Empirical Validation of the Lion Optimizer's Performance on Machine Learning Tasks

John Angelou[1], Aly Dimoglou[1], Kevin Kim[1], Anton Mlynczyk[1], Osman Sultan[1]

[1]University of Toronto

Abstract - This project compares the performance of Lion and Adam optimizers across image classification and Natural Language Processing (NLP) tasks in resource-constrained settings. Experiments on MNIST and Kaggle Card Dataset with various pre-trained image classifiers, and MRPC paraphrase detection using transformer models, reveal domain-specific trends. In image classification, Lion often excelled with deeper architectures, while Adam showed advantages in specific shallow models or noisy dataset scenarios. For NLP fine-tuning, Lion consistently offered faster training throughput, but generalization performance relative to Adam proved highly model-dependent, with neither optimizer demonstrating universal superiority across the tested transformers. Results indicate optimizer selection should consider the interplay between model architecture, task specifics, and computational constraints rather than assuming universal superiority of either optimizer.

Code Repository with Implementation: https://github.com/osman-sultan/lion-optimizer

1 Introduction

Optimization algorithms are the backbone of modern deep learning, dictating both the speed of training and the ultimate performance of models. While adaptive optimizers like Adam [1] have dominated the field due to their robustness across tasks, recent work introduces Lion [2], a non-adaptive optimizer that claims to outperform Adam in accuracy and memory efficiency. However, Lion's advantages have primarily been validated on large-scale datasets (e.g., ImageNet, JFT-300M) and computationally intensive architectures like Vision Transformers (ViTs). This raises a critical question: *Do Lion's benefits persist in resource-constrained settings, such as smaller datasets or simpler models?*

In this project, we evaluate the performance of Lion and Adam for transfer learning across both image classification and Natural Language Processing (NLP) tasks. For image classification, we use two datasets of varying complexity: MNIST (70k grayscale digits) [3] and the Kaggle Card Dataset (8k playing card images) [4], fine-tuning pre-trained models (e.g., ResNet, EfficientNet, SqueezeNet) by freezing 90% of their parameters and training the remaining layers for both optimizers. With respect to NLP, several pre-trained transformer-based language models (e.g., BERT, ALBERT, and DistilBERT) were fine tuned on the Microsoft Research Paraphrase Corpus (MRPC) from the GLUE benchmark [5]. This NLP task tests whether Lion's benefits extend to language understanding. Our experiments aim to answer three key questions:

1) Does Lion consistently outperform Adam on smaller, simpler classification tasks?
2) How does dataset and/or model complexity (e.g., class count, image diversity, sequence length) influence optimizer effectiveness?
3) Are there domain-specific advantages to either optimizer when comparing performance on different tasks (image classification and NLP)?

This investigation is motivated by practical considerations. While large-scale benchmarks demonstrate Lion's potential, real-world applications often involve limited data or hardware. For instance, our Card Dataset – with 53 classes and variations in lighting/angles – represents a situation where deploying ViTs or training from scratch is infeasible or unworthwhile. By testing Lion on these tasks, we provide insights into its suitability for scenarios where computational resources or data availability are constrained. It must be noted that more emphasis in this experiment has been put on image classification due to time and resource constraints, but the preliminary results of the optimizers' performances on NLP tasks are still insightful nevertheless.

2 Related Work

Work done in this project builds on foundational research in adaptive optimizers and automated algorithm discovery, with a focus on the Lion optimizer [2] and its comparisons to Adam [1]. Below, we contextualize these works within the broader landscape of optimization research.

The Adam optimizer revolutionized deep learning by combining momentum and adaptive learning rates, making it a default choice for tasks ranging from image classification to language modeling. Its variant, AdamW [6], later decoupled weight decay regularization to improve generalization, further solidifying Adam's dominance. Recent works like Adafactor [7] and AdaBelief [8] refined adaptive methods by addressing memory constraints and gradient uncertainty, but none fundamentally deviated from Adam's adaptive framework. Lion challenges this paradigm by replacing adaptive learning rates with a sign-based update rule, achieving comparable performance with reduced memory overhead [2].

The Lion optimizer was discovered through symbolic program search [2], a methodology inspired by Real et al. through AutoML-Zero [9]. Unlike earlier attempts to design optimizers via reinforcement learning [10] or meta-learning [11], symbolic search explores an infinite space of mathematical operations to derive algorithms from scratch. AutoML-Zero demonstrated this approach on simple tasks, but Lion scales it to state-of-the-art benchmarks like ImageNet and language modeling. Lion's ultimate finding was also based on the prioritization of simplifying algorithms [2].

Lion's design shares a key idea with the work of Bernstein et al. introducing SignSGD, an optimizer that uses the sign of gradients (positive/negative direction) to update model parameters efficiently [12]. However, Lion improves on this concept by incorporating momentum – a smoothed average of past gradients – to determine the update direction instead of relying on raw gradients. This subtle change stabilizes training and helps Lion avoid the pitfalls of SignSGD, which struggles on complex tasks like image classification. While SignSGD was designed for distributed systems with limited communication, Lion's momentum-based approach enables reliable performance in standard training setups, even matching or exceeding Adam's accuracy on benchmarks like ImageNet [2].

Our project extends the empirical analysis of Lion by evaluating its performance on smaller-scale datasets and simpler architectures. While the work by Chen et al. focused on large-scale tasks like ImageNet and JFT-300M, our work investigates whether Lion's advantages hold in resource-constrained settings – a question implicitly raised by Chen et al.'s observation that Lion's gains correlate with batch size and model complexity [2]. By comparing Lion to Adam under similar constraints, we contribute to the broader understanding of when and why non-adaptive optimizers outperform adaptive ones.

3 Datasets

3.1 Image Classification

As discussed, we evaluated our models and optimizers on two image classification datasets: MNIST [3] and a Kaggle Card Dataset [4]. Image classification is a well-established benchmark task that allows us to test the performance of different optimization techniques in a structured and measurable way. MNIST consists of 70,000 grayscale images of handwritten digits (50,000 for training) across 10 classes [3]. The Card Dataset contains 8,154 images (7,624 for training) across 53 classes, each representing different playing card types [4]. This dataset provides a more complex classification challenge due to the larger number of classes and variations in image patterns.

For these datasets, we applied several transformations using the *transforms.Compose()* function from the PyTorch library to enhance model generalization and improve robustness to variations in input data. Details regarding these transformations are included in Appendix A. While MNIST consists of grayscale images that do not require color-based augmentations, the Card Dataset benefits significantly from colour jittering and flipping to introduce variation in image conditions.

3.2 Natural Language Processing

In addition to image classification, we expanded our evaluation to include NLP tasks using the GLUE benchmark [5]. Specifically, we utilized the MRPC from the GLUE benchmark, which consists of sentence pairs drawn from news sources, along with human annotations indicating whether the sentences in each pair are semantically equivalent [13]. The MRPC dataset contains 3,668 training examples and 1,725 test examples, providing a focused task for evaluating optimizer performance in the NLP domain [13]. The MRPC dataset represents a binary classification task where each example contains two sentences and a label indicating whether they are paraphrases of each other [14]. This task challenges models to understand semantic similarity across different sentence structures and vocabulary choices. While we initially considered also evaluating models on the Stanford Question Answering Dataset (SQuAD) in our analysis, resource constraints made it impractical to train on this and other larger datasets.

For the MRPC dataset, we employed standard text preprocessing techniques through the Hugging Face Datasets library [15]. This included tokenization with appropriate special tokens, padding, and truncation to ensure consistent input dimensions. Details of the text preprocessing pipeline are included in Appendix A. The combination of these different dataset types and preprocessing techniques allows us to evaluate optimizer performance across diverse machine learning domains.

4 Methods

Rather than solving a problem that is modelled by mathematical programming methods, our work involves empirically demonstrating whether the Lion optimizer outperforms other benchmarked optimizers in terms of accuracy/loss and training time when used for machine learning. Since the objective of this project is to reveal details about Lion's superior usage and the specific scenarios in which this claim holds, the methods outlined does not require any modelling but rather an extensive

amount of experimentation with a variety of data and tasks. In order to facilitate this, transfer learning was conducted with a variety of pre-trained models for both image classification and NLP tasks.

