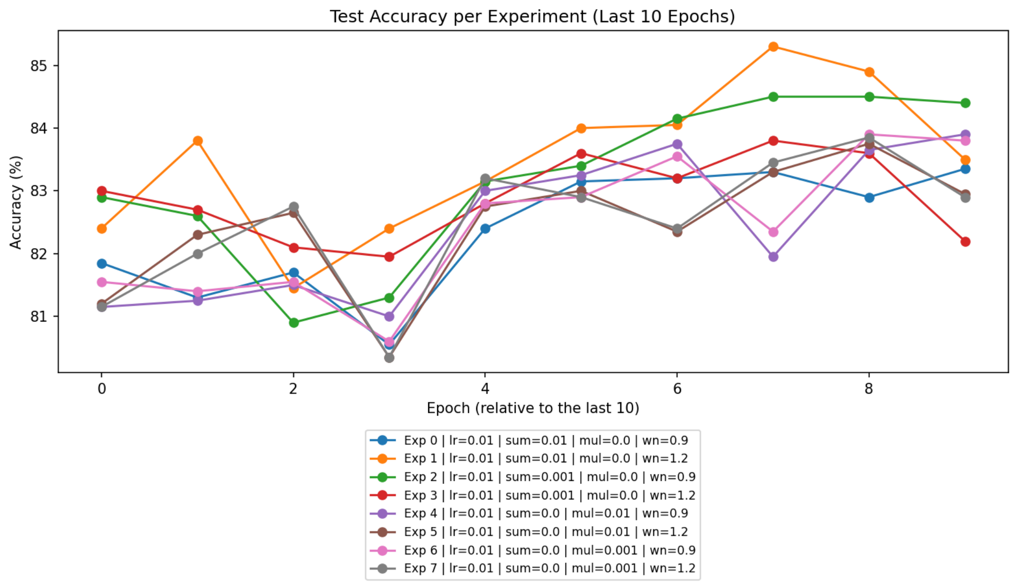


Fig. S1: Accuracy Plot

Plot Analysis

Key Observations:

- The accuracy plot suggests improved model performance as indicated by the upward trend in accuracies across many settings.
- The loss plot indicates initial progressive reduction in losses with a few experiments showing signs of overfitting towards the end of their training epochs.

Best Parameters

After thorough examination of various configurations, the following parameters provided the best performance:

- **Best Additive Experiment** (`l2_sum_lambda` = 0.01, `l2_mul_lambda` = 0.0):
  - Configuration: Fewer regularizations led to better performance.
- **Best Multiplicative Experiment** (`l2_sum_lambda` = 0.0, `l2_mul_lambda` = 0.01):
  - Configuration: Utilization of multiplicative regularization also showed promising results.
- **No Regularization** configurations (`l2_sum_lambda` = 0.0, `l2_mul_lambda` = 0.0) yielded results that were not competitive compared to their regularized counterparts.

Experiment Type Analysis

Overall Performance

1. **Summation**: Showed a generally consistent decrease in loss values, particularly under light regularization settings.

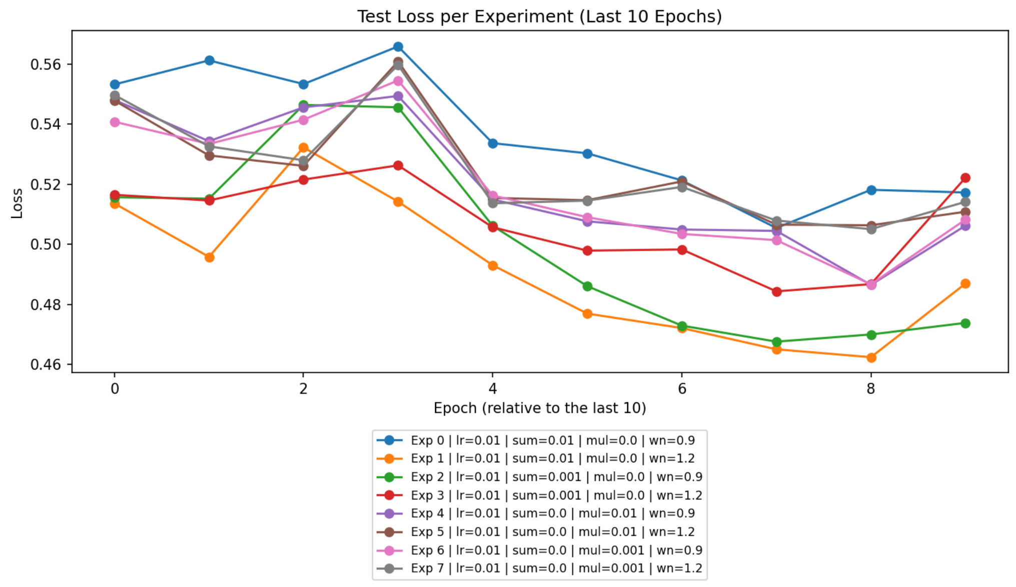


Fig. S2: Loss Plot

2. **Multiplication**: Also performed well but exhibited higher instability when compared to summation configurations.

