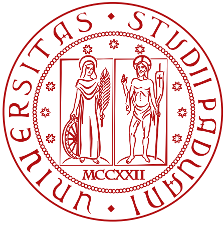
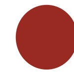


Università degli Studi di Padova



Dipartimento di Ingegneria dell'Informazione

Master Thesis in Computer Engineering

Evaluation of the Performance of the Alternating Direction Method of Multipliers in Artificial Neural Networks

Master Candidate

Ergys Meda

Student ID 2071547

Supervisor

Dr. Jacopo Pegoraro

University of Padova

Academic Year
2024/2025

To my family

Abstract

This thesis presents a comprehensive evaluation of the Alternating Direction Method of Multipliers (ADMM) algorithm for neural network optimization, comparing its performance against traditional gradient-based methods including Gradient Descent (GD) and Stochastic Gradient Descent (SGD). While gradient-based methods dominate deep learning optimization, their limitations in handling complex loss landscapes and distributed computing scenarios motivate the exploration of alternative approaches. ADMM, a technique originally developed for convex optimization problems, offers a promising alternative by decomposing the optimization problem into simpler subproblems.

Through systematic experimentation on the MNIST dataset, this work evaluates ADMM across three key dimensions: computational efficiency, convergence behavior, and scalability depending on its parameters. The results demonstrate that ADMM achieves significantly faster per-epoch processing timesapproximately 3 times faster than GD and 20 times faster than SGD while maintaining reasonable, though lower, final accuracy (82% versus 96% for gradient methods). Further investigation reveals that ADMM's performance is substantially influenced by hyperparameter configuration, with optimal settings identified for the penalty parameter (BETA=1) and activation constraint weight (GAMMA=1-5), as well as by the choice of initialization method, with He and Xavier initialization providing superior results.

Analysis of ADMM's scaling properties indicates improved performance when distributed across 4 parallel processes, demonstrating its potential for large-scale distributed training. The observed trade-off between computational efficiency and final accuracy positions ADMM as a valuable alternative optimization approach for scenarios where processing speed is prioritized over achieving state-of-the-art accuracy, or where distributed computing resources are available but communication overhead must be minimized.

This thesis contributes to the broader understanding of neural network optimization beyond gradient-based paradigms and provides practical guidelines for implementing ADMM in neural network training contexts, opening avenues for hybrid optimization approaches that could potentially combine the computational advantages of ADMM with the accuracy of gradient-based methods.

Contents

List of Figures xi

List of Tables xiii

List of Algorithms xvii

List of Code Snippets xvii

List of Acronyms xix

1 Introduction 1

2 Background and related work 5
- 2.1 Neural Network optimization challenges 5
- 2.2 Computational and generalization challenges 7
- 2.3 Traditional optimization methods 8
  - 2.3.1 Gradient Descent (GD) 8
  - 2.3.2 Stochastic Gradient Descent (SGD) 10
- 2.4 Alternating Direction Method of Multipliers (ADMM) 12
  - 2.4.1 Neural network training formulation 12
  - 2.4.2 ADMM Algorithm for Neural Networks 14
  - 2.4.3 Key ADMM parameters and theoretical advantages 15
- 2.5 Comparative studies and research gaps 16

3 Methodology 19
- 3.1 Dataset 20
  - 3.1.1 MNIST dataset 20
  - 3.1.2 Data preparation 21
- 3.2 Experimental framework 22

3.2.1 Network implementation 22
3.3 Optimization methods 23
3.3.1 ADMM optimization 23
3.3.2 Gradient-based methods 25
3.4 Hyperparameter configuration 26
3.4.1 ADMM hyperparameters 26
3.4.2 Gradient-based hyperparameters 26
3.5 Evaluation methodology 27
3.5.1 Performance metrics and experimental protocols 27
3.5.2 Implementation details 28

4 Results and discussion 31
4.1 Computational efficiency 31
4.1.1 Epoch processing time comparison and implications . . . 31
4.2 Convergence behavior 33
4.2.1 Loss convergence analysis and interpretation 33
4.2.2 Accuracy performance and comparative analysis 35
4.2.3 Final test accuracy and optimal configurations 36
4.3 Scalability analysis and parameter sensitivity 38
4.3.1 ADMM scalability with parallel processing 38
4.3.2 Parameter sensitivity analysis 41

5 Conclusions and future work 51
5.1 Summary of key findings 51
5.2 Limitations 52
5.3 Practical implications 53
5.4 Future work 54

References 57

Acknowledgments 63

Declaration 65

List of Figures

2.1 Comparison of neural network loss landscapes from Li et al. [33]: (a) ResNet-110 without skip connections exhibits a highly non-convex, rugged terrain with numerous local minima, steep barriers, and sharp transitions. (b) DenseNet with 121 layers presents a much smoother, more bowl-shaped landscape that is considerably easier to optimize. 6

3.1 Methodology pipeline for evaluating ADMM performance in neural networks 20
3.2 Sample images from the MNIST dataset showing handwritten digits from 0 to 9. 21
3.3 Neural network architecture with ADMM variables, showing the relationship between weight matrices (W_1, W_2, W_3) and auxiliary variables (z_1, a_1, z_2, a_2, z_3). 23

4.1 Epoch processing time comparison between ADMM, SGD, and GD optimization algorithms. 32
4.2 Loss function values per epoch for ADMM, SGD, and GD optimization algorithms. 34
4.3 Accuracy values per epoch for ADMM, SGD, and GD optimization algorithms. 35
4.4 Final test accuracy for different configurations of ADMM, SGD, and GD optimization algorithms. 37
4.5 Training accuracy of ADMM with 2 parallel processes across epochs. 39
4.6 Test accuracy of ADMM with 2 parallel processes across steps. . . 39
4.7 Test accuracy of ADMM with 4 parallel processes across steps. . . 40
4.8 Test accuracy of ADMM with 8 parallel processes across steps. . . 40

4.9 Epoch processing time for ADMM with different BETA values (1, 5, and 10) using 4 parallel processes. 42
4.10 Test accuracy for ADMM with different BETA values (1, 5, and 10) using 4 parallel processes. 43
4.11 Training accuracy for ADMM with different GAMMA values (1, 5, and 10) using 4 parallel processes and BETA=1. 44
4.12 Final test accuracy for ADMM with different GAMMA values (1, 5, and 10) using 4 parallel processes and BETA=1. 44
4.13 Memory usage (RAM in MB) for ADMM with different GAMMA values (1, 5, and 10) using 4 parallel processes and BETA=1. . . . 45
4.14 Test accuracy for ADMM with different WARMUP values (0, 2, and 5) using 4 parallel processes. 46
4.15 Training accuracy for ADMM with different initialization methods: Normalized, He, Xavier, and Zero. 48
4.16 Final test accuracy for ADMM with different initialization methods. 49

List of Tables

3.1 ADMM hyperparameter configuration 27
3.2 Gradient-based methods hyperparameter configuration 27

4.1 Optimal hyperparameter configurations and corresponding test accuracy for each optimization algorithm 37

List of Algorithms

1 ADMM for Neural Nets [45] 14

List of Code Snippets

3.1 Weight update implementation 24
3.2 Distributed weight update implementation 24

List of Acronyms

ADMM	Alternating Direction Method of Multipliers
SGD	Stochastic Gradient Descent
GD	Gradient Descent
NN	Neural Network
RAM	Random Access Memory
MB	Megabyte
GPU	Graphics Processing Unit
CPU	Central Processing Unit
BETA	Penalty Parameter in ADMM
GAMMA	Step Size Parameter in ADMM

1 Introduction

Neural networks have revolutionized machine learning, achieving remarkable success across diverse fields including computer vision [31], natural language processing [14], and reinforcement learning [42]. The training of these networks, however, presents significant computational challenges due to their complex, non-convex optimization landscapes and the ever-increasing scale of both models and datasets [19]. This has prompted extensive research into optimization algorithms that can efficiently train neural networks while maintaining or improving convergence properties.