For both tasks, the key metrics that were collected include training time and accuracy/loss. Training time indicates the total number of seconds it takes for a model to train its weights for all epochs. This was measured using the time library in Python for image classification [16]. For NLP, a different training time metric was extracted from Hugging Face's built-in training functionalities [17]. In terms of accuracy, for image classification, values were observed for the training and validation datasets by calculating the number of images that were predicted to have the correct label out of the total number of images. For NLP, the loss values for the training and validation datasets and accuracy against validation data were obtained to compare each optimizer. Hugging Face's Trainer was used to compute and display these values [17]. The loss function used was Cross Entropy due to the nature of the task [18]. To calculate accuracy for NLP, the "accuracy_score" function from the scikit-learn library was used and passed into the Trainer object [19].

The goal of this experiment was to find the optimizer (Lion or Adam) that is able to achieve the shortest training time as well as maximize (or minimize) accuracy (or loss). This was done by training various pretrained models with the aforementioned datasets, and comparing the training time and accuracy/loss values when using Lion and Adam. The result of each training instance was collected to perform a comparative analysis between the optimizers. As well, for the image classification tasks – since it had less computational demand than the NLP task – hyperparameter tuning was conducted to identify a set of parameters that demonstrates the shortest training time and best accuracy/loss. Below, we detail the intricacies of the machine learning tasks that were conducted and outline the team's approach that was taken to achieve the goal of this project.

4.1 Image Classification

The first machine learning task experimented on was image classification to verify and build upon the experimental findings of Chen et al. [2]. There was no discrimination with regard to any specific field of image classification; we simply wanted to evaluate whether deep learning models experience a performance improvement when using Lion instead of Adam for a variety of use cases (enabled by the distinct dataset selection). As previously mentioned, the two selected datasets for image classification have different numbers of classes and contain completely distinct images, which serve our purpose well. Rather than constructing our own deep learning model - since that is not the objective of this project - we employed transfer learning by using a variety of pre-trained models available through PyTorch [20]. Namely, SqueezeNet, EfficientNet-B0, AlexNet, ResNet18, ResNet34, and ResNet50 were selected [20]. Given the available resources, we took on a different approach for training the models by setting limits on our experiment for this task - which is further detailed in Section 5. The overview of the training mechanism involves training for a subset of the model's parameters by passing the model, optimizer type, training and validation dataset into a training function. The training function was iterative, in that batches of the training data were explored for each epoch, and the model conducted a forward pass, loss calculation, and backward propagation for each batch. Additionally, after each sweep of the training dataset, the accuracy against the training and validation dataset was calculated and stored for plotting. The specific setup of the training environment, parameters, as well as the final results are described in Section 5.1. The code that was executed for this experiment is also provided in the "ImageClassification.py" file in the attached code repository.

4.2 Natural Language Processing

We then further expanded our scope to NLP tasks, specifically on text similarity. The motivation behind this was to test if there are domain-specific advantages to either optimizer (Adam or Lion) by comparing performance when training on data involving text (instead of solely comparing based on image classification). To test this, we took on a similar approach by loading several pre-trained models and training them on new data. With the aforementioned dataset and using Hugging Face, the distilbert-base-cased [21], albert-base-v1 (uncased) [22], and bert-base-cased [23] models were retrieved and trained. Cased models indicate pre-trained models that distinguish capitalized and lowercase letters, whereas uncased models do not [21][22][23]. Selecting these three diversified and differently trained models provide more opportunity for exploration with this NLP task and can help identify patterns and scenarios where Lion may outperform Adam. The setup was less involved since the data preparation and training were conducted using various classes offered by Hugging Face [17][24]. The dataset was tokenized using the AutoTokenizer module which identifies and creates the proper tokenizer for the inputted pre-trained model type [24]. Training was conducted using the Trainer object provided by Hugging Face [17] – which removed the need to implement our own training algorithm. However, parameters still needed to be selected, and this process is outlined further in Section 5.2. The code for this experiment is provided in the "nlp.py" file in the attached repository.

5 Experiments and Results

5.1 Image Classification

As mentioned, we focus on six pre-trained convolutional neural networks: AlexNet, EfficientNet-B0, ResNet18, ResNet34, ResNet50, and SqueezeNet, implemented in PyTorch. For each model, we freeze 90% of the pre-trained weights to simulate resource constrained fine-tuning, retaining only the final classification layers (10% of parameters) for training on the new datasets. This approach mirrors practical scenarios where full model retraining is computationally prohibitive. Hyperparameters were systematically tested under hardware constraints. Learning rates (η) spanned {10e-4,

10e-3, 10e-2, 10e-1} to explore scenarios ranging from conservative to aggressive updates. Batch sizes {16, 32, 64, 128} were selected to balance gradient stability and memory limits, while weight decay (λ) values {10e-2, 10e-3} probed the impact of regularization strength. Due to limited GPU resources available for this task, we conducted partial sweeps: first, fixing λ = 0.01 and epochs = 10 to isolate the effects of η and batch size, then fixing batch size = 16 and η = 0.001 to examine interactions between λ and training duration (5 vs. 10 epochs). This staggered approach maximized insights from available computing resources. All models were trained locally on an NVIDIA RTX 4070 (12GB VRAM) GPU using PyTorch with CUDA acceleration. To ensure deterministic comparisons, we allocated exclusive GPU access during training by suspending non-essential background processes. The train-validation-test split varied slightly across datasets due to their inherent structure, which is detailed in Section 3. For the MNIST Torch dataset the split was: Train set: 50,000 examples, Validation set: 10,000 examples, Test set: 10,000 examples [3]). The split for the Kaggle cards dataset consisted of 7,624 images for training, 265 images for validation, and 265 images for testing [4]. As mentioned, training time was measured as the time to complete all training epochs, excluding data loading and preprocessing, and validation accuracy was evaluated on held-out test sets after each epoch to capture trends.

The experimental results, as summarized by the tables in Appendix B and the graphs in Appendix C, demonstrate a nuanced performance landscape for the Lion and Adam optimizers across architectures and datasets. On the **MNIST dataset**, Lion consistently matched or surpassed Adam in both peak and average accuracy, particularly excelling with deeper models. For instance, **ResNet50** achieved a peak accuracy of **97.3%** with Lion compared to Adam's **96.2%,** while maintaining similar training times (~844–858 seconds). Similarly, **EfficientNet-B0** under Lion reached **97.4%** accuracy, outperforming Adam's **96.6%**. Even in cases where Lion's peak accuracy slightly trailed Adam, such as with **AlexNet** (**90.7%** vs. **91.3%**), it exhibited significantly higher average accuracy (**82.2%** vs. **71.3%**), underscoring its robustness to suboptimal hyperparameters. Notably, **SqueezeNet** saw a stark divergence: Lion dominated on MNIST (**95.5%** vs. Adam's **94.8%**) but collapsed entirely on the Card Dataset, a trend we explore further below. Training times remained comparable across optimizers, with Lion's memory efficiency not directly translating to faster training time but reducing hardware strain (e.g., ResNet34: Lion **691.6s**, Adam **711.2s**).

On the more complex **Kaggle Card Dataset** (53 classes), results were mixed. Lion excelled with deeper architectures like **ResNet50**, achieving **83.4%** peak accuracy (vs. Adam's **81.9%**) and a **12.1%** higher average accuracy, while training marginally faster (**299.9s** vs. **306.3s**). However, Adam outperformed Lion in specific cases, such as **ResNet34**, where it reached **77.4%** peak accuracy (vs. Lion's **74.0%**) under a low learning rate (η = 0.0001) and shorter training (5 epochs). This suggests adaptive methods like Adam may better handle noisy gradients in the early stages of fine-tuning. Shallow models revealed stark contrasts: **AlexNet** favored Adam (**57.0%** vs. Lion's **46.8%** peak accuracy), while **SqueezeNet** catastrophically failed with Lion (**10.9%** vs. Adam's **32.1%**), aligning with the original Lion paper's observation that its benefits diminish in models with limited capacity [2].

