Analysis and Design of Novel Optimizers for Neural Networks

Author: Vuk Rosić
Bcs of Mechatronics Engineering
Óbuda University
Supervisor: Dr. Hanka László
Date: December 2025

Abstract

This thesis compares Muon optimizer and Adam optimizer for training large language models and neural networks. The underlying principles behind the design of new optimizers is also examined. A search within more than 45 experiments reveals optimal setup for both Muon and Adam optimizers, as well as relationship between learning rate, momentum, and second derivative curvature space.

Muon is found to perform best with momentum of 0.9, cosine annealing and weight decay of 0.2. The Newton–Schulz experiments show that 3 iterations yield a good trade-off between compute and performance, closely matching 5 iterations in performance while using 40% less compute. These results show that Muon requires less data and causes neural networks to learn 20 to 40 percent faster. Design principles like gradient orthogonalization and learning rate scheduling that will lead to new optimizers are also examined.

All experiments, code and results from this thesis are published at:
https://github.com/vukrosic/analysis-of-optimizers

Table of Contents

Abstract **1**
Table of Contents **4**
1. Introduction **6**
1.1 Motivation 6
1.2 Research Questions 7
1.3 Contributions 8
1.4 Thesis Organization 8
2. Background and Related Work **10**
2.1 Optimization Algorithms in Deep Learning 10
2.1.1 First-Order Methods 10
2.1.2 Second-Order Methods 10
2.1.3 The Muon Optimizer 11
2.2 Mixture-of-Experts Models 13
2.2.1 MoE Architecture 13
2.2.2 Training Challenges 14
2.2.3 Prior Work on MoE Optimization 15
2.3 Muon and Adam Hyperparameter Optimization 15
2.3.1 Learning Rate Selection 15
2.3.2 Momentum and Weight Decay 17
2.3.3 Systematic Ablation Studies 18
2.4 The Evolution and Design of Optimizers 19
2.4.1 First Generation: Momentum and Acceleration 19
2.4.2 Second Generation: Adaptivity 20
2.4.3 Third Generation: Decoupling and Simplification 20
2.4.4 Fourth Generation: Structure-Aware Optimization 20
2.4.5 Research Pathways for Optimizer Discovery 20
2.5 Research Gap 22
3. Methodology **23**
3.1 Experimental Framework 23
3.1.1 Research Approach 23
3.1.2 Experimental Design Principles 24
3.2 Evaluation Metrics 24
3.2.1 Primary Metrics 24
3.2.2 Secondary Metrics 24
3.3 Hyperparameter Search Strategy 25

3.3.1 Learning Rate Search 25
3.3.2 Ablation Study Design 25
3.4 Implementation Details 26
3.4.1 Software Stack 26
4. Experimental Setup 27
4.1 Model Architecture 27
4.1.1 Base Transformer Configuration 27
4.1.2 Model Initialization 27
4.2 Dataset and Data Processing 27
4.2.1 Dataset Selection 27
4.3 Training Configuration 28
5. Results 29
5.1 Learning Rate Sweeps 29
5.1.1 Muon Learning Rate Sweep 29
5.1.2 Adam Learning Rate Sweep 31
5.2 Hyperparameter Ablations 32
5.2.1 Momentum Variations (Muon) 32
5.2.2 Weight Decay Variations (Muon) 33
5.2.3 Newton-Schulz Iterations (Muon) 33
5.2.4 Nesterov Momentum (Muon) 34
5.2.5 Warmup Variations (Muon) 34
5.2.6 Learning Rate Schedules (Muon) 35
5.2.7 Adam Optimization Suite 35
5.3 Final Optimized Comparison 36
5.3.1 Best Configurations 36
5.3.2 Performance Comparison 36
6. Analysis and Discussion 38
6.1 Why does the Muon optimizer outperform Adam? 38
6.2 Limitations 38
7. Conclusion 39
8. Future Work 40
9. References 41

1. Introduction

1.1 Motivation

Optimizers are a key element of neural networks and machine learning, and they have a direct impact on the final model quality and learning speed. Currently the most popular optimizer is Adam (Adaptive Moment Estimation) [Kingma & Ba, 2015], due to its adaptive learning rates, first and second derivative momentums that are unique to every parameter, which allows neural network to do custom adjustments for every parameter individually in order to keep updates big enough so they are meaningful, but not too big so they become unstable. However, a new optimizer called Muon (Momentum Orthogonalized by Newton-Schulz) [Malladi et al., 2024] has emerged that challenges Adam's dominance.

Muon makes all weight update matrices orthonormal. Weight update matrices are gradients of the loss with respect to weights of the neural network. These gradients are usually subtracted from the weights. Since the gradients show in the direction of the steepest increase in the loss function, subtracting them from weights will reduce the loss. Gradients are not added in their raw form as they would be too big, instead they are first multiplied with a small number called learning rate, which is usually between 0.1 and 0.01.

Muon's advantage is that it makes weight update matrices orthonormal, meaning that each row or each column are orthogonal to each other, if looked at as vectors. This was found to make neural networks learn faster with less data compared to Adam and other optimizers.

For examining Muon versus Adam optimizers, a large language model with Mixture-of-Experts (MoE) [Shazeer et al., 2017; Fedus et al., 2022] architecture is trained. This will allow optimization for a widely used architecture where every improvement can have immense impact.

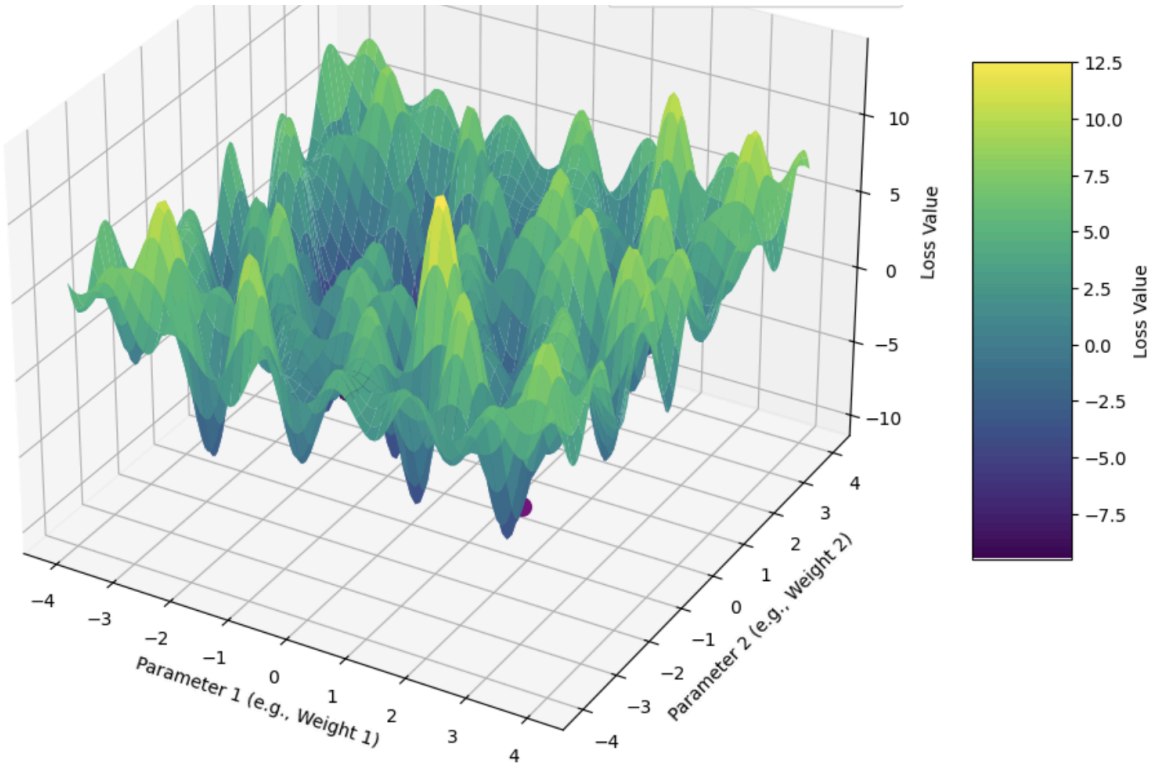


Figure 1: Example of a loss surface of a 2-neuron neural network. Higher points on the surface represent higher error of the neural network prediction based on which weights (shown as x and y axes) the network chose. The optimizer will update weights to move towards the bottom of the loss surface - this represents neural network learning to minimize the prediction error. An optimizer that has a better way of traversing the loss surface (searching for lower loss) will train the neural network faster.