Gradient Descent (GD) and its variants have long been the de facto standard for training neural networks [41]. These first-order methods rely on gradient information to iteratively update the model parameters, with various enhancements such as momentum [44], adaptive learning rates [15, 30], and batch normalization [27] that improve their performance. Despite these advances, GD-based methods face fundamental limitations in handling complex loss landscapes and network parameterization choices [50].

The Alternating Direction Method of Multipliers (ADMM) represents a promising alternative optimization approach originating from the field of convex optimization [8]. Initially developed for convex problems with separable structures, ADMM has earned attention for its ability to decompose complex optimization problems into simpler subproblems that can be solved more efficiently and potentially in parallel [36]. While ADMM has been successfully applied to various machine learning tasks such as regularized regression [48], support vector machines [21], and sparse representation [37], comprehensive

evaluations of its performance for training standard neural networks, particularly in comparison to conventional optimization methods like GD and SGD, remain limited in the literature.

Recent work by Taylor et al. [45] demonstrated that neural networks can be effectively trained using ADMM without traditional gradient descent steps. Their approach decomposes the network training problem into a sequence of simpler subproblems that can each be solved globally in closed form, avoiding many of the challenges associated with gradient-based methods such as saturation effects and poor conditioning. Moreover, their implementation exhibited strong linear scaling in distributed environments, highlighting ADMM's potential advantage for large-scale neural network training. Despite these promising results, comprehensive evaluations of ADMM's performance across different network architectures, hyperparameter configurations, and optimization scenarios are still needed to fully understand its capabilities and limitations compared to conventional methods.

In this thesis, we conduct a thorough evaluation of ADMM's performance in training standard neural networks through a series of systematic experiments. The research focuses on three key dimensions: computational efficiency, where we compare ADMM's epoch processing time against traditional methods; convergence behavior, analyzing both loss trajectories and accuracy metrics; and scalability, examining ADMM's performance across different parallel processing configurations and parameter settings. Unlike previous studies that primarily focused on ADMM's theoretical advantages, we provide extensive empirical evidence of how specific parameters (BETA, GAMMA, WARMUP) and initialization methods affect performance in practice. The comprehensive nature of our evaluation offers valuable insights for practitioners on when and how to effectively implement ADMM for neural network training, especially in distributed computing environments. Our findings demonstrate that with proper configuration, ADMM can offer significant advantages in terms of computational efficiency while maintaining competitive accuracy, particularly for large-scale neural network training tasks.

The thesis is organized as follows: **Chapter 1** introduces the research problem, outlines the motivation for exploring alternative optimization methods for neural networks, and establishes the research questions and objectives.

Chapter 2 provides a comprehensive background and review of related work, covering the challenges in neural network optimization, traditional gradient-

based methods, the theoretical foundations and development of ADMM with particular focus on Taylor et al.'s approach, and existing applications of ADMM in various machine learning contexts.

Chapter 3 details the methodology employed in this research, including the experimental design, implementation details of the neural network architectures, dataset selection and preparation, parameter configurations for the different optimization algorithms, and the evaluation metrics used to assess performance.

Chapter 4 presents the results and discussion of our experimental investigation, organised into sections examining computational efficiency, convergence behaviour, accuracy performance, and scalability analysis. This chapter includes detailed comparisons between ADMM, SGD, and GD across multiple performance dimensions.

Chapter 5 concludes the thesis by summarising the key findings, acknowledging limitations of the current study, discussing practical implications for practitioners, and suggesting promising directions for future research that could extend this work and address the identified challenges.

Background and related work

This chapter provides the theoretical foundation and context for evaluating ADMM's performance in neural network optimization. We begin by examining the fundamental challenges in neural network optimization, followed by an overview of traditional gradient-based methods and their limitations. We then present the mathematical formulation of ADMM based on Taylor et al.'s approach [45], and discuss the significance of various ADMM parameters. Finally, we review existing comparative studies and applications, highlighting the research gaps this thesis aims to address.

2.1 Neural Network optimization challenges

Neural network training fundamentally relies on the concept of loss functions, which quantify the discrepancy between a model's predictions and the ground truth. As Goodfellow et al. [19] explain, the optimization problem in machine learning is formulated by minimizing the expected loss on the training set, replacing the true data distribution with the empirical distribution defined by the training examples. These functions provide a measurable objective that optimization algorithms seek to minimize. Common loss functions include mean squared error for regression tasks and cross-entropy for classification problems. The value of this loss function, computed across training examples, creates a high-dimensional surface known as the loss landscape. The central challenge in neural network optimization stems from the highly non-convex nature of these loss landscapes. Unlike convex optimization problems with a single global min-

imum, neural network loss functions contain numerous local minima, saddle points, and flat regions that complicate the optimization process [19]. Recent theoretical work has provided deeper insights into these landscapes. Dauphin et al. [12] demonstrated that in high-dimensional spaces typical of neural networks, saddle points rather than local minima represent the primary challenge for optimization algorithms. These saddle points create regions where gradients approach zero in some directions but not others, causing gradient-based methods to significantly slow down. Visualizations of neural network loss landscapes by Li et al. [33] revealed how network depth and width affect the geometry of these landscapes. They found that deeper networks tend to have sharper local minima, while width and skip connections (as in residual networks) tend to smooth the landscape. This geometric complexity directly impacts optimization algorithm performance, with some methods better equipped to navigate these challenging terrains than others. As shown in Figure 2.1, the loss landscape

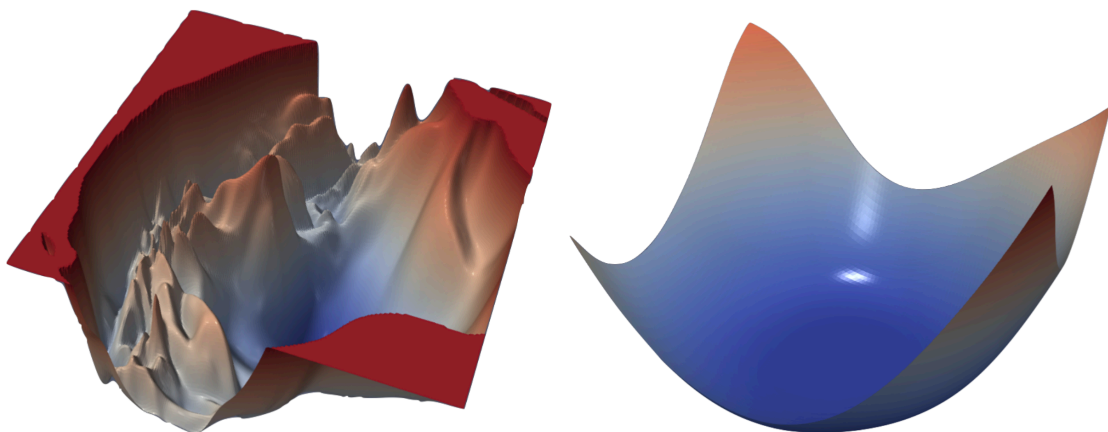


Figure 2.1: Comparison of neural network loss landscapes from Li et al. [33]: (a) ResNet-110 without skip connections exhibits a highly non-convex, rugged terrain with numerous local minima, steep barriers, and sharp transitions. (b) DenseNet with 121 layers presents a much smoother, more bowl-shaped landscape that is considerably easier to optimize.