5.1.1 Hyperparameter Analysis & Architectural Compatibility

The interplay between hyperparameters and optimizer performance revealed critical insights, with the most impactful hyperparameter appearing to be the learning rate (η). Lion achieved peak performance on the Card Dataset with moderate learning rates (η=0.001) for deeper models like ResNet50 (**83.4%** accuracy), while Adam excelled with smaller rates (η=0.0001) for the same architecture (**81.9%** accuracy). However, Lion's sensitivity to learning rate was pronounced in shallow models: SqueezeNet collapsed to **10.9%** accuracy even with η=0.0001, whereas Adam tolerated higher rates (η=0.001) to achieve **32.1%** accuracy. On MNIST, Lion's robustness shone through, delivering strong results across a wider range of η (e.g., ResNet50: **97.3%** at η=0.0001). Interesting insights also arose from analyzing weight decay values (λ). Lion's sign-based updates, which produce larger effective step sizes, necessitated stronger regularization on complex tasks. For example, ResNet50 on the Card Dataset achieved peak accuracy with λ=0.1, whereas Adam performed best at λ=0.01. This suggests Lion's updates amplify the impact of weight decay, requiring careful tuning to avoid underfitting or overfitting.

Another important inference can be made with respect to the architectural compatibility of the optimizers. The stark contrast in SqueezeNet's performance in particular underscores the importance of model-optimizer alignment. SqueezeNet's lightweight design, optimized for parameter efficiency [25], clashed with Lion's reliance on momentum-driven sign updates. On the Card Dataset, Lion's peak accuracy (**10.9%**) trailed Adam's (**32.1%**), suggesting that shallow architectures lack the capacity to leverage Lion's global update strategy. On the other hand, Adam's adaptive learning rates proved advantageous in low-capacity settings. For example, AlexNet on the Card Dataset achieved **57.0%** accuracy with Adam (vs. Lion's **46.8%**), as Adam's per-parameter scaling likely mitigated gradient noise in early training phases.

5.2 Natural Language Processing

We evaluate the performance of Lion and Adam on the MRPC paraphrase detection task from the GLUE benchmark. In this task, the objective is to determine whether two sentences are semantically equivalent by seeing if sentence 1 is a good paraphrasing of sentence 2 and vice versa. The train-validation-test split for the NLP data was also inherent in the GLUE MRPC dataset, which consisted of 3,668 examples for training, 408 examples for validation, and 1,725 examples for testing, following the standard distribution of the Microsoft Research Paraphrase Corpus benchmark [13]. We fine-tuned three pre-trained transformer-based language models (BERT, ALBERT, and DistilBERT) for 10 epochs (approximately

2300 iterations in total) using identical hyperparameters (learning rate η = 1e-4, weight decay λ = 1e-2, per-device training batch size of 16, evaluation batch size of 64, and 500 warmup steps) to ensure a fair comparison between the optimizers. All models were fine-tuned in their entirety (no parameters were frozen), and all experiments were conducted on a personal PC equipped with an RTX 3060ti. This experimental setup reflects a resource-constrained environment typical of many practical applications.

For BERT, the evaluation accuracy obtained with Lion was **78.19%** with an evaluation loss of **0.5278**, whereas Adam achieved an evaluation accuracy of **74.51%** with an evaluation loss of **0.5358**. In addition, training speed differed noticeably between the optimizers: with Lion, BERT processed about **1.55** iterations per second, corresponding to roughly **24 minutes and 44 seconds** for 10 epochs; in contrast, Adam achieved about **1.10** iterations per second, which translates to about **34 minutes and 51 seconds**.

For ALBERT, Lion yielded an evaluation accuracy of **68.38%** with an evaluation loss of **0.6234**, while Adam reached an evaluation accuracy of **80.64%** with an evaluation loss of **0.3991**. The training speed for ALBERT was about **1.75** iterations per second with Lion (roughly **21 minutes and 54 seconds** for 10 epochs) and about **1.65** iterations per second with Adam (approximately **23 minutes and 14 seconds**).

For DistilBERT, Lion achieved an evaluation accuracy of **75.00%** with an evaluation loss of **0.5271**. In comparison, Adam produced an evaluation accuracy of **72.55%** with an evaluation loss of **0.5636**. Training speed for DistilBERT was nearly **3.19** iterations per second with Lion (approximately **12 minutes and 1 second** for 10 epochs) and about **3.06** iterations per second with Adam (roughly **12 minutes and 32 seconds**).

Table D1 in Appendix D summarizes these results in detail, including evaluation metrics and training durations.

5.2.1 Comparative Insights

Generalization: Within this specific fine-tuning setup (10 epochs, fixed hyperparameters), the relative generalization performance varied by model architecture. Lion achieved higher validation accuracy for BERT and DistilBERT, despite significantly higher final training loss and apparent lack of convergence (see Appendix E). Adam, conversely, achieved substantially better validation results for ALBERT. This suggests that under these constrained conditions, the interaction between optimizer dynamics and model architecture heavily dictates generalization outcomes.

Convergence Dynamics: Adam consistently achieved lower final training loss values, suggesting more effective minimization of the training objective within the limited training budget. In contrast, Lion's training loss curves (Appendix E) indicate it did not converge within the 10 epochs for any model, stabilizing at comparatively high loss values. This highlights differing convergence rates and behaviors under the chosen hyperparameters and epoch limit.

Resource Efficiency: Lion consistently exhibited higher training throughput (iterations/second) than Adam across all architectures in this experiment. The efficiency gain was most notable for BERT. This observed speed advantage makes Lion potentially favorable when computational time is limited, though its convergence properties warrant consideration.

5.2.2 Limitations

A primary limitation of these NLP experiments stems from resource constraints, which precluded extensive hyperparameter tuning, unlike the approach potentially feasible for simpler tasks like image classification [Reference if applicable, otherwise remove]. Consequently, a single set of fixed hyperparameters (learning rate, weight decay, batch sizes) was applied across all model and optimizer combinations. This non-optimized setup means the observed performance differences may not reflect the full potential of each optimizer when properly tuned for a specific architecture. Furthermore, the fixed training duration of 10 epochs may be insufficient for full convergence, particularly for optimizers like Lion, whose training loss curves (Appendix E) did not appear to reach a plateau.

Therefore, while Lion consistently demonstrated faster training throughput in our tests, conclusions about its relative generalization ability compared to Adam are tentative and highly dependent on the specific model and the constrained experimental conditions. The mixed validation results (Lion superior for BERT/DistilBERT, Adam superior for ALBERT) underscore that performance is sensitive to these factors. Further investigation with dedicated hyperparameter optimization and potentially longer training runs for each model-optimizer pair is necessary to draw more definitive conclusions about their relative merits on these NLP tasks

5.3 Takeaways

We revisit the three main questions asked in the introduction of this paper:

1) **Does Lion consistently outperform Adam on smaller, simpler classification tasks?**

Yes, Lion demonstrated superior performance on simpler tasks like MNIST, particularly with deeper architectures. Lion's robustness to hyperparameter variations also translated to higher average accuracy across configurations, making it a reliable choice for smaller, well-structured datasets. In our NLP fine-tuning experiments under fixed hyperparameters, the results were inconsistent. While Lion achieved better validation accuracy for the larger BERT and mid-sized DistilBERT models despite its higher final training loss, Adam substantially outperformed Lion when fine-tuning the much smaller ALBERT model. Lion did, however, consistently achieve faster training throughput (iterations/sec) across all NLP models tested. These

observations highlight that relative generalization performance between the optimizers in this NLP setting is highly model-dependent and sensitive to the specific experimental conditions.

2) How does dataset and/or model complexity influence optimizer effectiveness?

Dataset and model complexity significantly shape optimizer performance. Lion excelled with deeper models (e.g., ResNet50) and moderate-complexity tasks (MNIST), where its momentum-driven sign updates and memory efficiency shine. However, on the complex Card Dataset, Adam outperformed Lion in shallow architectures, highlighting its adaptive learning rates' ability to mitigate noisy gradients. Notably, SqueezeNet's catastrophic failure with Lion underscored the incompatibility of Lion's global update strategy with lightweight, low-capacity models. In the NLP domain, Adam performed best on the smallest model (ALBERT), whereas Lion showed better results on the larger BERT and the mid-sized DistilBERT. This suggests that within this specific NLP fine-tuning context, there isn't a straightforward correlation between model size/parameter count and the optimal optimizer choice. This further reinforces the highly model-specific nature of the results under these constraints.

3) Are there domain-specific advantages to either optimizer when comparing performance on different tasks (image classification and NLP)?