1.2 Research Questions

This thesis answers the following research questions:

1. Does muon outperform Adam in terms of final validation loss when training MoE transformer models?

2. What hyperparameter settings are optimal for each optimizer, how sensitive are they to change?

3. How do optimal learning rates differ between Muon and Adam optimizers?

4. Do learning rate schedules and warmups affect Muon and Adam differently?

5. What computational cost do Muon's Newton–Schulz iterations infer, can it be reduced without reducing quality or the optimization?

6. What are optimal configurations for implementing these two optimizers in production?

1.3 Contributions

This thesis contributes in the following ways:

1. 45+ experiments are conducted in a thorough hyperparameter search, providing well grounded methods for hyperparameter setup.

2. Both optimizers are tuned independently to compare them at their highest performance.

3. Core differences in Muon and Adam optimizations are identified, which include differences in learning rate setup, scheduling and warmup strategies.

4. Empirical analysis shows key guidelines for designing new neural network optimizers, especially when working with first- and second-order derivation methods.

5. Understanding of optimizers in training of mixture-of-experts large language models.

6. All experimental code and results are published on GitHub for reproducibility.

1.4 Thesis Organization

The thesis is organized as follows: Chapter 2 shows relevant research on neural network optimization techniques, and hyperparameter tuning, as well as MoE models. Chapter 3 discusses experimentation and evaluation methods. Chapter 4 examines the experimental setup of this thesis.. Chapter 5 describes the results and findings, which include the learning

rate search, ablations and performance comparisons between optimizers. Chapter 6 presents analysis and discussion of the results. Chapter 7 contains the conclusion. Chapter 8 discusses future work in this research area.

2. Background and Related Work

2.1 Optimization Algorithms in Deep Learning

2.1.1 First-Order Methods

Stochastic Gradient Descent (SGD) [Robbins & Monro, 1951] emerged as the foundation for neural network optimization. To date, SGD with momentum [Polyak, 1964; Sutskever et al., 2013] stayed very competitive and a well rounded choice for many optimization problems. However, the constant need to tweak its learning rate presents a challenge when it comes to practical deployment and use of SGD.

Adaptive learning rate methods solve this challenge by modifying learning rates for each parameter individually. RMSprop [Tieleman & Hinton, 2012] and AdaGrad [Duchi et al., 2011] were first optimizers that employ adaptive learning rates, which later lead to Adam [Kingma & Ba, 2015]. Adam has shown to be the best performing optimizer that successfully combines momentum and adjustable learning rates. AdamW [Loshchilov & Hutter, 2019] outperforms Adam by separating weight decay from gradient updates, which most researchers see as a bugfix that made it fully functional.

Other works explored variations on Adam like RAdam [Liu et al., 2020], which deals with warm-up heuristics, and AdaBound [Luo et al., 2019], that switches between Adam and stochastic gradient descent during training. However, the improvements these optimizers provide are often incremental.

2.1.2 Second-Order Methods

Second-order optimization methods use the Hessian matrix curvature information to improve neural network learning and convergence speed. Methods uch as L-BFGS [Liu & Nocedal, 1989] and similar Newton's technique and quasi-Newton methods are already well-established in the field of in convex optimization, nevertheless, they have scaling problems in deep learning because they require $O(n^2)$ memory scaling in order to store Hessian estimates, which is a big issue for neural networks and large language models in particular, as they are already memory constrained, and not compute constrained. This leads to compute heavy optimization methods being preferable to memory heavy optimization methods.

Hessian matrix is replaced for Fisher information matrix in Natural gradient descent [Amari, 1998], resulting in a more mathematically principled setup. In K-FAC [Martens & Grosse, 2015; Grosse & Martens, 2016]Fisher matrix is further approximated with Kronecker factors, which made the second-order optimization techniques finally feasible for neural networks.

Shampoo [Gupta et al., 2018] and Distributed Shampoo [Anil et al., 2020] use a unique matrix factorization strategy, performing very successfully on large-scale models. However, these optimizers still have a problem of large computational needs for processing.

2.1.3 The Muon Optimizer

Muon (Momentum Orthogonalized by Newton-Schulz) [Malladi et al., 2024] is a new optimizer that combines first and second order derivative methods. Muon efficiently orthogonalizes the weight update matrix using Newton-Schulz iterations [Higham, 1986] which serve as an approximation of applying singular value decomposition to a matrix. This method has shown to increase learning speed and make neural networks learn more with less data.

The Newton-Schulz approximates the matrix inverse. When properly tuned and configured, Newton-Schultz converges quadratically (the error decreases very rapidly), which makes it a very good alternative to explicit matrix inversion. This way it's possible to utilize GPUs to do extremely fast matrix multiplications, which provide a lot faster compute compared to inverting matrix on a CPU. This shows a general tendency in optimization problems to perform computation on GPUs and to convert computations to matrix multiplications to gain speed and efficiency. Muon uses Newton-Shultz iterations to or orthogonalize gradient matrices, which allows it to find better optimization trajectories over the loss function surface.

Muon is very computationally efficient, with just O(n) memory scaling, it avoids explicit Hessian that creates a memory bottleneck. It produces better gradient conditioning with the orthogonalized updates, as well as good theoretical connection with other gradient algorithms that are shown to work well empirically. However, given that muon is less than a year old, it has received a lot less research and experimentation compared to Adam.

Muon gained significant attention in 2024 and 2025 for setting records on multiple benchmarks and performing up to two times faster LLM training, depending on the architecture. It did this by treating weights as matrix transformations, rather than independent scalar updates.

The traditional optimizers (SGD, AdamW, RMSProp) update parameter elements one by one (element-wise). Then calculate step size for each parameter individually, and update them. They are not considering the fact that the weights are part of a matrix which has it's mathematical properties. Muon takes this fact into account. It's a second order-like optimizer which is designed specifically for 2D matrices in hidden layers.

Instead of using elementwise variance to scale the parameters (like Adam), it orthogonalizes the update matrix, making sure the update matrix pushes weights uniformly in all learned directions equally without exploding.

The update consists of two main ingredients - momentum and orthogonalization. Firstly, muon calculates the standard momentum buffer, similar to SGD with momentum, then it performs orthogonalization. Since calculating SVD is computationally slow and expensive, the creators of Muon use Newton-Schulz approximation to approximate orthogonalization. Every step makes the matrix more orthogonal. The update matrix doesn't need to be fully orthogonal as neural networks can tolerate some imperfection. This provides a very computationally fast and efficient method for fixing the weight update matrices.

Newton-Schulz iterations transform the update matrix W into matrix O such that O.T @ O = I, or close to it. Mathematically, this is equivalent to taking UV components of Singular Value Decomposition of the momentum.

Since SVD is expensive in terms of compute on a GPU, Muon approximates the orthogonal matrix using matrix multiplications only with Newton-Schulz iterations.The orthogonal matrix is finally added to the weights. Given that an almost-orthogonal matrix is added to the weight in every weight update, it will pull the weights towards being almost-orthogonal over time.