of neural networks can vary dramatically depending on architectural choices. The rugged, complex landscape of networks without skip connections creates particular challenges for gradient-based optimization, as algorithms can easily become trapped in poor local minima or slow down significantly near saddle points. These visualization techniques provide crucial insights into why certain optimization algorithms may perform better than others on different network ar-

chitectures. Another fundamental challenge specific to training neural networks is the vanishing and exploding gradient problem, which directly impacts optimization effectiveness. During backpropagation, gradients must be propagated through multiple layers. In deep networks with certain activation functions, these gradients can either decay exponentially (vanish) or grow exponentially (explode) as they propagate backward. The vanishing gradient problem affects both recurrent and feed-forward neural networks. While initially identified for recurrent networks by Hochreiter [25] and Bengio et al. [5], its impact on deep feed-forward networks was thoroughly analyzed by Glorot and Bengio [18], who proposed weight initialization strategies to mitigate this issue. While architectural innovations such as residual connections [23], careful initialization strategies [24, 18], and normalization techniques [27] have mitigated these issues, they remain fundamental challenges for gradient-based optimization methods. Any optimization approach for neural networks must contend with these gradient pathologies, either by addressing them directly or circumventing gradient calculation altogether.

2.2 Computational and generalization challenges

Modern neural networks often contain millions or even billions of parameters, trained on massive datasets. Recent models like GPT-3 [9] and vision transformers [10] have pushed these boundaries even further. The scale of these problems introduces significant computational challenges, making optimization efficiency a critical concern. The computational burden of neural network training manifests in two dimensions: time and resources. Training large models can take days or weeks even with specialized hardware like GPUs or TPUs. The growing computational demands have led to concerns about both economic costs and environmental impact [47]. These scalability challenges have motivated research into distributed and parallel optimization strategies [4]. However, many optimization methods scale poorly in distributed environments due to communication overhead and synchronization requirements. Recent comprehensive surveys by Mayer and Jacobsen [35] have highlighted that communication between computational nodes remains a significant bottleneck in distributed training, often limiting the scaling efficiency of parallelization approaches. Furthermore, memory constraints can limit batch sizes and model complexity, forcing practitioners to use sophisticated techniques like pipeline

parallelism [26], gradient compression [34], and memory-efficient optimizers [39]. Optimization in neural networks also presents an interesting paradox: the ultimate goal is not merely to minimize the training loss but to develop models that generalize well to unseen data. Recent research has revealed complex relationships between optimization choices and generalization performance. Work by Zhang et al. [52] demonstrated that neural networks can easily memorize random labels, suggesting that the optimization objective alone may not guide models toward meaningful generalizations. Keskar et al. [29] found that small-batch methods often generalize better than large-batch methods, despite the latter achieving lower training loss. This suggests that how we optimize can be as important as what we optimize. More recent work by Charles et al. [2] and Lei et al. [32] has begun to establish theoretical connections between optimization and generalization in deep learning, building on earlier foundational work by Arora et al. [3] and Allen-Zhu et al. [1]. These findings indicate that optimization algorithms that induce certain implicit biases may lead to solutions with better generalization properties. These challenges include complex optimization problems, issues with gradients, scaling difficulties, and the balance between training accuracy and real-world performance. Because traditional methods struggle with these problems, researchers are looking for new ways to improve optimization. Methods like ADMM provide a different approach that can handle these challenges better.

2.3 Traditional optimization methods

This section examines the foundational gradient-based optimization algorithms widely used in neural network training. We analyze their mathematical formulations, theoretical properties, and practical limitations, providing context for understanding ADMM's potential advantages in neural network optimization.

2.3.1 Gradient Descent (GD)

Gradient Descent represents the cornerstone of neural network optimization, serving as the basis for most modern optimization algorithms [41]. Its formulation follows directly from calculus principles, iteratively updating parameters in the direction of steepest descent.

Gradient Descent is an iterative first-order optimization algorithm for finding the minimum of a differentiable function. To find a local minimum, GD takes repeated steps in the opposite direction of the gradient at the current point, as this is considered the direction of steepest descent [22]. Conversely, stepping in the gradient's direction would result in a local maximum of the function. In the context of neural networks, we seek to minimize a loss function $J(\theta)$ parameterized by $\theta \in \mathbb{R}^d$.

The core principle of gradient descent is to iteratively adjust parameters in the direction of steepest descent of the objective function. The update rule is defined as:

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t) \tag{2.1}$$

where:

- θ_t represents the parameter vector at iteration t
- $\eta > 0$ is the learning rate (step size)
- $\nabla J(\theta_t)$ is the gradient of the objective function evaluated at θ_t

For neural networks, the objective function typically takes the form of an empirical risk minimization problem:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f_\theta(x_i), y_i) + \lambda \mathcal{R}(\theta) \tag{2.2}$$

where $\mathcal{L}$ represents the loss function, f_θ is the neural network with parameters θ, (x_i, y_i) are training examples, $\mathcal{R}(\theta)$ is a regularization term, and λ controls the regularization strength [19].

Strengths and limitations

Gradient Descent's popularity in optimization comes primarily from its simplicity, as emphasized by Boyd and Vandenberghe [7]. For convex objective functions, GD provides robust convergence guarantees, reliably reaching the global minimum when paired with an appropriate learning rate schedule. This theoretical soundness combined with its straightforward implementation makes GD a reliable choice for well-structured optimization problems.

However, despite its strengths, Gradient Descent exhibits significant limitations. Its sensitivity can lead to extremely slow convergence, particularly for

problems with condition numbers exceeding 100. In more challenging scenarios, with condition numbers approaching 1000 or higher, the gradient method can become so inefficient that it becomes practically unusable. Computational considerations further complicate the method's effectiveness. While an exact line search can potentially enhance convergence, it often introduces substantial computational overhead. The method may require an impractical number of iterations to converge for poorly conditioned problems [7].

2.3.2 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is an iterative first-order optimization algorithm that modifies the standard gradient descent approach by computing the gradient based on a single randomly selected training example or a small batch of examples in each iteration. The term "stochastic" refers to a mechanism connected to random possibility; therefore, instead of using the entire dataset for each iteration, a few samples are randomly chosen [22]. SGD aims to find the global minimum by changing the network structure after each training stage. This approach reduces the error by approximating the gradient for a randomly chosen batch instead of finding the gradient for the whole dataset. In practice, random sampling is achieved by randomly shuffling the dataset and moving stepwise through batches [22]. This stochastic approximation allows for faster updates and improved efficiency, especially when working with large datasets.

In the context of neural networks, we aim to minimize a loss function $J(\theta)$ parameterized by $\theta \in \mathbb{R}^d$. The core principle of SGD is to iteratively adjust the parameters using an approximation of the full gradient. The update rule for a single training example is given by:

$$\theta_{t+1} = \theta_t - \eta \nabla \mathcal{L}(f_\theta(x_i), y_i) \tag{2.3}$$

where:

- θ_t represents the parameter vector at iteration t
- $\eta > 0$ is the learning rate (step size)
- (x_i, y_i) is a randomly selected training example
- $\nabla \mathcal{L}(f_\theta(x_i), y_i)$ is the gradient of the loss computed on the selected example

However, in practice, a more common variant is **mini-batch Stochastic Gradient Descent** [38], where the gradient is computed over a small subset (mini-batch) of the training data:

$$\theta_{t+1} = \theta_t - \eta \frac{1}{m} \sum_{i=1}^{m} \nabla \mathcal{L}(f_\theta(x_i), y_i) \tag{2.4}$$

where m is the mini-batch size.