Comparing across domains, a potential domain-specific advantage observed for Lion in NLP was its consistent edge in training throughput (iterations/second) across all three tested transformer models. While its validation accuracy performance relative to Adam was mixed and highly model-dependent (better for BERT/DistilBERT, worse for ALBERT), this consistent speed advantage suggests Lion might offer efficiency benefits specifically within this NLP fine-tuning context under the tested constraints. In contrast, Lion's advantages in image classification were more conditional on factors like model architecture depth (excelling with deeper models) or showed specific incompatibilities (SqueezeNet), and a consistent speed advantage across all CV scenarios wasn't highlighted as a primary finding in the same way. Therefore, while generalization performance remained highly model-dependent in both domains, Lion demonstrated a consistent efficiency advantage in our NLP experiments that wasn't as uniformly observed or emphasized in the image classification results.

6 Conclusion and Future Work

This work investigated the performance of the Lion and Adam optimizers across varying model architectures and dataset complexities, addressing three core questions, which we now have clearer answers to. We found that Lion's strengths lie in its memory efficiency and hyperparameter robustness, making it ideal for fine-tuning deeper networks on tasks like MNIST. However, its limitations in shallow architectures emphasize the need for careful optimizer-architecture pairing. Conversely, Adam's adaptive learning rates provide stability for low-capacity models or noisy, small-scale datasets but incur higher memory costs.

Concerning NLP fine-tuning, our findings under resource-constrained conditions revealed different trends. Lion consistently demonstrated faster training throughput across the tested transformer models. However, its relative generalization performance compared to Adam was highly model-dependent, with no clear winner emerging across all architectures. Furthermore, Lion exhibited slower convergence in terms of training loss within the limited training epochs compared to Adam. These results underscore that in NLP fine-tuning under such constraints, the choice between these optimizers involves a trade-off between Lion's potential speed benefits and Adam's generally more effective training loss reduction, with final validation performance being sensitive to the specific model architecture.

Therefore, selecting the optimal optimizer requires careful consideration of the specific context. For image classification, a hybrid approach might be suitable based on model depth and dataset characteristics as previously discussed. For NLP fine-tuning under constraints similar to ours, the choice involves weighing Lion's consistent speed advantage against its variable generalization outcomes and slower training loss convergence. Adam may be preferred for its more predictable convergence on the training objective, but model-specific testing is crucial to determine which optimizer yields better validation performance for a given transformer architecture and task. These findings underscore the importance of context-driven optimizer selection, balancing architectural constraints, task specifics, available training time, and computational resources.

Given more resources and time, there are a few efforts that would be ideal extensions to our current project. First, our team would have liked to expand our comparison and analysis across 8 more optimizers (comparing 10 total), and extend the machine learning use case to tasks related to image/text generation. Secondly, it would be interesting to define an optimization problem where the objective function aims to minimize loss, and the hyperparameters such as momentum and learning rate are set as decision variables. The optimal solution to this problem may eliminate the need for hyperparameter tuning and may uncover insights that can help develop a new optimizer (perhaps one we can call "424-Optimizer").

Overall, this project was valuable in emphasizing the existing limitations in machine learning regarding parameter optimization and the manual burden of this task. It would not be surprising to see new discoveries and developments in which hyperparameter tuning and optimizer selection become obsolete in the near future.

7 Contributions of Team Members

Osman Sultan
Report: Wrote 5.2 NLP and added NLP insights/background to intro and conclusion

Presentation: Created and presented our goal slides and NLP results

Code: Setup entire NLP pipeline, and created GitHub Repo for project, and trained on my personal PC

Kevin Kim
Report: Wrote the Methods section (Section 4) and Future Work section (Section 6). Proofread the entire document, revised all sections for better flow and corrections where necessary.

Presentation: Created the entire initial slide deck and template for teammates to populate with data/results. Created slides for and presented the first two sections of the presentation (problem introduction and reasons for its relevance).

Coding/Training: Found Card dataset. Conducted initial training for SqueezeNet on MNIST and Card data for both optimizers (Adam and Lion). Conducted training for ALBERT on GLUE for both optimizers.

John Angelou
Report: Wrote the Experiments and Results for image classification (5.1). Proofread and edited the document.

Code: Conducted all training / hyperparameter tuning / data aggregation for image classification. Analysed the data eventually delivering the results seen in the report.

Presentation: Created and Presented the image classification results slides using the data / charts collected during the training phase.

Anton Mlynczyk
Report: Wrote Introdution (section 1), Related Work (section 2), Experiments & Results for Image classification (sections 5.1 and 5.1.1), the first 2 parts of the Takeaways (section 5.3), Conclusion (section 6). Proofread and edited entire document.

Presentation: Created and presented slides for Related Work and Next Steps

Code/Training: Performed initial testing of EfficientNet on MNIST and Cards datasets for both optimizers.

Aly Dimoglou
Report**:** Wrote the Abstract, Dataset Section (Section 3), answered Question 3 in Takeaways (section 5.3), discussion of train-test split in results, and contributed to the introduction (Question 3 discussion). Proofread the entire document, commented, and revised comments on my sections.

Coding/Training: Conducted training for AlexNet (pretrained on imagenet) on MNIST and Card data. Conducted training for DistilBERT on GLUE MRPC dataset and tested SQuAD and BabyLM, finding they were too big to be trained/tested with our resources.

8 References

[1] D. P. Kingma and J. L. Ba, *Adam: A Method for Stochastic Optimization*, Dec. 2014. doi:10.48550/arXiv.1412.6980

[2] X. Chen *et al.*, *Symbolic Discovery of Optimization Algorithms*, Feb. 2023. doi:10.48550/arXiv.2302.06675

[3] "MNIST," PyTorch, https://pytorch.org/vision/main/generated/torchvision.datasets.MNIST.html (accessed Mar. 10, 2025).

[4] Gerry (gpiosenka), "Cards Image Dataset-Classification," Kaggle, https://www.kaggle.com/datasets/gpiosenka/cards-image-datasetclassification (accessed Mar. 10, 2025).

[5] "GLUE", Huggingface, https://huggingface.co/datasets/nyu-mll/glue (Accessed: April 07, 2025)

[6] I. Loshchilov and F. Hutter, *Decoupled Weight Decay Regularization*, Nov. 2017. doi:10.48550/arXiv.1711.05101

[7] N. Shazeer and M. Stern, *Adafactor: Adaptive Learning Rates with Sublinear Memory Cost*, Apr. 2018. doi:10.48550/arXiv.1804.04235

[8] J. Zhuang *et al.*, *AdaBelief Optimizer: Adapting Stepsizes by the Belief in Observed Gradients*, Oct. 2020. doi:10.48550/arXiv.2010.07468

[9] E. Real, C. Liang, D. R. So, and Q. V. Le, *AutoML-Zero: Evolving Machine Learning Algorithms From Scratch*, Mar. 2020. doi:10.48550/arXiv.2003.03384

[10] I. Bello, B. Zoph, V. Vasudevan, and Q. V. Le, *Neural Optimizer Search with Reinforcement Learning*, Sep. 2017. doi:10.48550/arXiv.1709.07417

[11] M. Andrychowicz *et al.*, *Learning to learn by gradient descent by gradient descent*, Jun. 2016. doi:10.48550/arXiv.1606.04474

[12] J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar, *signSGD: Compressed Optimisation for Non-Convex Problems*, Feb. 2018. doi:10.48550/arXiv.1802.04434

[13] "MRPC, Huggingface, https://www.tensorflow.org/datasets/catalog/glue (Accessed: April 07, 2025)

[14] Papers with Code, "MRPC (Microsoft Research Paraphrase Corpus)," https://paperswithcode.com/dataset/mrpc (Accessed: April 07, 2025)

[15] Hugging Face, "Quickstart — Datasets Documentation v1.13.0," https://huggingface.co/docs/datasets/v1.13.0/quickstart.html (Accessed: April 07, 2025)

[16] "time — Time access and conversions," Python documentation, https://docs.python.org/3/library/time.html (accessed Apr. 6, 2025).

[17] "Trainer," Hugging Face, https://huggingface.co/docs/transformers/en/trainer (accessed Apr. 6, 2025).

[18] Hugging Face, "transformers/src/transformers/models/bert/modeling_bert.py at 94B3F544A1F5E04B78D87A2AE32A7AC252E22E31 · Hugging Face/Transformers," GitHub, https://github.com/huggingface/transformers/blob/94b3f544a1f5e04b78d87a2ae32a7ac252e22e31/src/transformers/models/bert/modeling_bert.py#L1548 (accessed Apr. 6, 2025).

[19] "accuracy_score," scikit-learn, https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html (accessed Apr. 6, 2025).