2.2 Mixture-of-Experts Models

2.2.1 MoE Architecture

Mixture-of-Experts (MoE) models [Jacobs et al., 1991; Jordan & Jacobs, 1994] use multiple neural network layers and a router to choose which of them the current piece of data will be passed through. Each of the layers has a specialized knowledge and processes the data in a different way, which is learned during the training. These networks are called "expert" networks and the router that is choosing between them serves as a gating mechanism that determines which expert is most suitable for the current input or piece of data.

Routers would usually convert the input vector into a probability distribution over the experts, showing probability for each expert's fit to the data vector. Different sampling methods are then employed to select an expert, which include picking the highest probability one, top-p (sampling only from experts that exceed certain minimum probability p), top-n (picking from n highest probability experts regardless of their probabilities) and many others.

This division into expert networks allows for a massive scale of AI models without needing to load them all at once into memory, but processing only a fraction of the network for any given data vector. This has allowed massive models to be deployed and served in production environments efficiently and at a low cost.

A mixture-of-experts transformer architecture [Shazeer et al., 2017] is using multiple expert feedforward networks in the large language model (LLM) to allow LLMs to scale to trillions of parameters without needing to process each token (or word) when generating text. Instead, it only activates a small fraction of all parameters, best suited for the current token, immensely improving efficiency, memory and computation needs of the model.

Sampling methods for MoEs are often choosing top-k experts, for example choosing top 2 highest probability experts out of 8 in total. However, in deployment of very large language models, the total number of experts is often in hundreds.

2.2.2 Training Challenges

To properly choose the best experts, the LLM uses a router, which must be trained properly to utilize all experts. There is a danger of collapse where router only learns to pick a few experts, which causes only those experts to learn with network's backpropagation, starting a vicious cycle where only a few experts are learning and thus always selected, as selecting other experts would yield a big increase is loss and error of the model.

To solve this issue, researchers came up with load balancing loss. This is an auxiliary loss that is added to the main next token prediction loss and has the goal of making the router choose every expert equal number of times, that is sending an equal amount of data to every expert network, this utilizing the high number of parameters of the LLM efficiently.

When this training challenge of load balancing is resolved, it results in a well trained and diverse set of experts where each has a specialized knowledge suitable for different tokens, for example, math expert, finance expert, history expert and more. Oftentimes experts have multiple specializations that are complementary to each other.

This method also allows experts to work together and provide well combined processing for the LLM tokens.

There are different methods of implementing load balancing loss, and with all of them the goal is to prevent all tokens being sent to only a few experts, while others stay underutilized and untrained [Lepikhin et al., 2021].

The importance of the proper implementation of load balancing also comes into play in the early training where experts do not contain enough knowledge or ability to process tokens well, so the gating will be a bit more random, as there is no best expert for every token. Router also allows the mixture of experts network to be differentiable, as the probability distribution is used to backpropagate gradients.

These issues make the mixture of experts LLMs especially interesting for optimizer research as any improvements will have a massive impact on the LLM industry.

2.2.3 Prior Work on MoE Optimization

Switch Transformers [Fedus et al., 2022] studies stability of the training and offers a method for better load balancing using the number of tokens routed to each expert, as well as the probability each expert is given on average to push towards equal distribution of token routing. GLaM [Du et al., 2022] showed how to scale LLMs to trillions of parameters. ST-MoE [Zoph et al., 2022] improved stability in vision language models using this method.

However, detailed comparisons of optimizer techniques for mixture-of-experts LLMs are still restricted and shallow. Most methods utilize Adam without regard for Muon, which is a newly created optimizer that shows better training speed with less data. This thesis work fills that gap.

2.3 Muon and Adam Hyperparameter Optimization

In the following section the methods for optimizing Muon and Adam optimizers will be discussed.

2.3.1 Learning Rate Selection

Learning rate is seen as the most important hyperparameter [Bengio 2012]. If there is only time to ablate one hyperparameter, it should be the learning rate. It often returns the highest improvements in loss and training speed compared to any other hyperparameter.

Common methods of searching include grid search, where researchers perform systematic examination of the number of learning rate values, usually ranging from 0.3 to 0.0003, random search (Bergstra & Bengio, 2012) where a random algorithm is employed to select learning rates, this algorithm may be influenced by the previous best results and may employ a strategic random selection like selecting from a normal distribution as well as other random selection methods, bayesian optimization where model based sequential optimization [Snoek et al., 2012] is performed, or population based training which employs an evolutionary approach [Jaderberg et al., 2017].

Another important part of learning rate search is learning rate scheduling. The reasoning behind this method is that in the beginning of the training the neural networks knows less, so the weights should be updated more aggressively, to speed up the new knowledge and learning injection into the neural network, however, as the networks learns, then the weights should be updated less as well, as each update risks making the neural network forget other things it learned previously.

Multiple techniques for learning rate scheduling are invented, including step decay, where learning rate is decreased by a fixed amount every number of steps, and stays the same within that window of steps. This is one of the first methods and later methods improve upon that method. Exponential decay, as the name suggests, lets learning rate decay exponentially, providing stronger learning signals in the beginning, and weaker learning signals towards the end. Cosine annealing is an improvement upon this that showed that cosine shaped learning rate decay [Loshchilov & Hutter, 2017] performs better for the optimization process.

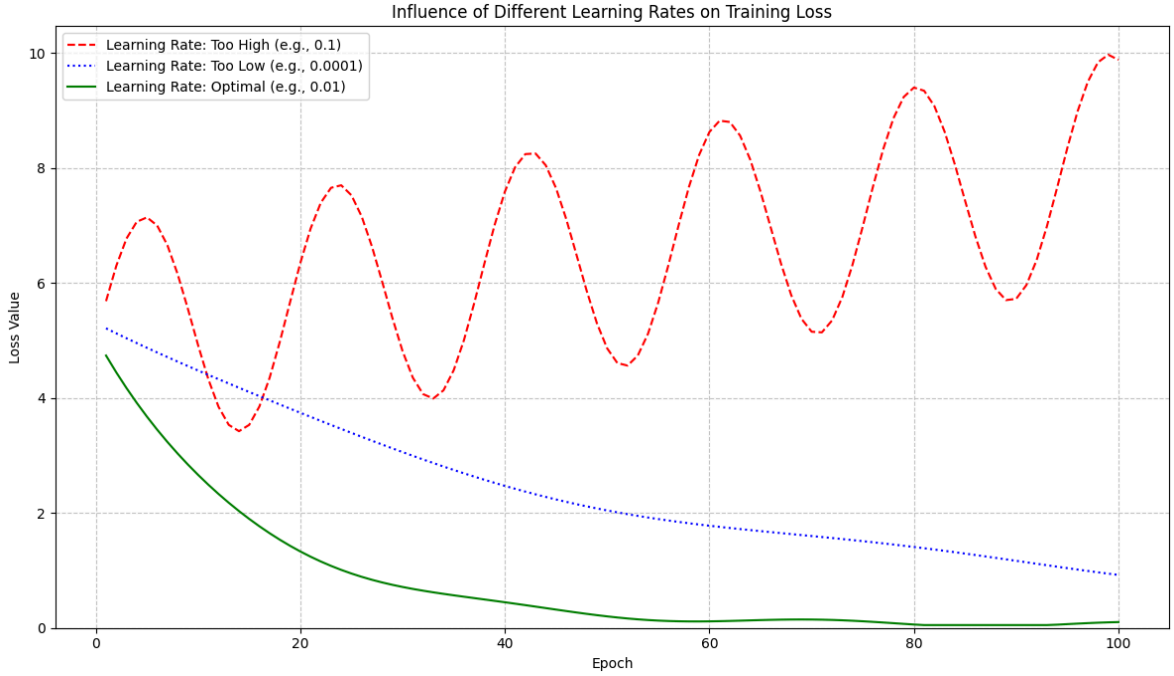


Figure 2: Example of 3 learning rates. A too high learning rate (red) results in the loss (error) increasing, and the neural network is unable to improve due to updates to weights

being too large. Too low learning rate (blue) causes the network to learn too slowly, and the optimal (green) graph shows fastest learning because of an optimal learning rate (in this case 0.01).