For neural networks, the objective function follows the empirical risk minimization framework:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f_\theta(x_i), y_i) + \lambda \mathcal{R}(\theta) \tag{2.5}$$

where:

- $\mathcal{L}$ represents the loss function
- f_θ is the neural network with parameters θ
- (x_i, y_i) are training examples
- $\mathcal{R}(\theta)$ is a regularization term
- λ controls the regularization strength

The key distinction from batch gradient descent is that SGD approximates the true gradient by computing it on a single example (or a small mini-batch), trading some accuracy for:

- Faster and more memory-efficient updates
- The ability to escape sharp local minima due to added noise in updates
- More frequent updates, leading to faster convergence in certain cases

This stochasticity introduces some variance in updates, which can be beneficial in avoiding poor local minima and improving generalization [40, 19]. Despite its strengths, SGD faces critical challenges. A significant issue is the high variance in gradient estimates, which can lead to unstable convergence and oscillations around optimal solutions [6, 20]. The method is particularly sensitive to hyperparameter choices. For instance, Jastrzbski et al. demonstrated that the critical control parameter is not the batch size or learning rate independently, but their ratio [28]. This finding illustrates the delicate balance required in SGD

optimization: small changes in hyperparameters can dramatically alter model performance [43].

To address these limitations, researchers have developed advanced variants like Momentum SGD [44], Adam [30], RMSprop, and AdaGrad[16]. These methods introduce adaptive learning rates and sophisticated gradient scaling to improve convergence and generalization.

2.4 Alternating Direction Method of Multipliers (ADMM)

The Alternating Direction Method of Multipliers (ADMM) represents a fundamentally different approach to neural network optimization compared to gradient-based methods. This section introduces the mathematical foundations of ADMM, with particular focus on the formulation proposed by Taylor et al. [45] for training neural networks without gradient steps.

2.4.1 Neural network training formulation

Taylor et al. [45] presented a detailed application of ADMM to neural network training that completely avoids gradient calculations. Their paper examined how ADMM could be effectively applied to neural network optimization by reformulating the standard training problem to facilitate decomposition. They demonstrated this approach on practical neural network problems, showing how the alternating update structure of ADMM could be leveraged to distribute computation across multiple processors while avoiding the backpropagation of gradients.

Consider a standard feedforward neural network with L layers. Each layer l is defined by a linear operator W_l (typically a weight matrix) and a non-linear activation function h_l. Given a set of input activations a_0 and target outputs y, the network computes a composite function:

$$f(a_0; W) = W_L(h_{L-1}(W_{L-1}h_{L-2}(\dots h_1(W_1 a_0)))) \tag{2.6}$$

where $W = \{W_l\}_{l=1}^{L}$ denotes the set of all weight matrices.

The conventional training problem aims to find weights W that minimize a

loss function ℓ measuring the discrepancy between network outputs and targets:

$$\min_{W} \ell(f(a_0; W), y) \tag{2.7}$$

The key insight in Taylor et al.'s approach is to decouple the weights from the nonlinear activation functions using a splitting technique. Rather than directly feeding the output of layer l into the next layer, they introduce auxiliary variables:

- $z_l = W_l a_{l-1}$ represents the output of the linear operator at layer l
- $a_l = h_l(z_l)$ represents the output of the activation function at layer l

This splitting transforms the original problem into a constrained optimization:

$$\min_{\{W_l\},\{a_l\},\{z_l\}} \quad \ell(z_L, y) \tag{2.8}$$

$$\text{subject to} \quad z_l = W_l a_{l-1}, \quad \text{for } l = 1, 2, \ldots, L \tag{2.9}$$

$$a_l = h_l(z_l), \quad \text{for } l = 1, 2, \ldots, L-1 \tag{2.10}$$

This constrained problem can be addressed by introducing penalty terms to form an unconstrained problem:

$$\min_{\{W_l\},\{a_l\},\{z_l\}} \quad \ell(z_L, y) + \beta_L \|z_L - W_L a_{L-1}\|^2 \tag{2.11}$$

$$+ \sum_{l=1}^{L-1} \left[\gamma_l \|a_l - h_l(z_l)\|^2 + \beta_l \|z_l - W_l a_{l-1}\|^2\right] \tag{2.12}$$

where $\{\beta_l\}$ and $\{\gamma_l\}$ are penalty parameters controlling the weight of each constraint.

To ensure exact enforcement of constraints, Taylor et al. introduce Lagrange multipliers, specifically using a Bregman iteration approach. The final objective becomes:

$$\min_{\{W_l\},\{a_l\},\{z_l\}} \quad \ell(z_L, y) + \langle z_L, \lambda \rangle + \beta_L \|z_L - W_L a_{L-1}\|^2 \tag{2.13}$$

$$+ \sum_{l=1}^{L-1} \left[\gamma_l \|a_l - h_l(z_l)\|^2 + \beta_l \|z_l - W_l a_{l-1}\|^2 \right] \tag{2.14}$$

where λ is a vector of Lagrange multipliers with the same dimensions as z_L.

2.4.2 ADMM Algorithm for Neural Networks

The ADMM optimization proceeds by alternately updating each set of variables while holding the others fixed, following the algorithm outlined by Taylor et al. [45].

Algorithm 1 ADMM for Neural Nets [45]

Input: training features $\{a_0\}$, and labels $\{y\}$
Initialize: allocate $\{a_l\}_{l=1}^{L-1}$, $\{z_l\}_{l=1}^{L}$, and λ
repeat
  for $l = 1, 2, \cdots, L-1$ **do**
    $W_l \leftarrow z_l a_{l-1}^{\dagger}$
    $a_l \leftarrow (\beta_{l+1} W_{l+1}^T W_{l+1} + \gamma_l I)^{-1} (\beta_{l+1} W_{l+1}^T z_{l+1} + \gamma_l h_l(z_l))$
    $z_l \leftarrow \arg\min_z \gamma_l \|a_l - h_l(z)\|^2 + \beta_l \|z_l - W_l a_{l-1}\|^2$
  end for
  $W_L \leftarrow z_L a_{L-1}^{\dagger}$
  $z_L \leftarrow \arg\min_z \ell(z, y) + \langle z_L, \lambda \rangle + \beta_L \|z - W_L a_{L-1}\|^2$
  $\lambda \leftarrow \lambda + \beta_L (z_L - W_L a_{L-1})$
until converged

The key advantage of this approach is that each update step has a simple closed-form solution:

Weight update: The update for each weight matrix W_l is a simple least-squares problem, solved as:

$$W_l \leftarrow z_l a_{l-1}^{\dagger} \tag{2.15}$$

where $a_{l-1}^{\dagger}$ denotes the pseudo-inverse of a_{l-1}.

Activation update: The activation variables are updated by solving another least-squares problem:

$$a_l \leftarrow (\beta_{l+1} W_{l+1}^T W_{l+1} + \gamma_l I)^{-1} (\beta_{l+1} W_{l+1}^T z_{l+1} + \gamma_l h_l(z_l)) \tag{2.16}$$

Output update: The update for z_l involves solving:

$$z_l \leftarrow \arg\min_z \gamma_l \|a_l - h_l(z)\|^2 + \beta_l \|z - W_l a_{l-1}\|^2 \tag{2.17}$$

While this appears complex, Taylor et al. note that when using common activation functions like ReLU, this decomponents into independent one-dimensional problems that can be solved in closed form.