[20] "Models and pre-trained weights," PyTorch, https://pytorch.org/vision/main/models.html (accessed Apr. 6, 2025).

[21] "distilbert/distilbert-base-cased," Hugging Face, https://huggingface.co/distilbert/distilbert-base-cased (accessed Apr. 6, 2025).

[22] "albert/albert-base-v1," Hugging Face, https://huggingface.co/albert/albert-base-v1 (accessed Apr. 6, 2025).

[23] "google-bert/bert-base-cased," Hugging Face, https://huggingface.co/google-bert/bert-base-cased (accessed Apr. 6, 2025).

[24] "Auto Classes," Hugging Face, https://huggingface.co/docs/transformers/model_doc/auto (accessed Apr. 6, 2025).

[25] F. N. Iandola *et al.*, *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size*, Feb. 2016. doi:10.48550/arXiv.1602.07360

Appendix A: Transformation Details

Image Classification Processing

- **transforms.ToTensor()**: Converts the images into PyTorch tensors, which are required for model training. This transformation was applied for both the MNIST and Cards Dataset.
- **transforms.Resize((224, 224))**: Resizes all images to **224x224 pixels**, which is the standard input size for many pre-trained models (on imagenet). MNIST images are originally **28x28**, so resizing ensures they can be processed by deeper networks.
- **transforms.Grayscale(num_output_channels=3)**: Converts grayscale images to **three channels (RGB format)**. Since MNIST images are originally **grayscale**, this transformation allows them to be processed by models designed for colour images, such as those trained on ImageNet.
- **transforms.ColorJitter()**: Introduces random variations in brightness, contrast, saturation, and hue. This helps the model become more invariant to lighting conditions and colour distortions. This transformation was only applied to the Card Dataset, where card colours may vary due to different lighting environments. This was not applied to the MNIST dataset, as it consists of grayscale images that do not require colour-based augmentations.
- **transforms.RandomHorizontalFlip()** & **transforms.RandomVerticalFlip()**: Randomly flips images horizontally and vertically with a 50% probability. This augmentation increases the Card Dataset's diversity, preventing the model from overfitting to specific orientations of the images. Since digit orientation is important for the MNIST dataset, these transformations were not applied.
- **transforms.Normalize(mean, std)**: Normalizes the image pixel values to match the standard normalization used in pre-trained models. This ensures that the input data is scaled appropriately, improving the model's convergence during training. This transformation was applied for both the MNIST and Card Dataset.

NLP Preprocessing

- **Tokenization**: We employed the AutoTokenizer from Hugging Face's Transformers library with the pre-trained tokenizer for each model. This converts the raw text pairs into token IDs that the model can process.
- **Padding**: Sentences were padded to a maximum length to ensure uniform input dimensions across the dataset. This is crucial for efficient batch processing during training.
- **Truncation**: Longer sequences were truncated to the maximum length supported by the model (typically 512 tokens for transformer-based models). This prevents information loss from exceeding model capacity.
- **Special Tokens**: The tokenizer automatically added special tokens like [CLS] (for classification) and [SEP] (to separate sentence pairs) to properly structure the input for the transformer model.
- **Batched Processing**: The preprocessing was applied in batches to the entire dataset using the map() function with batched=True, which significantly improves preprocessing efficiency.

Appendix B: Image Classification Results

Table B1. MNIST Results

Model	Optimizer	Accuracy of Best Hyperparameters	Accuracy of the Average over all Hyperparameter Combinations	Time Taken to Train the Max Accuracy Model over all Epochs (s)	Best Performing Hyperparameters
AlexNet	Lion	0.91	0.82	288	BS = 32, LR = 0.0001, WeightDecay = 0.01, Epochs = 5
	Adam	0.91	0.71	308	BS = 16, LR = 0.001, WeightDecay = 0.001, Epochs = 5
EfficientNet	Lion	0.97	0.75	508	BS = 16, LR = 0.0001, WeightDecay = 0.01, Epochs = 5
	Adam	0.97	0.61	501	BS = 32, LR = 0.0001, WeightDecay = 0.01, Epochs = 5
ResNet18	Lion	0.97	0.73	856	BS = 16, LR = 0.001, WeightDecay = 0.001, Epochs = 10
	Adam	0.93	0.53	438	BS = 16, LR = 0.001, WeightDecay = 0.001, Epochs = 5
ResNet34	Lion	0.97	0.77	1179	BS = 16, LR = 0.001, WeightDecay = 0.001, Epochs = 10
	Adam	0.97	0.62	562	BS = 32, LR = 0.0001, WeightDecay = 0.01, Epochs = 5
ResNet50	Lion	0.97	0.72	845	BS = 16, LR = 0.0001, WeightDecay = 0.01, Epochs = 5
	Adam	0.96	0.59	858	BS = 16, LR = 0.0001, WeightDecay = 0.01, Epochs = 5
SqueezeNet	Lion	0.95	0.56	339	BS = 32, LR = 0.0001, WeightDecay = 0.01, Epochs = 5
	Adam	0.95	0.53	336	BS = 16, LR = 0.0001, WeightDecay = 0.01, Epochs = 5

Table B2. Cards Dataset Results

Model	Optimizer	Accuracy of Best Hyperparameters	Accuracy of the Average over all Hyperparameter Combinations	Time Taken to Train the Max Accuracy Model over all Epochs (s)	Best Performing Hyperparameters
AlexNet	Lion	0.47	0.36	237	BS = 16, LR = 0.0001, WeightDecay = 0.01, Epochs = 5
	Adam	0.57	0.31	468	BS = 16, LR = 0.001, WeightDecay = 0.01, Epochs = 10
EfficientNet	Lion	0.74	0.37	262	BS = 32, LR = 0.0001, WeightDecay = 0.01, Epochs = 5
	Adam	0.75	0.28	261	BS = 32, LR = 0.0001, WeightDecay = 0.01, Epochs = 5
ResNet18	Lion	0.76	0.41	500	BS = 16, LR = 0.01, WeightDecay = 0.01, Epochs = 10
	Adam	0.65	0.24	497	BS = 16, LR = 0.001, WeightDecay = 0.001, Epochs = 10
ResNet34	Lion	0.74	0.44	270	BS = 16, LR = 0.001, WeightDecay = 1, Epochs = 5
	Adam	0.77	0.32	267	BS = 32, LR = 0.0001, WeightDecay = 0.01, Epochs = 5
ResNet50	Lion	0.83	0.44	300	BS = 16, LR = 0.001, WeightDecay = 0.1, Epochs = 5
	Adam	0.82	0.31	306	BS = 16, LR = 0.0001, WeightDecay = 0.01, Epochs = 5
SqueezeNet	Lion	0.11	0.03	241	BS = 16, LR = 0.0001, WeightDecay = 0.01, Epochs = 5
	Adam	0.32	0.05	480	BS = 16, LR = 0.001, WeightDecay = 0.01, Epochs = 10

Appendix C: Image Classification Charts

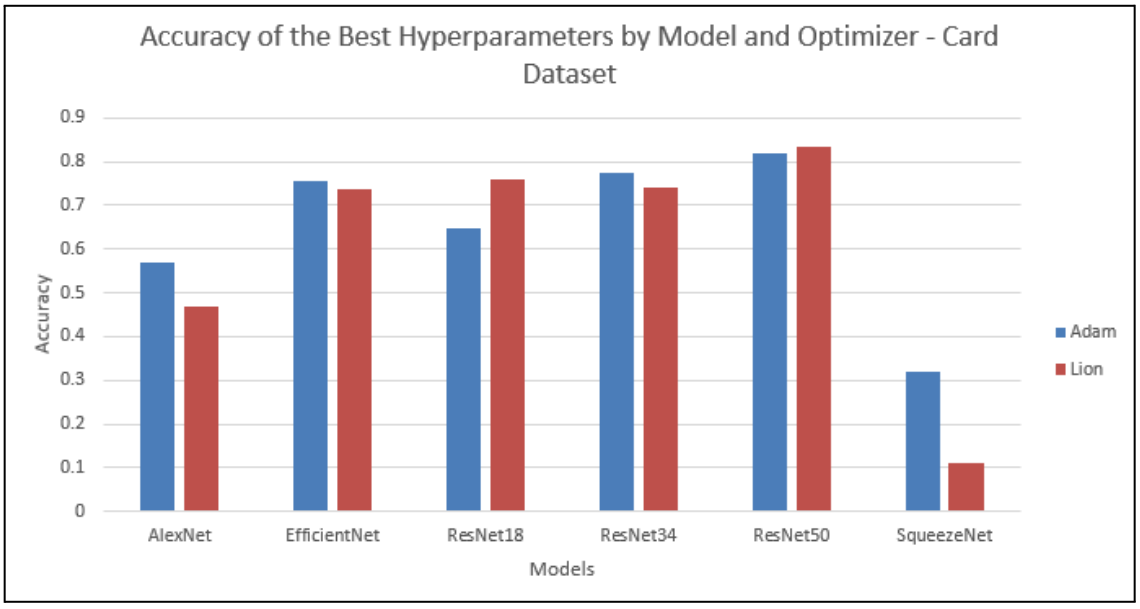

Figure C1. Accuracy of each model's best hyperparameters on the MNIST dataset for both optimizers