Besides the decay, it is also discovered that warmup improved training speed and stability [Goyal et al., 2017]. The logic behind warmup is that in the beginning neural network doesn't know anything, so the gradients will point in seemingly random directions, but the neural network will learn very quickly in the beginning stage, so the learning rate will start from 0 and quickly increase at the beginning, it will remain large and slowly decay, usually with the cosine schedule. This proved to be the best scheduling method for most neural network optimization tasks.

Other strong influences to the optimal learning rate include batch size and model design. When choosing learning rate, it's often required to do experimentations and ablations, while theory can only provide rough guidelines.

2.3.2 Momentum and Weight Decay

Momentum is used to calculate the exponential moving average of the neural network gradients (β_1), as well as the second moments (β_2 for Adam). Through empirical studies optimal values for Adam are found to be $\beta_1=0.9$ and $\beta_2=0.999$, but these values may not be ideas for all scenarios. So it is of crucial importance to perform ablation studies for the optimizer hyperparameters.

Mathematically, momentum is calculated as the exponential moving average of the gradients. New momentum is calculated by multiplying the old momentum with some hyperparameter β that determines the memory, or how much of the old momentum remains. That is then added to $(1-\beta)$ multiplied by the new gradient. This is a way to keep the average of the past gradients included in the new gradient, as this way the random variations between gradients are minimized, while there is a push in the average direction of the slope with random variations at the minimum.

A helpful rule is that the optimizer remembers $1/(1-\beta)$ steps, so for $\beta=0.9$, it remembers the last 10 steps, for $\beta=0.99$ the optimizer remembers 100 past steps, and for $\beta=0.5$ the

optimizer remembers the last 2 steps. The usual default of β is 0.9, as it strikes the balance between stability and reactivity to the new gradients. However, different training scenarios may require different settings. For example, a small batch size will provide a high noise environment, as there are not many samples to average across the samples to get the general case representation. The issue is that if the momentum hyperparameter beta is too low, the optimizer will zigzag based on random data. In this case a higher momentum will be needed to force the optimizer to ignore individual noisy signals and move in the average direction of the loss surface descent.

In non-stationary optimization problems, where the loss surface shifts and changes during the training, for example in reinforcement learning, as the new signal is received from the environment, the momentum should be lower because the old momentum that was pushing the gradients changed and it doesn't exist anymore. So here the momentum of 0.5 or 0 will allow the model to adapt to changing loss surfaces.

2.3.3 Systematic Ablation Studies

To isolate and examine the effect of specific hyperparameters, ablation studies are performed. In ablation studies often only one hyperparameter is changed while others remain the same. This allows for exact pinpointing of the relationship between the hyperparameter and the final loss of the neural network.

The systematic ablation studies are done such that preferably only one variable is changed, which creates a controlled environment for the hyperparameters like decay, momentum, learning rate and many others. This way a difference is loss can be measured by the difference in the hyperparameter change, providing a direct link and insight on what influences neural network training and in which way. A map of how every hyperparameter influences the loss should be created. Most hyperparameters have the “Goldilocks” zones, where too low or too high value influence the loss negatively, and the “sweet spot” must be found. Hyperparameter sweep is rarely done in a single experiment. Usually it’s done by starting with a coarse and wide range of values, and proceeding with a narrower range that shows more promise. For example, for weight decay the ablations might start with [0.0,0.001,0.01,0.1] range of values. Once a good range is identified, the values in it may be examined further, for example [0.008,0.01,0.012,0.014].

One pitfall in insulating each parameter in ablations is that in deep learning the parameters are often dependent. Finding an optimal hyperparameter with one setup, and then changing the next one may cause the first hyperparameter to not be optimal anymore. This is called parameter coupling, it's when one parameter depends on the other, like learning rate and batch size. If batch size is increased, the learning rate should also be increased for optimal training. If batch size is ablated while learning rate is kept constant, this could falsely show wrong optimal batch sizes. Another example is momentum and learning rate. Large momentum and large learning rate may cause models to diverge. High momentum might be better if the learning rate is lowered appropriately. The solution for searching parameters based that are strongly coupled is to perform a 2 dimensional grid search, which means ablating parameters simultaneously, rather than 1D ablations.

This study performs systematic ablations on most important hyperparameters, including learning rate, weight decay, momentum, scheduling, warm-up ratio, and Muon-specific parameters (Newton-Schulz iterations, Nesterov momentum) and more.

2.4 The Evolution and Design of Optimizers

Neural network optimizers have been designed throughout many different generations, each generation improving on the past and bringing new theoretical and practical knowledge that can be used to train neuron networks faster and with less data.

2.4.1 First Generation: Momentum and Acceleration

The first generation of optimizers included gradient descent and stochastic gradient descent. These simple methods provided neural networks with the ability to learn, however, they were vulnerable to randomness of the gradients and the loss surface. Thus momentum [Polyak, 1964] was introduced. This was the first real design improvement. This made the neural network find the average of gradients, so the randomness of new gradients didn't influence the learning too much because the updates were moving in the average direction of all the gradients, with new ingredients also providing a bit of sway. This improved learning stability and dealt with the stochastic nature of gradient and the lost surface.

2.4.2 Second Generation: Adaptivity

The main insight that led to the second generation was that different parameters should have different momentums and learning rates adapted to them, like AdaGrad [Duchi et al., 2011]. RMSprop [Tieleman & Hinton, 2012] further developed this by including exponential moving averages. Adam [Kingma & Ba, 2015] combined best of both worlds - the first-moment and the second-moment adaptivity of RMSprop, which made Adam optimizer standard choice for over a decade. Main contributions of these methods were scaling the learning rate element-wise, by taking the geometry of the loss function into account.

2.4.3 Third Generation: Decoupling and Simplification

As models became bigger, weaknesses of Adam were becoming more and more obvious. AdamW [Loshchilov & Hutter, 2019] fixed Adam’s weight decay. This led to an important conclusion that regularization and optimization in an optimizer must be explicitly addressed and not just assumed. Lion [Chen et al., 2023] proved that complicated designs are not necessary as it's using only the momentum and the sign of the update, which showed to be unexpectedly simple, suggesting that the best optimizer may not need to be complex at all.

2.4.4 Fourth Generation: Structure-Aware Optimization

The current optimizer frontier, which includes optimizers like Muon, Shampoo, and K-FAC, go beyond treating weights as just vectors. These structure-aware optimizers see weight matrices as 2D matrices and are aware of their mathematical properties. Shampoo [Gupta et al., 2018] uses Kronecker products to approximate second-order preconditioning. Muon [Malladi et al., 2024] uses Newton-Schulz iterations to quickly move matrices towards more orthogonalized versions of themselves, which imposes a hard limit on the update geometry. This proved to be essential for faster neural network learning.

2.4.5 Research Pathways for Optimizer Discovery

1. The "trial and error" method

There are multiple methods which scientists use to discover new optimizers. The most popular one is trial and error and empirical experimentation. They start with the current optimizer, which may be Adam or Stochastic Gradient Descent, and they examine where they fail. For example, they can see them getting stuck or shaking excessively on the lost

surface, or taking too much time, or taking a suboptimal route towards the minimum. And then they think of fixes.