Lagrange Multiplier Update: The multipliers are updated using the Bregman iteration:

$$\lambda \leftarrow \lambda + \beta_L (z_L - W_L a_{L-1}) \tag{2.18}$$

2.4.3 Key ADMM parameters and theoretical advantages

The performance of ADMM for neural network optimization is influenced by several key parameters:

BETA Parameter: The BETA parameter (β_l) controls the penalty weight for the constraint $z_l = W_l a_{l-1}$. This parameter affects the trade-off between satisfying the constraint exactly and minimizing the original objective. Higher values enforce the constraint more strictly but may slow convergence.

GAMMA Parameter: The GAMMA parameter (γ_l) controls the penalty weight for the constraint $a_l = h_l(z_l)$. Similar to BETA, it balances constraint satisfaction with objective minimization.

WARMUP Parameter: Taylor et al. introduced a WARMUP parameter that specifies the number of ADMM iterations to perform before beginning Lagrange multiplier updates. This allows the algorithm to find reasonable initial values before enforcing constraints exactly.

Initialization methods: Various initialization strategies for the activation matrices $\{a_l\}$ and output matrices $\{z_l\}$ can significantly impact ADMM's convergence behavior.

The ADMM approach proposed by Taylor et al. offers several theoretical advantages over gradient-based methods:

Avoiding gradient pathologies: By decoupling the layers, ADMM eliminates the need for backpropagation and thus avoids vanishing/exploding gradient problems entirely. Each weight matrix is updated using only local information, without requiring gradient chains to propagate through multiple layers.

Global solutions to subproblems: Each substep in ADMM is solved to global optimality. While the overall problem remains non-convex, the ability to exactly solve each subproblem may help avoid poor local minima that plague gradient methods.

Parallelization potential: The ADMM formulation enables multiple types of parallelism:

- Data parallelism: The algorithm can be distributed across cores with different subsets of training data
- Model parallelism: Updates for different layers can potentially be computed in parallel

Linear scaling: Taylor et al. demonstrated empirically that their ADMM approach exhibits linear scaling with the number of computational cores, even up to thousands of cores. This contrasts with diminishing returns often observed when scaling gradient-based methods.

2.5 Comparative studies and research gaps

The field of neural network optimization has witnessed extensive research comparing various optimization methods, though comprehensive evaluations including ADMM alongside traditional gradient-based approaches remain limited. Ruder [41] provided a thorough overview of gradient-based optimization

algorithms, showing that while adaptive methods often converge faster, SGD with momentum frequently achieves better generalization. Wilson et al. [49] further demonstrated cases where adaptive methods led to worse generalization despite faster training. Research specifically comparing ADMM with gradient-based methods for neural networks has been sparse. Taylor et al. [45] demonstrated ADMM's potential advantages for neural networks, particularly in distributed environments, showing promising convergence properties and linear scaling with computational nodes. Zeng et al. [51] provided theoretical analysis of ADMM's convergence in neural networks, establishing guarantees under certain conditions and demonstrating its ability to avoid saturation issues that affect gradient-based methods. In the broader optimization literature, Ghadimi et al. [17] and Teixeira et al. [46] analyzed optimal parameter selection for ADMM, showing that convergence rates vary significantly based on parameter choices, though their direct applicability to non-convex neural network optimization remains uncertain. Despite these advances, several research gaps remain: (1) most studies focus on theoretical properties rather than comprehensive benchmarking on standard tasks; (2) ADMM's sensitivity to hyperparameters in neural network contexts is understudied and (3) empirical evaluations of ADMM's scaling efficiency in distributed neural network training are limited. This thesis addresses these gaps by conducting a systematic comparative evaluation of ADMM against GD and SGD for neural network optimization, implementing a consistent framework for evaluation, analyzing hyperparameter sensitivity, and examining performance in both single-node and distributed settings. The findings aim to establish when and how ADMM may offer advantages over traditional gradient-based methods for neural network optimization.

3 Methodology

This chapter presents the methodological framework employed in this research to evaluate the performance of the Alternating Direction Method of Multipliers (ADMM) in standard neural networks. Building upon the theoretical foundations established in chapter 2, this thesis implements and compares multiple neural network optimization approaches, with a primary focus on ADMM. The comparative evaluation includes traditional gradient-based methods: Gradient Descent (GD) and Stochastic Gradient Descent (SGD) in order to provide comprehensive insights into the relative strengths and limitations of ADMM for neural network training.

The methodology is designed to systematically analyze performance across various dimensions, including convergence behavior, computational efficiency, and scalability in distributed environments.

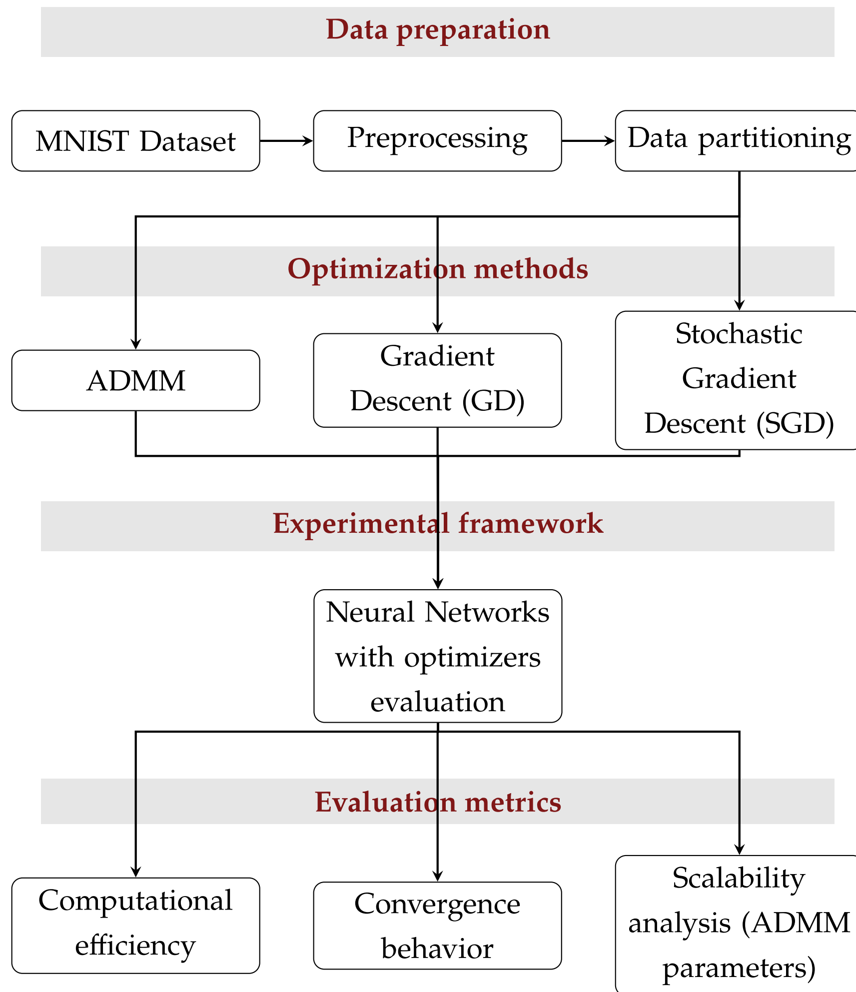


Figure 3.1: Methodology pipeline for evaluating ADMM performance in neural networks

3.1 Dataset

This section describes the dataset used for the experimental evaluation of optimization algorithms. We selected a standard benchmark dataset to ensure comparability with existing research and to provide a well-understood problem space for analyzing algorithmic performance.