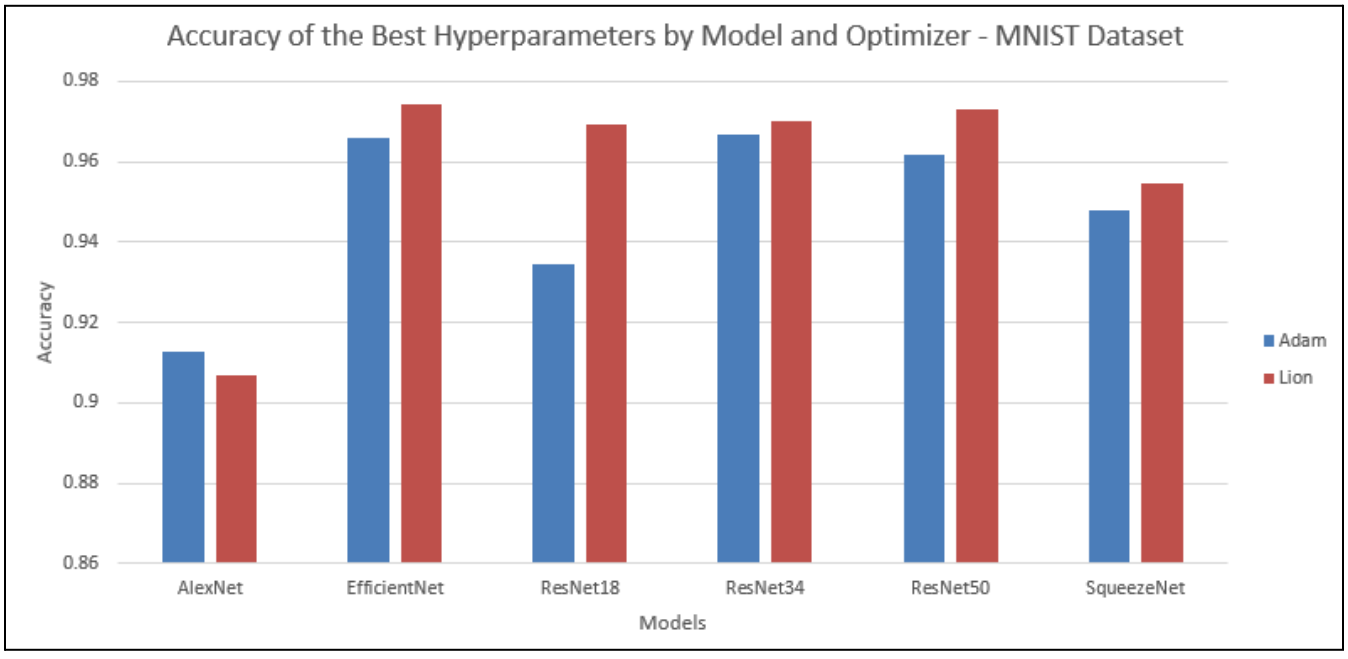

Figure C2. Accuracy of each model's best hyperparameters on the Cards dataset for both optimizers

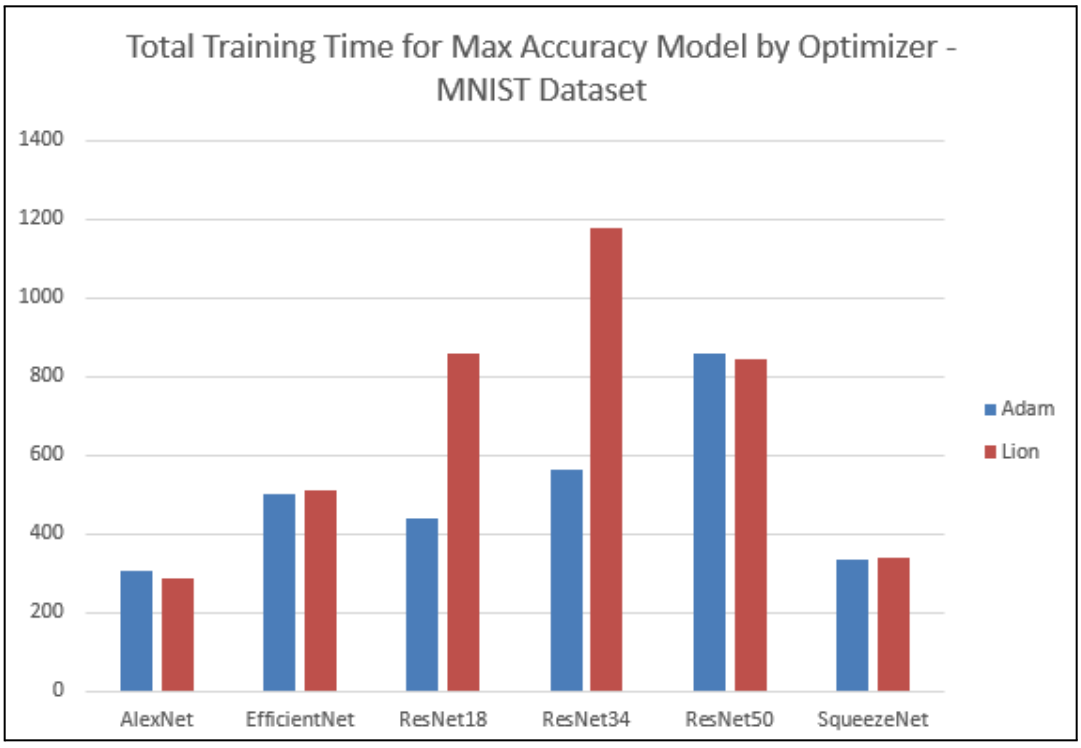

Figure C3. Total training time taken for each model's best hyperparameter model on the MNIST dataset

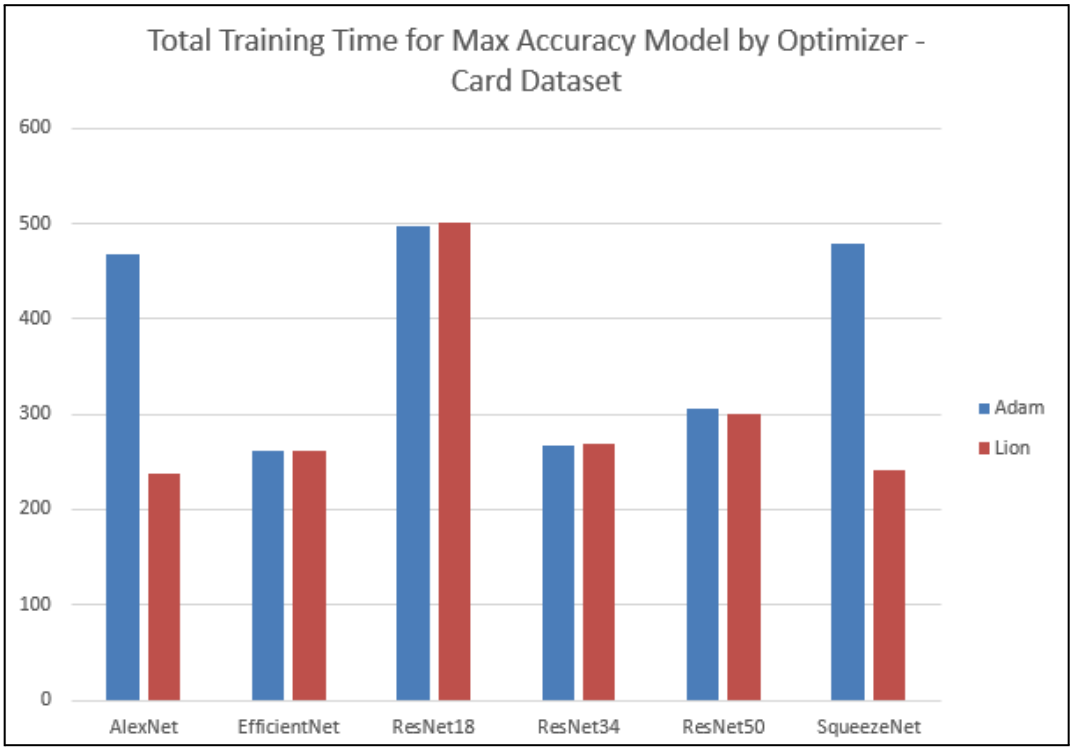

Figure C4. Total training time taken for each model's best hyperparameter model on the MNIST dataset