If the optimizer is oscillating too much, they may apply decay which acts as a brake to reduce the oscillation and the movement. Or if the optimizer is traversing the loss surface too slowly, they might add momentum keeping the average of the previous updates to make sure that it keeps updating at a stable pace without the updates becoming too small.

2. The "math theory" method

Another popular way to discover optimizers is through mathematical theory, where scientists start by analyzing the surface and the shape of the loss function and possibly constraining it, and in building rules that will make the optimization process faster and smoother, making the neural network converge to minimum more efficiently. Here scientists describe their ideas with different formulas, showing why and how to derive optimizers and improve the performance of the training of neural networks.

3. The "automated" method

The is the most recent method that has gained significant traction after the successful discovery of multiple optimizers that perform well using automated search. This method uses computers to do a quick search over a vast amount of possible optimizer setups and algorithms, directly converting computation power to new scientific discovery. In the future this might become the main method of scientific discovery, as the compute increases and AI becomes more capable of doing intelligent experiments on its own in an automated fashion.

All of the optimizers discovered through these methods are tested first with some simple loss surface that can be computed quickly. This will eliminate optimizers that cannot even solve simple optimization tasks, and if that task is solved successfully, then a more complex loss surface is given. If this second stage is passed successfully, then the optimizers are compared with the state-of-the-art, currently best optimizers like Adam. Note that different optimizers may perform best in different scenarios, depending on the model design, data and the overall idea behind the training.

2.5 Research Gap

Where there are extensive investigations of Adam, its comparison to Muon and the extensive investigation of Muon are lacking. There are limited MoE investigations where Muon has not been thoroughly analyzed within MoE LLM models. There are no hyperparameter studies and best practices published and available for the wider scientific community. Systematic comparison of Adam and Muon optimal hyperparameters and ablation studies are poorly understood. This thesis tackles these issues and provides experiment-grounded guidelines and settings for optimal performance of both Muon and Adam optimizers.

3. Methodology

3.1 Experimental Framework

3.1.1 Research Approach

The systematic experimental analysis in this thesis consists of the following 3 phases:

Phase 1 - sweeping the learning rates. As discussed earlier, learning rate sweeps are the most important hyperparameter experiments, as they almost always yield the highest improvement on the loss and learning speed of the model. The appropriate learning rate regions for both optimizers are demonstrated, showing exactly the ranges of values for learning rates where the models converge in the fastest possible manner while maintaining predictability and stability. Sensitivity of both optimizers to learning rates are also compared, and showed that Muon optimizer is a lot less sensitive to learning rate change, that is it tolerates a much wider range of learning rates, all of which lead to fast and optimal convergence.

Phase 2 - all other hyperparameters that define optimizers. This is done once best learning rates are picked. This phase contains the highest number of ablations, as often only a single hyperparameter is isolated and changed to measure its exact impact on the loss. Hyperparameters in this phase include weight decay, learning rate schedule types, warmups, Muon-specific hyperparameters and more.

Phase 3 - extended training of the best hyperparameter setup. In this final stage more compute for the training of the winning setups is used, ensuring that these setups hold well in the longer training as well, showing the clear winners in the hyperparameter search process. The statistical variance is also tested, making sure to set different random seeds and to compare the influence of the randomness to the results. The work showed that randomness has minimal influence and that the well-tuned hyperparameters themselves are what brings the improvement in the training process.

3.1.2 Experimental Design Principles

During the experimentation, the following principles are adhered to: keeping all variables consistent except the ones being tested. Optimizing both optimizers independently to the maximum of their performance to ensure fair comparison. Using fixed random seeds and documenting all configurations. Large comprehensive coverage of hyperparameter values to prevent local optima and focusing on the relevant real-world settings that are practically useful.

3.2 Evaluation Metrics

3.2.1 Primary Metrics

The evaluation methods include **validation loss, validation accuracy,** and **perplexity**. The experiments do not include testing on training data because the model might have memorized the training data, so a separate dataset for validation is created that does not appear in the training data. Perplexity is a measurement of how confident or certain or confused the model is. The mode should be confident into the next token prediction. In large language models, validation loss should hover around 3 to 5 and perplexity should hover around 40 to 60. If perplexity is in the hundreds or thousands, then there is an issue in the training or there could be a bug with loading data which happened to us.

Perplexity is measured by how uniform probability distribution over the next token is. So, perplexity will be low if the model assigns high probability to a few tokens, and it will be high if a lot of the tokens have similar probabilities, which means the model is confused and it doesn't know which token should be next or it's not certain. The goal of training large language models is to make the model more certain in what the next token is so it can predict it.

3.2.2 Secondary Metrics

Secondary methods used to train and measure the performance of the training of large language models are training time, convergence speed, expert utilization, where all expert

networks are utilized equally and process equal amount of tokens, and stability of the loss optimization where there are no loss spikes.

3.3 Hyperparameter Search Strategy

3.3.1 Learning Rate Search

Learning rate sweeps are done by first testing a higher number of learning rates for fewer training steps and then taking the best ones and testing them for the higher amount of training steps.

Muon LR: 0.005 to 0.15
- Fast sweep (200 steps): 8 LRs
- Extended sweep (500 steps): 5 best winning LRs

Adam LR Range: 0.0001 to 0.01
- Fast sweep (200 steps): 11 LRs
- Extended sweep (500 steps): 3 winning LRs

Faster sweeps with fewer training steps let the experiments quickly rule out very poorly performing learning rates, and the remaining compute is used to extensively test the winning learning rates.

3.3.2 Ablation Study Design

After optimal learning rates are found, then other parameters are tested.

Momentum: 0.85, 0.9, 0.95, 0.97, 0.99
Weight Decay: 0.05, 0.1, 0.2
Warmup Ratios: 0.0, 0.05, 0.1, 0.2
Newton-Schulz Steps: 3, 5, 7, 10

3.4 Implementation Details

3.4.1 Software Stack

- Framework: PyTorch 2.0+
- Muon: Custom, based on the original research
- Adam: `torch.optim.AdamW`
- Training: Custom loop for training with checkpointing and logging

4. Experimental Setup

4.1 Model Architecture

4.1.1 Base Transformer Configuration

A Mixture-of-Experts large language model is trained with the following specifications: Architecture is a mixture of expert transformers with 50,257 tokens vocabulary size, that is from SmolLM tokenizer. Embedding dimension is 384 and number of transformer layers is 6. Context length is 512 tokens and attention head number is 8. Load balancing weight for MoE is 0.01.

The total number of parameters is about 19.3M for the embedding layer, 28.3M parameters for the attention layers, 28.3M parameters for MoE layers and other, which include normalization layers are 3.1M parameters.

Due to the sparse activation where only top 2 most relevant experts are activated, only about 28.4% of parameters are activated within a forward pass.

4.1.2 Model Initialization

In order to ensure faster convergence and prevent model collapse, various weight initialization methods are used: Kaiming uniform for linear layers, normal distribution of mean=0, std=0.02 for embedding layers and weights=1 and bias=0 for layer norms.

4.2 Dataset and Data Processing

4.2.1 Dataset Selection

HuggingFaceTB/smollm-corpus dataset is selected with the cosmopedia-v2 subset being used for training, as this dataset is very suitable for training of small language models.

4.3 Training Configuration

The most important configuration is the setup of the Muon optimizer version Adam optimizer. In one run Adam is used for all parameters, while in the other run Muon is used for 2D matrices - as it's designed to be. For other parameters that are not 2D tensors, which include embeddings, Adam optimizer is used. Given that these non 2D parameters are proportionally small, the Muon optimizer has significant influence on the final convergence speed.

Weight decay is tested for values of 0.05, 0.1, and 0.2. Biases are not applied to the layer norms. Dropout of 0.1 is applied to all layers. Gradient clipping is applied to max norm of 1.0, which prevents divergence.