3.1.1 MNIST dataset

We used the MNIST dataset [13], a widely recognized benchmark in machine learning. MNIST consists of handwritten digit images, providing an ideal testbed for evaluating neural network optimization methods due to its moderate

complexity and established performance baselines.

The MNIST dataset includes:

- 60,000 training images
- 10,000 test images
- 10 classes (digits 0-9)
- 28x28 grayscale images (784 pixels per image)

The MNIST dataset has established itself as a fundamental benchmark in machine learning and computer vision research. Its popularity stems from several key attributes: the task of digit recognition is conceptually straightforward, the dataset's compact size requires minimal computational resources, and researchers benefit from its well-documented structure and straightforward implementation [11]. These characteristics made MNIST particularly suitable for comparing different optimization approaches, as it provided sufficient complexity to be meaningful while remaining computationally tractable.

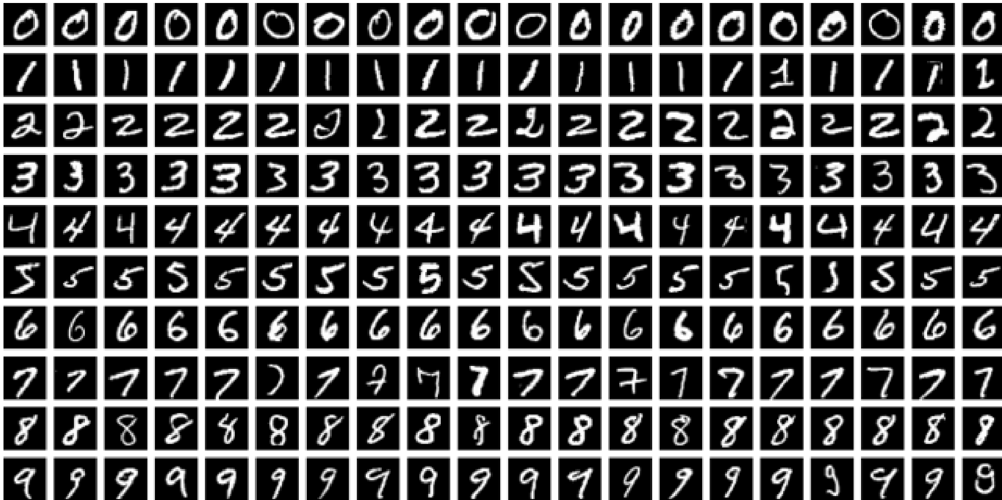


Figure 3.2: Sample images from the MNIST dataset showing handwritten digits from 0 to 9.

3.1.2 Data preparation

The dataset underwent several preprocessing steps to ensure optimal performance across all optimization methods. Images were normalized to have zero mean and unit variance using the standard values for MNIST (mean=0.1307, std=0.3081) and flattened into 784-dimensional vectors for input to the neural

network. For ADMM specifically, labels were one-hot encoded as required by the implementation, and data was transposed to match the expected format (features x samples).

For distributed ADMM experiments, the training data was partitioned across computing nodes. Each node received approximately $\frac{60{,}000}{N_{nodes}}$ training examples, with partitioning done sequentially without shuffling to ensure reproducibility. All nodes had access to the complete test set for evaluation. This partitioning scheme allowed for evaluating how ADMM performed in distributed settings where each node had access to only a subset of the training data.

3.2 Experimental framework

All experiments in this study used an equivalent neural network architecture to ensure fair comparison between optimization methods. The network consisted of:

- Input dimension: 784 neurons (flattened 28x28 MNIST images)
- Hidden layer: 256 neurons with ReLU activation
- Output dimension: 10 neurons (one per class for MNIST)

3.2.1 Network implementation

While the architecture was equivalent across methods, the implementation differed between ADMM and gradient-based approaches.

The ADMM implementation explicitly modeled the network with weight matrices and auxiliary variables:

- Weight matrices: $W_1 \in \mathbb{R}^{256\times784}$, $W_2 \in \mathbb{R}^{256\times256}$, $W_3 \in \mathbb{R}^{10\times256}$
- Auxiliary variables: z_1, a_1, z_2, a_2, z_3 represented layer outputs and activations
- Lagrangian multipliers: λ for enforcing constraints

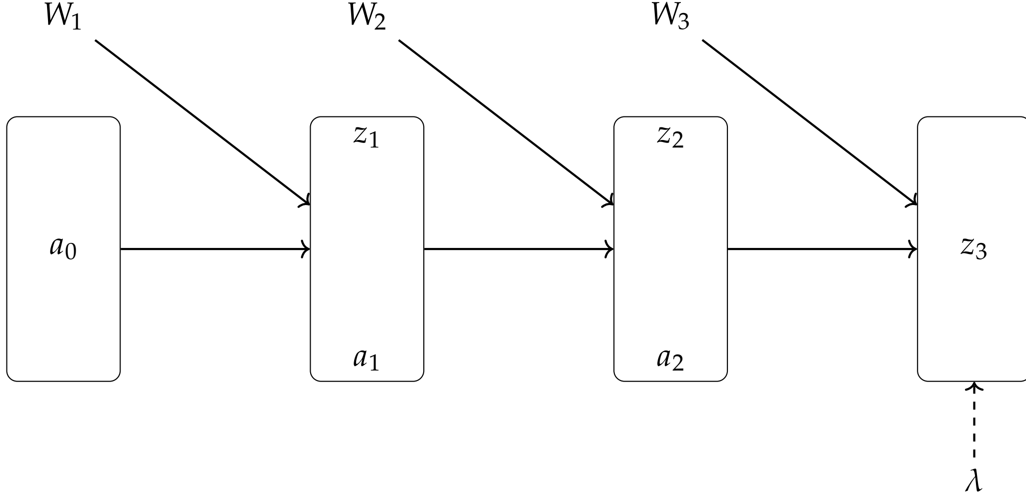


Figure 3.3: Neural network architecture with ADMM variables, showing the relationship between weight matrices (W_1, W_2, W_3) and auxiliary variables (z_1, a_1, z_2, a_2, z_3).

For gradient-based methods (GD and SGD), the network was implemented using PyTorch's standard neural network modules with the same overall structure.

3.3 Optimization methods

This section describes the three optimization approaches that were implemented and compared in this study. Building upon the theoretical foundations established in chapter 2, the implementation details and specific algorithmic considerations for each method are presented.

3.3.1 ADMM optimization

The ADMM approach reformulated the neural network training as a constrained optimization problem. For the implemented network, the formulation introduced auxiliary variables that decoupled the optimization across layers:

$$\text{Inputs:} \quad a_0 \in \mathbb{R}^{n_{inputs} \times batch_size} \tag{3.1}$$

$$\text{Layer 1:} \quad z_1 = W_1 a_0, \quad a_1 = \sigma(z_1) \tag{3.2}$$

$$\text{Layer 2:} \quad z_2 = W_2 a_1, \quad a_2 = \sigma(z_2) \tag{3.3}$$

$$\text{Output:} \quad z_3 = W_3 a_2 \tag{3.4}$$

Weight updates. The weight matrices were updated using a least-squares approach:

$$W_l = z_l a_{l-1}^{\dagger} \tag{3.5}$$

where $a_{l-1}^{\dagger}$ represented the pseudoinverse of a_{l-1}. This was implemented as:

```
def _weight_update(self, layer_output, activation_input):
    if dist.is_initialized():
        return self._distributed_weight_update(layer_output,
    activation_input)
    else:
        return torch.mm(layer_output.float(),
                        torch.pinverse(activation_input.float()))
```