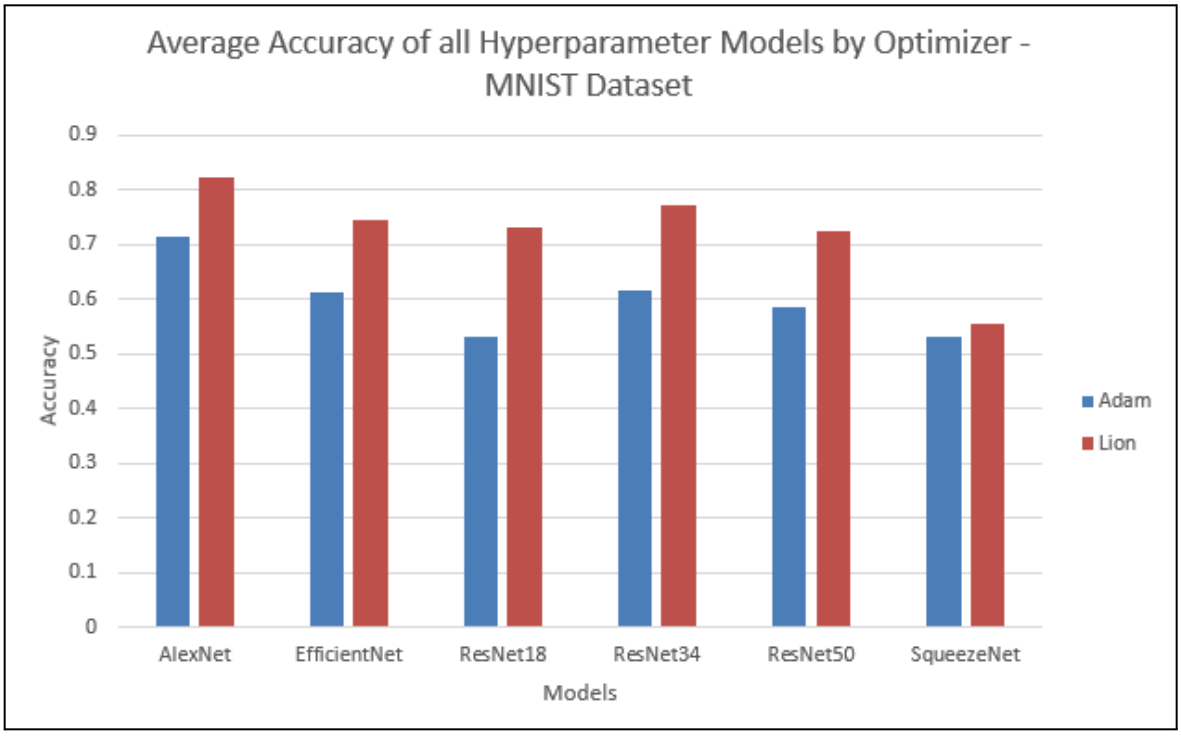

Figure C5. Average accuracy for every hyperparameter combination for each model on the MNIST dataset for both optimizers

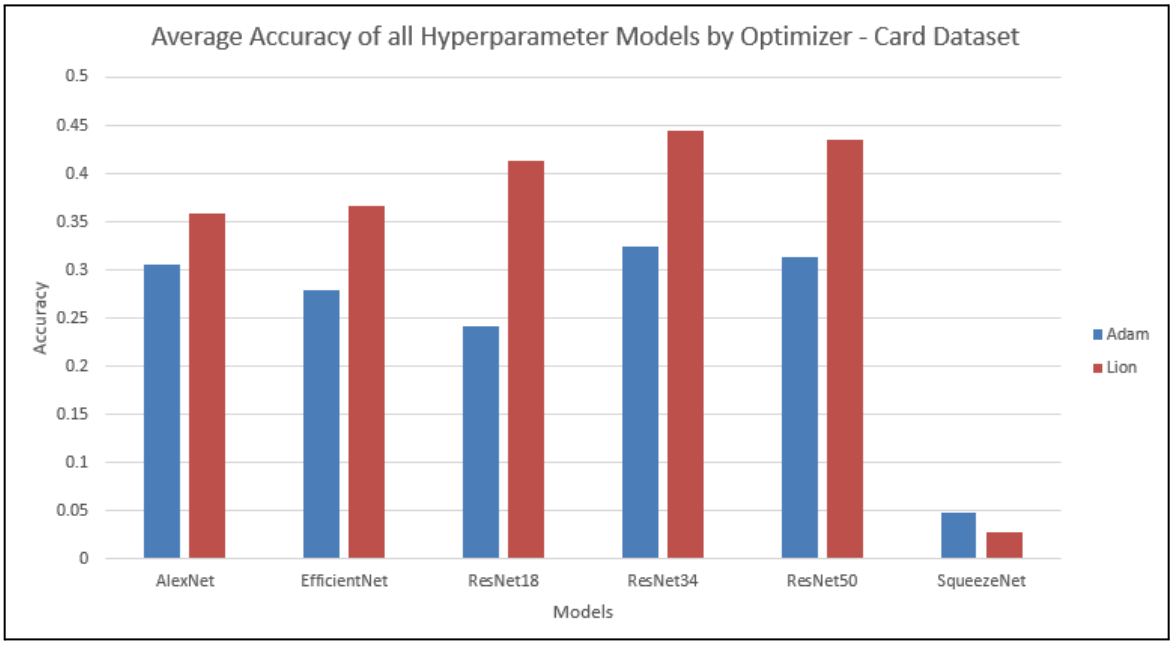

Figure C6. Average accuracy for every hyperparameter combination for each model on the Cards dataset for both optimizers

APPENDIX D: NLP RESULTS

Table D1. NLP Results

Model	Optimizer	Final Train Loss	Evaluation Loss	Evaluation Accuracy	Iterations per Second	Training Time (10 epochs)
BERT-base-cased	Lion	0.6700	0.5278	78.19%	1.55	24 min 45 sec
BERT-base-cased	Adam	0.2488	0.5358	74.51%	1.10	34 min 50 sec
ALBERT-base-v1	Lion	0.6843	0.6234	68.38%	1.75	21 min 54 sec
ALBERT-base-v1	Adam	0.4442	0.3991	80.64%	1.65	23 min 14 sec
DistilBERT-base-cased	Lion	0.6868	0.5271	75.00%	3.19	12 min 00 sec
DistilBERT-base-cased	Adam	0.1170	0.5636	72.55%	3.06	12 min 32 sec