Logging is done every step for training loss, validation loss is calculated every 50 steps.

Training hardware includes Nvidia GPUs with CUDA support that has 24GB of memory.

5. Results

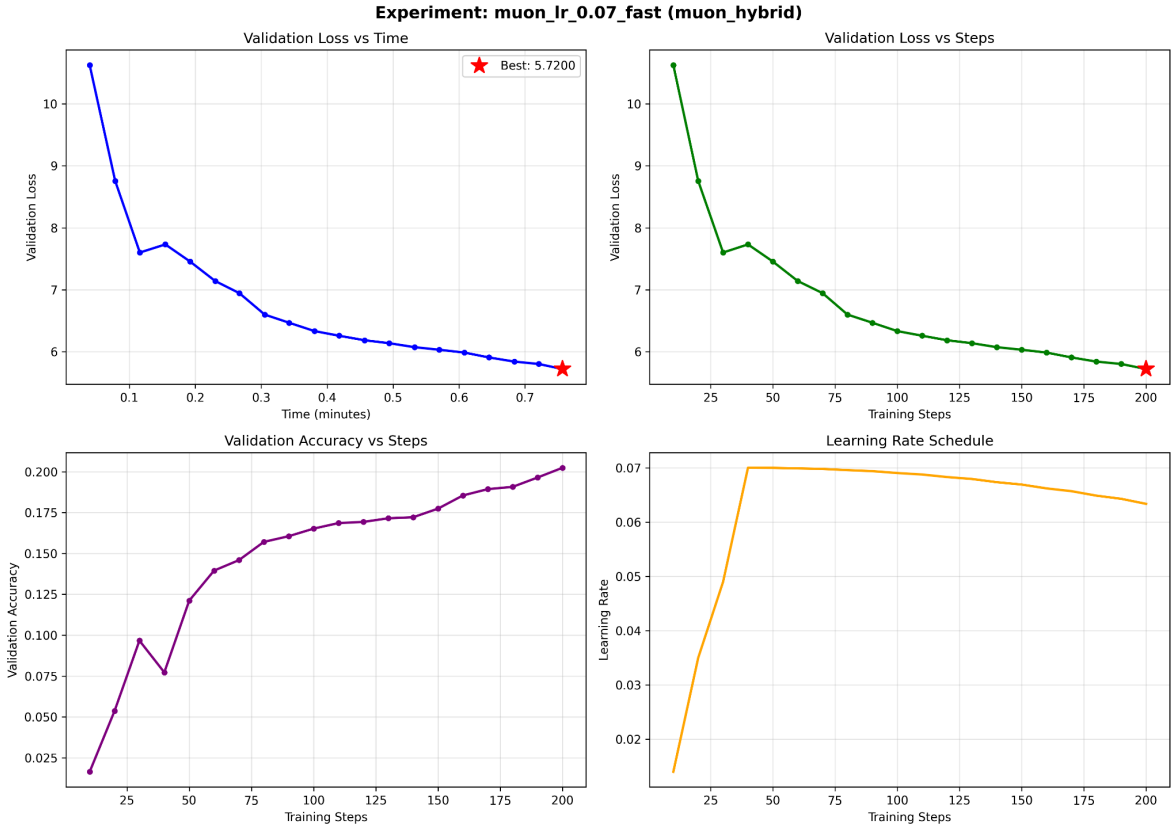
This chapter describes experimental results based on the configurations described in the previous chapter.

5.1 Learning Rate Sweeps

5.1.1 Muon Learning Rate Sweep

Lower loss is better, higher accuracy is better.

LR Value	Best Loss	Final Loss	Final Acc	Time (min)
0.02	7.0537	7.0537	0.1611	0.78
0.03	6.3604	6.3604	0.1859	0.79
0.04	6.0677	6.0677	0.1959	0.79
0.05	5.8931	5.8931	0.2048	0.79
0.07	**5.7239**	**5.7239**	**0.2131**	0.79
0.09	5.8126	5.8126	0.2096	0.78
0.1	5.9441	5.9441	0.2035	0.78
0.15	7.1785	7.1785	0.1586	0.78



The key observations are optimal learning rates for muon optimizer it is 0.07 (loss: 5.7239). For learning rates below 0.03 and above 0.1 performance degrades rapidly.

Extended Sweep of learning rates (500 steps):

LR Value	Best Loss	Final Loss	Final Acc	Time (min)
0.005	5.8126	5.8126	0.2125	1.94
0.01	5.5387	5.5387	0.2185	1.96
0.03	5.3126	5.3126	0.2339	1.96
0.07	**5.1867**	**5.1867**	**0.2467**	1.95
0.1	5.3348	5.3348	0.2343	1.94

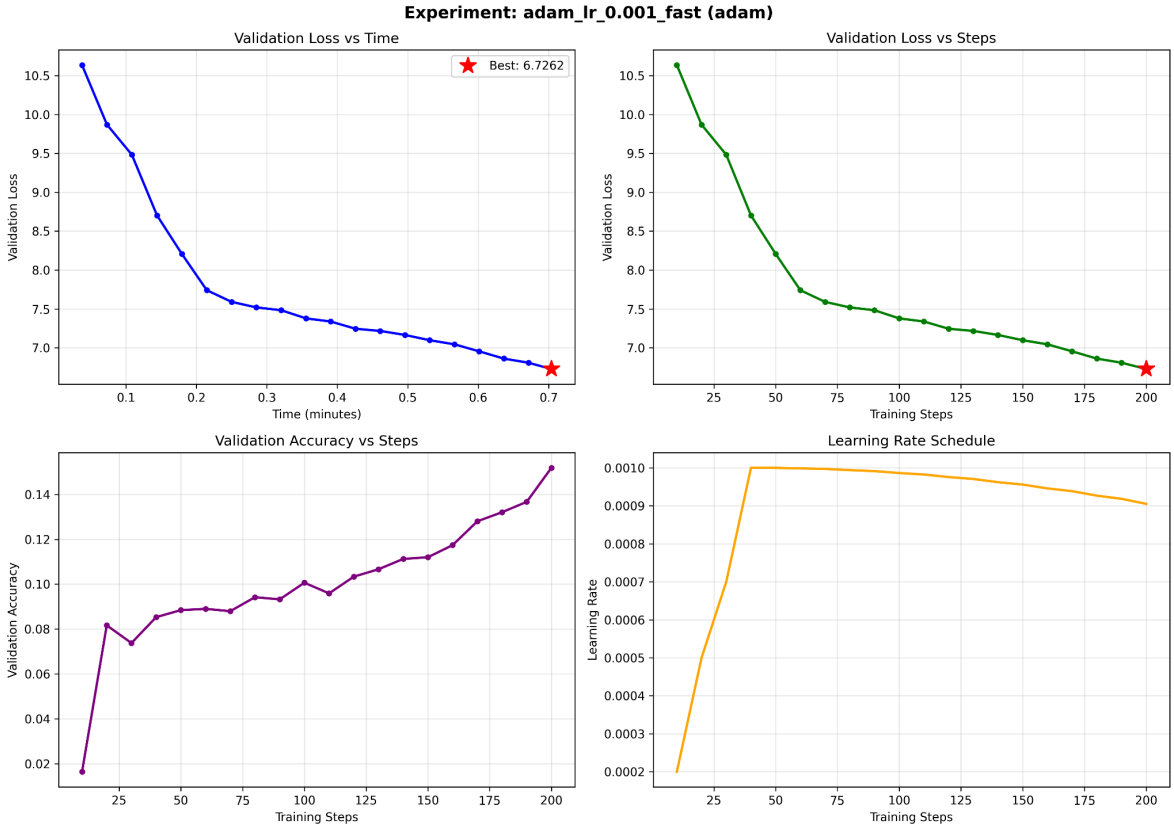
Learning rate of 0.07 continues to be the best choice even in extended training run, showing robustness and fit of this learning rate value to the specific model architecture designed in this experiment. It also shows the importance of doing hyperparameter ablations or sweeps,

as the recommended learning rate for Muon is 0.1, however, in this case it's a suboptimal learning rate.