Code 3.1: Weight update implementation

In the distributed setting, this update was implemented through the aggregation of local statistics:

$$S_1^{local} = z_l a_{l-1}^T \tag{3.6}$$

$$S_2^{local} = a_{l-1} a_{l-1}^T \tag{3.7}$$

```
def _distributed_weight_update(self, layer_output, activation_input):
    local_S1 = torch.mm(layer_output.float(), activation_input.float
    ().t())
    local_S2 = torch.mm(activation_input.float(), activation_input.
    float().t())
    if dist.is_initialized():
        dist.all_reduce(local_S1, op=dist.ReduceOp.SUM)
        dist.all_reduce(local_S2, op=dist.ReduceOp.SUM)
    weight_matrix = torch.mm(local_S1, torch.pinverse(local_S2))
    return weight_matrix
```

Code 3.2: Distributed weight update implementation

These statistics were aggregated across computing nodes, and the weight matrix was computed as:

$$W_l = S_1 S_2^{-1} \tag{3.8}$$

Activation updates. The post-activation variables were updated by solving:

$$a_l = \arg\min_a \beta |z_{l+1} - W_{l+1} a|_F^2 + \gamma |a - \sigma(z_l)|_F^2 \tag{3.9}$$

This had the closed-form solution:

$$a_l = (\beta W_{l+1}^T W_{l+1} + \gamma I)^{-1}(\beta W_{l+1}^T z_{l+1} + \gamma \sigma(z_l)) \tag{3.10}$$

where β and γ were hyperparameters that balanced the consistency between layers and the nonlinear activation constraint.

Pre-activation updates. The pre-activation variables were updated by solving:

$$z_l = \arg\min_z \beta |z - W_l a_{l-1}|_F^2 + \gamma |a_l - \sigma(z)|_F^2 \tag{3.11}$$

Output layer and Lagrangian updates. For the final layer, the update incorporated Lagrangian multipliers:

$$z_3 = \frac{y - \lambda + \beta W_3 a_2}{1 + \beta} \tag{3.12}$$

where y represented the target values and λ was the Lagrangian multiplier. The Lagrangian multiplier was then updated:

$$\lambda = \lambda + \beta(z_3 - W_3 a_2) \tag{3.13}$$

Training process. The ADMM training process consisted of two main phases. The warming phase initialized and stabilized the auxiliary variables before the main training, performing a specified number of ADMM iterations without evaluating model performance. This phase was critical for establishing a reasonable starting point, particularly with random initialization. The main training proceeded with the same update steps and evaluated model performance after each iteration, continuing until either the maximum number of epochs was reached or an early stopping criterion was triggered.

3.3.2 Gradient-based methods

Gradient Descent (GD). Mini-batch GD updated the model parameters using the average gradient computed over a batch of training examples:

$$\theta_{t+1} = \theta_t - \eta \frac{1}{|B|} \sum_{i \in B} \nabla_\theta L(f_\theta(x_i), y_i) \tag{3.14}$$

where θ represented the model parameters, η was the learning rate, B was the current batch, and L was the loss function. The implementation used PyTorch's automatic differentiation and optimizer framework with a batch size of 128 and a learning rate of 0.01.

Stochastic Gradient Descent (SGD). Pure SGD updated the model parameters using the gradient computed on individual training examples:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(f_\theta(x_i), y_i) \tag{3.15}$$

This approach was implemented with the same learning rate (0.01) as GD, offering potentially faster convergence in terms of data passes but with higher variance in updates.

3.4 Hyperparameter configuration

The optimization methods were configured with specific hyperparameters that influenced their behavior and performance.

3.4.1 ADMM hyperparameters

The ADMM implementation involved several key hyperparameters summarized in Table 3.1.

The values in Table 3.1 represented the default configuration, though the implementation supported a grid search across different combinations of β (1, 2, 5), γ (1, 5, 7), and warming epochs (0, 1, 2, 5, 10) to identify optimal settings.

3.4.2 Gradient-based hyperparameters

For the gradient-based methods, the hyperparameters were selected based on common practices in the literature, as shown in Table 3.2.

The epochs for gradient-based methods were selected to ensure a fair comparison with ADMM in terms of total data passes while accommodating the computational differences between the methods.

Hyperparameter	Value(s)	Description
β	1	Penalty parameter in the ADMM formulation, controlling the weight of consistency between auxiliary variables and constraints
γ	1	Balance parameter for the nonlinear activation constraints, determining the importance of matching the ReLU activation function
Warming epochs	2	Number of initial iterations dedicated to stabilizing the auxiliary variables before beginning the main training loop
Maximum epochs	50	Upper limit of training iterations before termination
Grace patience	5	Number of epochs with suboptimal performance before triggering early stopping

Table 3.1: ADMM hyperparameter configuration

Hyperparameter	GD	SGD	Description
Learning rate	0.01	0.01	Step size controlling how much to adjust weights based on the gradient
Loss function	CrossEntropy	CrossEntropy	Loss function for classification tasks

Table 3.2: Gradient-based methods hyperparameter configuration

3.5 Evaluation methodology

The evaluation of the optimization methods was based on several key metrics designed to provide a comprehensive comparison of their performance.

3.5.1 Performance metrics and experimental protocols

We employed multiple metrics to assess the optimization methods' effectiveness and efficiency, along with standardized protocols for each method to ensure fair comparisons.

Accuracy. Classification accuracy was calculated as the proportion of correctly predicted labels:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of examples}} \times 100\% \tag{3.16}$$

For ADMM, this was computed using argmax on the one-hot encoded predictions and labels, while for GD and SGD, it was computed directly using PyTorch's categorical accuracy calculation.

Loss. The loss function values were tracked throughout training to monitor convergence.

Training time. The total training time was measured in seconds, recording how long each method required to complete its training process. For ADMM, this included both the warming phase and the main training loop.

ADMM protocol. For the ADMM experiments, the procedure involved: initializing the model with the specified architecture and hyperparameters; partitioning training data across available nodes for distributed experiments; executing the warming phase; running the main training loop while recording metrics; applying early stopping if needed; restoring the best model; and evaluating on the test set for final performance metrics.

Gradient-based protocol. For GD and SGD experiments, the procedure involved: initializing the model with the PyTorch implementation; creating the appropriate data loader; training for the specified number of epochs while recording metrics; and evaluating the final model on the test set.

3.5.2 Implementation details

The experiments were conducted using PyTorch for tensor operations and neural network functionalities, with specific configurations for different computational environments.

Software stack. Our implementation utilized PyTorch for tensor operations and automatic differentiation, torch.distributed for multi-node communication,

and torchvision for dataset loading and preprocessing. For monitoring and logging, we employed Weights & Biases (wandb) for experiment tracking and visualization.

Hardware and distributed setup. The experiments were conducted on computing infrastructure with multi-core processors for parallel computation and sufficient RAM for the MNIST dataset and model parameters. For ADMM, distributed training was implemented using PyTorch's distributed communication framework, with process initialization using environment variables (RANK, WORLD_SIZE, LOCAL_RANK), communication backends (NCCL for GPU-based communication, Gloo for CPU-based communication), and distributed operations (All-reduce for aggregating statistics, Broadcast for synchronizing model parameters).

4

Results and discussion

This chapter presents the experimental results and analysis of the Alternating Direction Method of Multipliers (ADMM) algorithm in standard neural networks. The findings are analyzed to evaluate ADMM's practical utility for neural network optimization, highlighting both its strengths and limitations compared to traditional approaches.

4.1 Computational efficiency

The computational efficiency of optimization algorithms is a critical factor in their practical adoption for neural network training. This section compares the processing time requirements of ADMM against traditional optimization methods (SGD and GD), revealing significant differences in their computational demands.