APPENDIX E: NLP TRAIN AND EVAL LOSS CHARTS

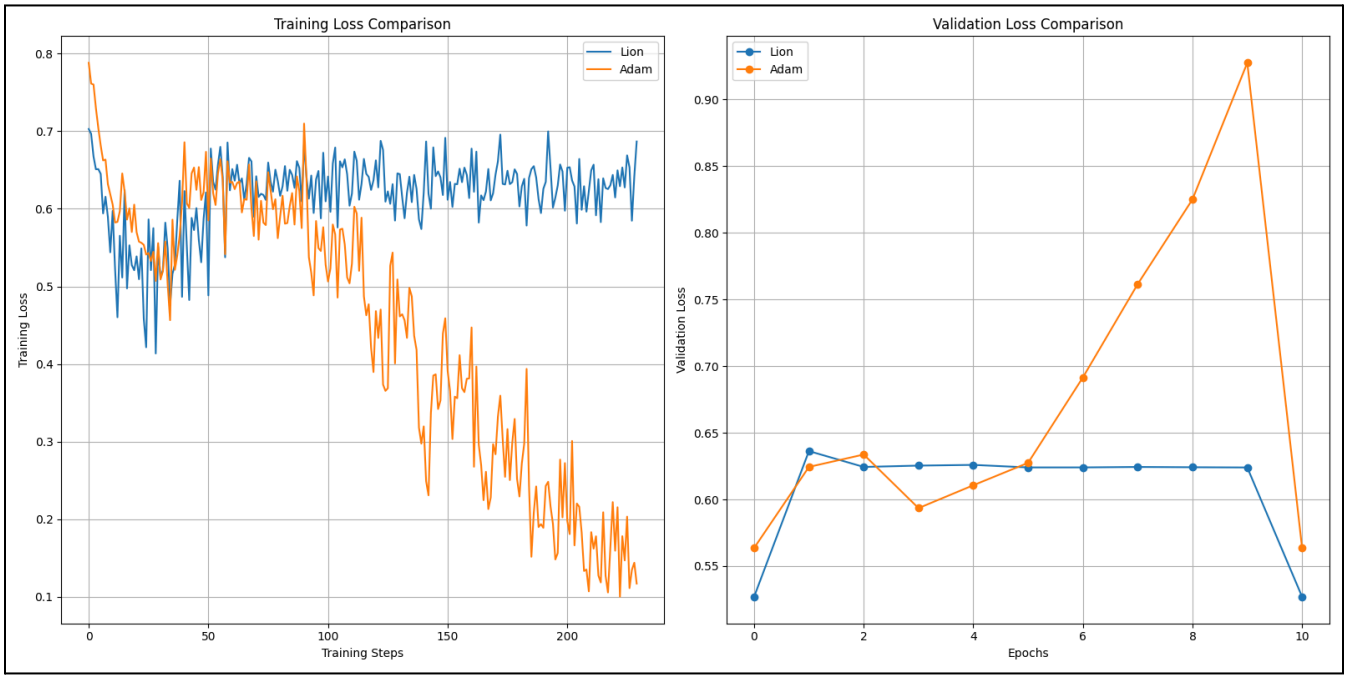

Figure E1. DistilBERT Train and Validation Loss Graphs

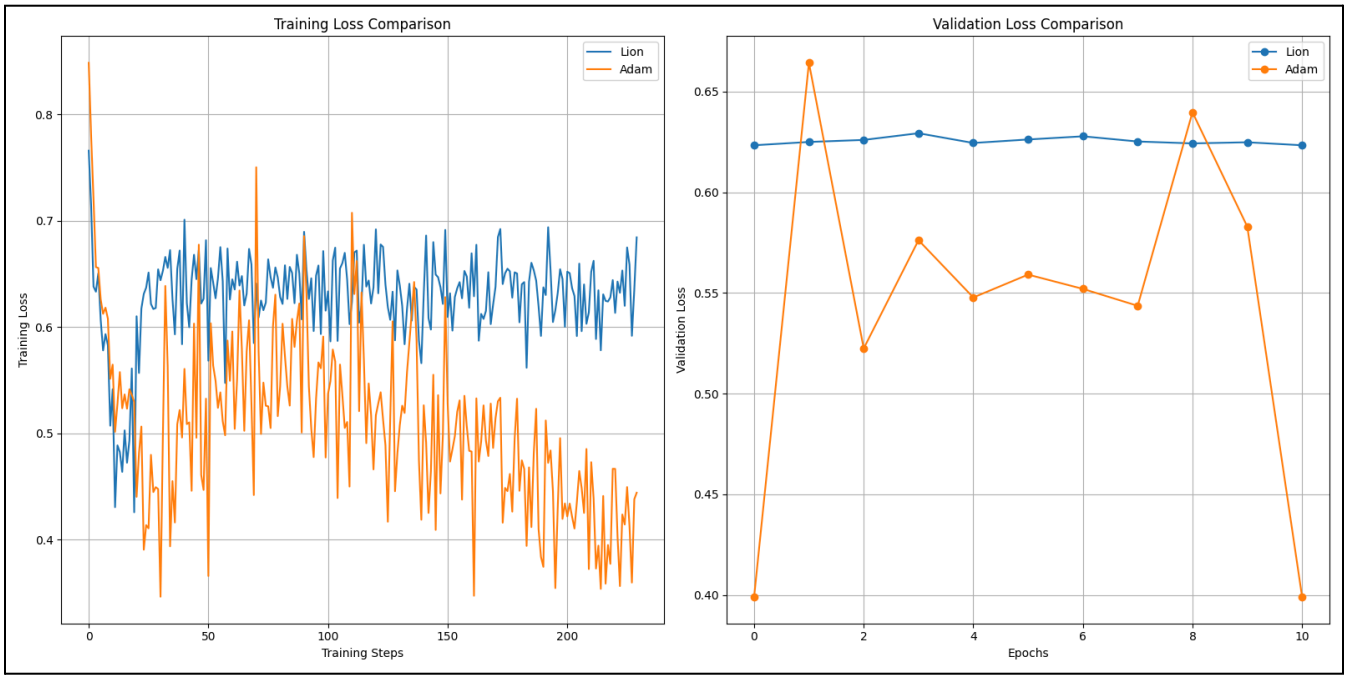

Figure E2. ALBERT Train and Validation Loss Graphs

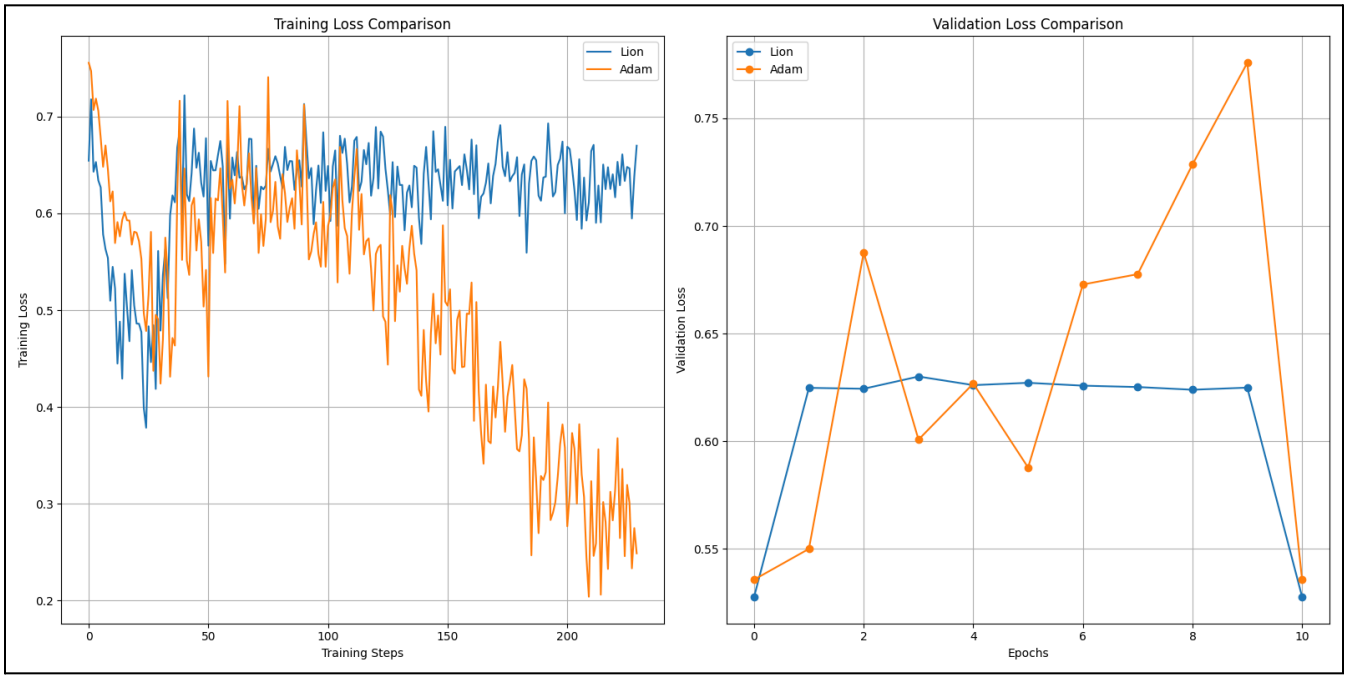

Figure E3. BERT Train and Validation Loss Graphs