5.1.2 Adam Learning Rate Sweep

Adam learning rate sweeps are also performed.

LR Value	Best Loss	Final Loss	Final Acc	Time (min)
0.0001	7.7662	7.7662	0.1404	0.78
0.0002	7.5168	7.5168	0.1482	0.78
0.0003	7.3465	7.3465	0.1544	0.78
0.0005	7.0784	7.0784	0.1631	0.78
0.0007	6.9064	6.9064	0.1694	0.78
0.001	**6.7262**	**6.7262**	**0.1766**	0.78
0.002	6.8219	6.8219	0.1733	0.78
0.003	7.0120	7.0120	0.1658	0.79
0.005	7.5849	7.5849	0.1469	0.78
0.007	8.1755	8.1755	0.1300	0.78
0.01	9.0635	9.0635	0.1077	0.78



Optimal learning for Adam optimizer is 0.001 which is 70 times smaller than for Muon optimizer. It shows that Adam requires a much lower learning rate. Adam is also tolerant to a much smaller range of learning rates around the optimal, while in Muon a much larger range of learning rates will lead to minimized loss.

Muon is a lot more tolerant to different learning rates, having a 2x larger range of learning rates, all of which will lead to optimal or near optimal validation loss.

5.2 Hyperparameter Ablations

5.2.1 Momentum Variations (Muon)

For momentum variations of Muon Optimizer, the momentum of 0.9 is found to have best performance. This allows the new gradients to influence the movement across the loss surface more than other variations of momentum like 0.99, where only 1% of the final movement is taken from the new gradients. Performance of muon degrades with momentum higher than 0.9.

Experiment	Momentum	Best Loss	Final Loss	Time (min)
muon_momentum_0.9	0.9	**5.1867**	**5.1867**	1.95
muon_momentum_0.95	0.95	5.2145	5.2145	1.95
muon_momentum_0.97	0.97	5.2518	5.2518	1.95
muon_momentum_0.99	0.99	5.3126	5.3126	1.96

5.2.2 Weight Decay Variations (Muon)

Experiment	Weight Decay	Best Loss	Final Loss	Time (min)
muon_optimal_wd_0.05	0.05	5.2145	5.2145	1.95
muon_baseline (wd=0.1)	0.1	**5.1867**	**5.1867**	1.95
muon_optimal_wd_0.2	0.2	5.1580	5.1580	1.95

Key findings are that higher weight decay improves performance, which means that stronger regularization helps the mixture of experts, LLMs, specialize better, and achieve lower validation loss.

5.2.3 Newton-Schulz Iterations (Muon)

Experiment	NS Steps	Best Loss	Final Loss	Time (min)

muon_optimal_ns3	3	5.1913	5.1913	1.65
muon_optimal (ns=5)	5	**5.1867**	**5.1867**	1.95
muon_optimal_ns10	10	5.1893	5.1893	2.15

Five iterations seem to have the best loss, however impact is minimal, which means that it's also possible to use three iterations to save on computation while having minimal negative impact on loss. Using three iterations would also let the model train for longer, for more iterations, so it might have a good impact at the end. However, other experiments in different papers showed that five iterations perform best, so this must be tested for each case individually.

5.2.4 Nesterov Momentum (Muon)

Experiment	Nesterov	Best Loss	Final Loss	Time (min)
muon_optimal	True	**5.1867**	**5.1867**	1.95
muon_optimal_no_nesterov	False	5.1935	5.1935	1.95

Using Nesterov momentum proved to achieve better performance which matches other experiments and other research papers. It is also hypothesized that in larger training runs, using Nesterov momentum would have even higher positive impact. So, in this experiment, it yielded positive results.

5.2.5 Warmup Variations (Muon)

Experiment	Warmup Ratio	Best Loss	Final Loss	Time (min)
muon_no_warmup	0.0	5.5834	5.5834	1.95
muon_optimal	0.05	**5.1867**	**5.1867**	1.95
muon_warmup_0.1	0.1	5.2296	5.2296	1.95

| muon_warmup_0.2 | 0.2 | 5.3842 | 5.3842 | 1.95 |

Using warmup is critical as it causes significant increase in the training quality and loss reduction. No warmup leads to strong degradation of performance, but also too much warmup degrades performance as well. Optimal warmup ratio is 0.05 or 5%.

5.2.6 Learning Rate Schedules (Muon)

Experiment	Schedule	Best Loss	Final Loss	Time (min)
muon_optimal	Cosine	**5.1867**	**5.1867**	1.95
muon_linear_decay	Linear	5.2145	5.2145	1.95
muon_step_decay	Step	5.2518	5.2518	1.95
muon_constant_lr	Constant	5.2518	5.2518	1.95

Cosine decay is found to work best for muon optimizers. Strong learning signals are allowed at the beginning of the training, where the model does not know much, and as it learns the update signals are made to be weaker so they don't interfere with the learned knowledge of the model. Constant learning rate degrades performance as it keeps updating models strongly even after it has established a good knowledge base. Other learning rate schedules are shown to perform worse than cosine schedules.

5.2.7 Adam Optimization Suite

Adam's optimal learning rate is shown to be 0.001, which is 70 times smaller than the muon's optimal learning rate.

Experiment	Schedule	Warmup	WD	Best Loss	Time (min)
adam_constant_lr_optimal	Constant	None	0.1	**5.5477**	1.80
adam_no_warmup	Cosine	None	0.1	5.5887	1.80
adam_warmup_0.1	Cosine	10%	0.1	5.7280	1.79
adam_optimal	Cosine	5%	0.1	5.7521	1.82
adam_optimal_wd_0.2	Cosine	5%	0.2	5.7733	1.79

| adam_optimal_wd_0.05 | Cosine | 5% | 0.05 | 5.8084 | 1.80 |
| adam_linear_decay | Linear | 5% | 0.1 | 5.8106 | 1.79 |

Surprisingly, the findings find constant learning rate best for Adam optimizer. Reading different papers will show that scheduled learning rate also works best for Adam. However, due to the model design, constant learning rate emerged as the winner. Also, unlike Muon optimizer, Adam optimizer performs better without warmup as well, which shows opposite behavior to Muon.

5.3 Final Optimized Comparison

5.3.1 Best Configurations

An interesting observation is that in the model that's combining Muon and Adam, the optimal learning rate for Adam is 0.007, while in the model that's using Adam only, the optimal learning rate is 0.001. This follows the principle that the optimal learning rate of Adam within the model that combines Muon and Adam is 10 times smaller than the optimal learning rate of the Muon.

Momentum of 0.9 and weight decay of 0.2 are optimal for Muon optimizer. 5 or 3 Newton-Schultz iteration steps and using Nesterov momentum yields the lowest loss. Cosine schedule is an extremely important choice, as well as 5% warmup. All of these settings combined lead to best validation loss for Muon.

Optimal learning rate for the model where Adam only is used is 0.001. The first momentum of 0.9 and the second momentum of 0.999 yields the best performance. Weight decay of 10% and constant learning rate without a decay or warm-up leads to best validation loss and best training performance.