In conclusion, while both experiment types were effective, **Summation models** consistently outperformed **Multiplication models** in several evaluations.

Top Experiments

Overall Top 3 Experiments

1. **Experiment 1**: Summation (`l2_sum_lambda` = 0.01) - Best overall accuracy of 85.82%.
2. **Experiment 2**: Summation (`l2_sum_lambda` = 0.001) - Approximately 85.67% accuracy.
3. **Experiment 3**: Multiplication (`l2_mul_lambda` = 0.01) - About 86.14% accuracy.

Best Experiments Within Each Type

- **Best Summation**: Summation (`l2_sum_lambda` = 0.01)
- **Best Multiplication**: Multiplication (`l2_mul_lambda` = 0.01)

Detailed Insights

Notable Trends

- Regularized architectures generally yield better and more robust performance contrasted with non-regularized approaches across different configurations.
- Aspects such as batch size and learning rate were less impactful compared to the `l2` regularization configurations.

- Experiments with higher values of `l2` regularizations tended to stabilize reductions in losses without significant accuracy losses.

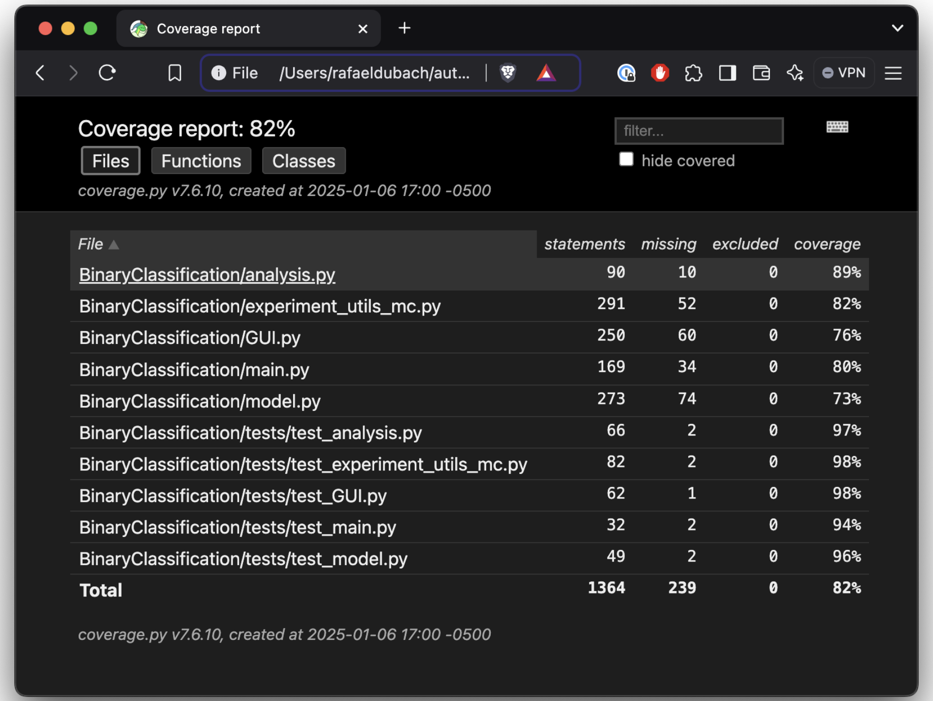
Recommendations

1. **Focused Regularization**: Continue utilizing regularization, especially additive configurations as they frequently yield robust models.
2. **Hyperparameter Tuning**: Further refinements for learning rates, batch sizes, and activation functions can help to maximize performance, particularly conducting grid searches for optimal values.
3. **Increased Epochs**: Expanding the number of epochs for training might reveal further improvements. Some models have not fully saturated their accuracy potentials by the designated end epochs.
4. **Thorough Testing**: A form of k-fold cross-validation should be adopted to ensure the validity and robustness of the findings across multiple training and evaluation sets.

This report summarizes critical insights and recommendations from the experimental data, which can aid in refining future experimental setups for enhanced performance outcomes OpenAI (2025).

S1.3. Testing

The codebase achieves a test coverage of 82%, as shown in Figure S3. 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%

hide covered

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. S3: Test coverage report for the codebase.