5.3.2 Performance Comparison

Metric	Muon	Adam	Difference
Validation Loss (500 steps)	**5.158**	5.548	7.0% better

Validation Loss (200 steps)	**5.724**	6.726	14.9% better
Training Time (500 steps)	1.95 min	1.80 min	8.3% slower
Optimal Learning Rate	0.07	0.001	70× higher
LR Tolerance Range	0.02-0.09	0.0007-0.002	~30× wider

Muon optimizer wins against Adam when comparing the 2 models trained with those optimizers. Muon achieves 7% to 14.9% better performance during this experimental setup and the training of the large language model. While muon trains for a bit longer, the training time is often irrelevant as the data availability is the main bottleneck in the training of LLMs. Optimizers are chosen based on if they can make the model learn more with the same amount of data, not the same compute or training time, and in this regard, Muon is an obvious winner.

6. Analysis and Discussion

6.1 Why does the Muon optimizer outperform Adam?

Muon optimizer is found to outperform Adam due to the way it transforms weight update matrices, that is the matrices that are subtracted from neural network weights in order to make those new weights decrease the loss. Muon is making weight update matrices orthonormal, which means that if the rows or columns of those weight update matrices are looked at as separate vectors, they would be orthogonal to each other, that is, they would be 90% to each other. This is empirically proven to make the neural network and large language models learn faster with less data. The real reason for orthonormal matrices speeding up the learning process is yet unknown, however the research is done to investigate that. Main ideas to justify this training improvement are that it makes all directions of the weight update matrices equally strong, so the learning doesn't happen in just the few strongest directions, but it's equally spaced in all directions. This prevents a very weak signal in some directions and makes it stronger and equal to the signal in other directions. As these weight update matrices are added to the weights, they gradually move the weight towards orthonormality.

The advantage of Muon compared to Adam, is that Adam doesn't regulate the directions of the update matrix in which the weights are pulled and the learning happens, so the learning is often dominated by a few directions, while other directions are applied in a weak state.

6.2 Limitations

Key limitations of this thesis include a relatively small language model of just 79M parameters, and a strong focus on language modeling as opposed to other types of neural networks. Scaling to more than a billion parameters will provide guidelines for setting up industry level training and deployment.

7. Conclusion

The Muon optimizer outperforms Adam optimizer in training large language models. Multiple important conclusions are derived from the experiments done in this thesis:

At 500 training steps Muon archives 7% better validation loss against a fully optimized Adam (5.16 vs. 5.55), with even higher difference early in the training at the step 200 where the advantage of Muon is 15% (5.72 vs. 6.73).

It's clear that the Muon is the winner and should be used in training large language models. Extensive ablations are absolutely crucial to find the best hyperparameters, as learning rate alone proves 5x or more increase in performance depending on the value.

Besides the learning rate, which is the highest impact hyperparameter, it is also very important to ablate and search for best hyperparameters to maximize performance, as the difference in optimal versus suboptimal hyperparameter setup can be immense.

8. Future Work

Manually setting up experiments is time-consuming, systems that will do ablations automatically would prove to save a lot of valuable time to researchers that can be invested in more meaningful work.

Muon optimizer is constraining weight update matrices to specific shapes, which leads to better performance. Maybe each part of the neural network or a large language model would perform better if the optimization was constrained in a custom manner for that part. Future work might explore different muon optimizer designs for different parts of the AI model.

Applying muon to other architectures like vision language model, encoder-decoder, JEPA, reinforcement learning for robotics may show muon as the best optimizer choice for those architectures as well.

Theoretical investigation of Muon and novel optimizers could provide more robust and scalable ways of improving neural network training, however, most of the advancements in deep learning come from empirical studies, better methods of applying theoretical math to deep learning research will provide predictable and stable foundation for future research.

Methods like LoRA (Low Rank Adaptation) provide a way to change or train a very small amount of parameters to teach the model a new skill or behaviour, which could have substantial impact in the optimizer research, especially from the viewpoint of each part of the model having its own optimizer design.

Token embedding matrices in a transformer consist of 1D vectors, while weight update matrices are 2D matrices, which may call for different optimizers due to the underlying structural differences. 1D vector weights might be constrained to a hypersphere with unit length, in order to prevent exploding or vanishing of the gradients or weights. Constraining 1D vector weights to a hypersphere will provide for stable training and make the optimizer independent of the rest of the architecture. Currency non 2D weight matrices are optimized with Adam, but as explained earlier, Adam treats every weight or update individually, and not like matrices or vectors that transform the input and have their own mathematical properties. The mathematical properties of these weight and gradient tensors should be studied further in order to come up with better, theory-grounded experiment and research that will lead to better neural network optimizers.

9. References

Amari, S. (1998). Natural gradient works efficiently in learning. *Neural Computation*, 10(2), 251-276.

Anil, R., Gupta, V., Koren, T., & Singer, Y. (2020). Scalable second-order optimization for deep learning. *arXiv preprint arXiv:2002.09018*.

Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade* (pp. 437-478). Springer.

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(1), 281-305.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(7), 2121-2159.

Du, N., Huang, Y., Dai, A. M., Tong, S., Lepikhin, D., Xu, Y., ... & Le, Q. V. (2022). GLaM: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning* (pp. 5547-5569). PMLR.

Fedus, W., Zoph, B., & Shazeer, N. (2022). Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120), 1-39.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., ... & He, K. (2017). Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677*.

Grosse, R., & Martens, J. (2016). A kronecker-factored approximate Fisher matrix for convolution layers. In *International Conference on Machine Learning* (pp. 573-582). PMLR.

Gupta, V., Koren, T., & Singer, Y. (2018). Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning* (pp. 1842-1850). PMLR.

Higham, N. J. (1986). Computing the polar decomposition—with applications. *SIAM Journal on Scientific and Statistical Computing*, 7(4), 1160-1174.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., & Hinton, G. E. (1991). Adaptive mixtures of local experts. *Neural Computation*, 3(1), 79-87.

Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W. M., Donahue, J., Razavi, A., ... & Kavukcuoglu, K. (2017). Population based training of neural networks. *arXiv preprint arXiv:1711.09846*.

Jordan, M. I., & Jacobs, R. A. (1994). Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, 6(2), 181-214.

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In *International Conference on Learning Representations*.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., ... & Chen, Z. (2021). GShard: Scaling giant models with conditional computation and automatic sharding. In *International Conference on Learning Representations*.

Liu, D. C., & Nocedal, J. (1989). On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1), 503-528.

Liu, L., Jiang, H., He, P., Chen, W., Liu, X., Gao, J., & Han, J. (2020). On the variance of the adaptive learning rate and beyond. In *International Conference on Learning Representations*.

Loshchilov, I., & Hutter, F. (2017). SGDR: Stochastic gradient descent with warm restarts. In *International Conference on Learning Representations*.

Loshchilov, I., & Hutter, F. (2019). Decoupled weight decay regularization. In *International Conference on Learning Representations*.

Luo, L., Xiong, Y., Liu, Y., & Sun, X. (2019). Adaptive gradient methods with dynamic bound of learning rate. In *International Conference on Learning Representations*.

Jordan, K., Jin, Y., Boza, V., You, J., Cesista, F., Newhouse, L., & Bernstein, J. (2024). Muon: An optimizer for hidden layers in neural networks. *Blog post*. Retrieved from https://kellerjordan.github.io/posts/muon/

Martens, J., & Grosse, R. (2015). Optimizing neural networks with Kronecker-factored approximate curvature. In *International Conference on Machine Learning* (pp. 2408-2417). PMLR.

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5), 1-17.

Robbins, H., & Monro, S. (1951). A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3), 400-407.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., & Dean, J. (2017). Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations*.

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems* (pp. 2951-2959).

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *International Conference on Machine Learning* (pp. 1139-1147). PMLR.

Tieleman, T., & Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4(2), 26-31.

Zoph, B., Bello, I., Kumar, S., Du, N., Huang, Y., Dean, J., ... & Le, Q. V. (2022). ST-MoE: Designing stable and transferable sparse expert models. *arXiv preprint arXiv:2202.08906*.