4.1.1 Epoch processing time comparison and implications

Figure 4.1 presents the processing time per epoch for the three optimization algorithms: ADMM, SGD and GD. The horizontal axis represents the training epochs (0-35), while the vertical axis shows the time units (seconds) required to complete each epoch.

As shown in the figure, ADMM required the least processing time, averaging approximately 0.5-1 time units per epoch throughout the entire training process. GD demonstrated moderate efficiency with processing times of approximately

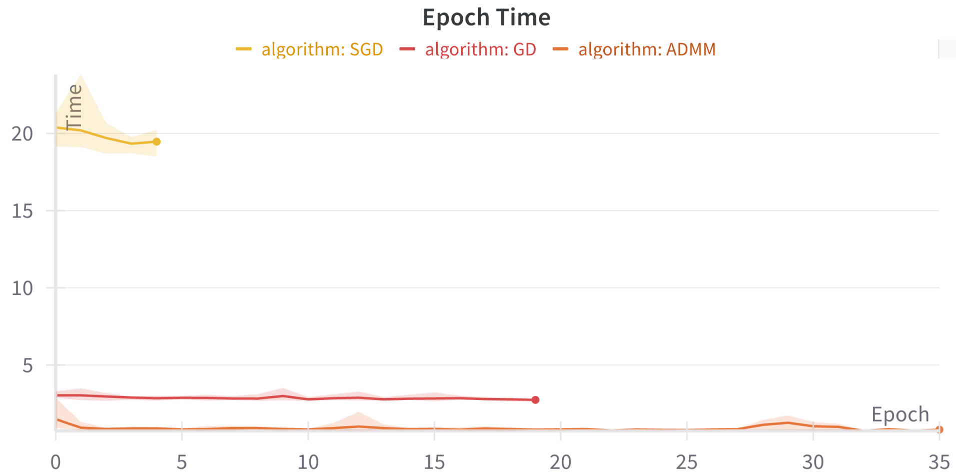


Figure 4.1: Epoch processing time comparison between ADMM, SGD, and GD optimization algorithms.

2.5-3 time units per epoch and was run for about 20 epochs. SGD showed significantly higher computational demands, with epoch times of approximately 20 time units, and was terminated after 5 epochs.

It is important to note that the SGD and GD experiments were intentionally stopped after sufficient data had been collected to establish the clear differences in computational requirements between the three algorithms. The experiments confirmed that ADMM offers substantially lower per-epoch processing time compared to the alternatives. ADMM was run for the full 35 epochs, maintaining consistent performance throughout the entire process with only minor fluctuations observed between epochs 25-30.

The significant difference in processing times between these optimization algorithms (with ADMM being approximately 3 times faster than GD and 20 times faster than SGD) was apparent from the early epochs, making extended runs of the more time-intensive algorithms unnecessary for this comparison.

Implications of computational efficiency findings

These results demonstrate ADMM's potential as a computationally efficient alternative for neural network optimization. The substantial reduction in processing time 3x faster than GD and 20x faster than SGD could translate to significant practical advantages in resource-constrained environments or applications requiring rapid model training.

The efficiency likely stems from ADMM's approach of decomposing the optimization problem into simpler subproblems that can be solved in closed form, without requiring the expensive gradient calculations that characterize traditional methods. This fundamental difference in approach represents a trade-off that practitioners should consider when selecting optimization algorithms for different applications.

ADMM's consistent performance across multiple epochs also suggests reliable scalability for longer training sessions, which is particularly valuable for complex models that require extended training periods. These efficiency gains position ADMM as a promising candidate for real-time applications, initial model exploration, or scenarios where computational resources are limited.

4.2 Convergence behavior

While computational efficiency addresses the resource demands of optimization algorithms, their convergence characteristics determine how effectively they navigate the loss landscape to produce high-quality models. This section analyzes the convergence trajectories of ADMM, SGD, and GD across multiple metrics to evaluate their practical utility for neural network training.

4.2.1 Loss convergence analysis and interpretation

Figure 4.2 presents the loss function values across training epochs for the three optimization algorithms. The horizontal axis represents the training epochs (0-14), while the vertical axis shows the loss value.

SGD (black line) showed the most rapid convergence, starting at a loss value of approximately 0.2 and quickly decreasing to around 0.05 by epoch 4, where the line ends. GD (red line) began with the highest initial loss value near 0.75, dropped significantly within the first two epochs to around 0.35, and then gradually declined, stabilizing around 0.15 by the end of the training period.

ADMM (blue line) showed a unique convergence pattern, beginning with a moderate loss value of about 0.75, then exhibiting a spike to approximately 0.8 at epoch 1 before rapidly decreasing to around 0.3 by epoch 2. It then continued a more gradual descent, experiencing some fluctuations particularly around epochs 11-13, before finishing at a value comparable to GD at around 0.15.

The shaded blue region surrounding the ADMM curve represents variance

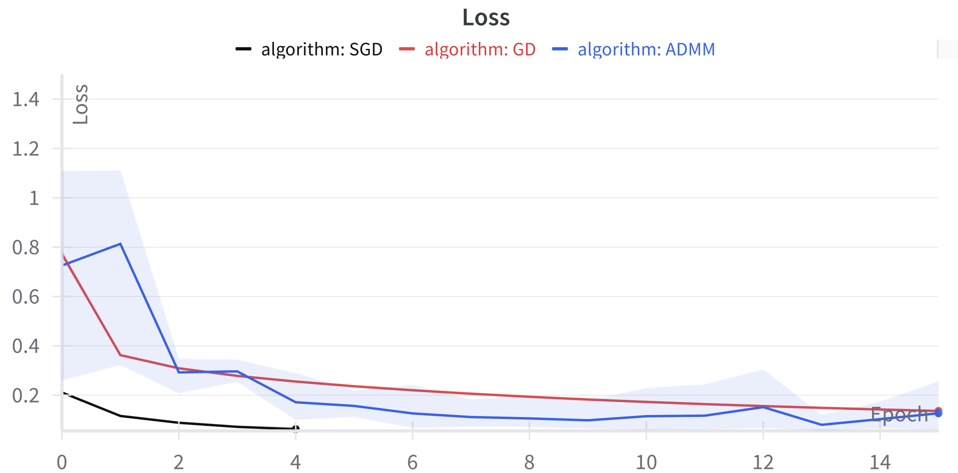


Figure 4.2: Loss function values per epoch for ADMM, SGD, and GD optimization algorithms.

across experimental runs, indicating that ADMM exhibits higher variability in performance compared to the other methods.

Discussion of loss convergence patterns

The distinct convergence patterns observed provide important insights into how these optimization approaches navigate the loss landscape. SGD's exceptionally rapid convergence demonstrates its effectiveness in quickly finding promising regions of the parameter space, achieving the lowest overall loss values before terminating at epoch 4.

GD shows a more gradual and steady convergence pattern, without the volatility seen in ADMM. Its consistent downward trajectory suggests reliable performance, though it doesn't achieve the same low loss values as SGD within the observed epochs.

ADMM's loss trajectory reveals both strengths and limitations of this approach. The initial spike followed by rapid decrease indicates that ADMM may temporarily move away from optimal solutions before converging more effectively. The fluctuations observed around epochs 11-13 suggest potential instability in ADMM's optimization process. This instability may be attributed to ADMM's decomposition approach, which can occasionally lead to subproblems that temporarily push the solution away from the optimal path before reconverging.

The wider variance observed in ADMM's performance (as shown by the blue shaded region) indicates less predictable behavior compared to gradient-based methods. This higher variability is an important consideration for applications where consistent, predictable convergence patterns are required. Despite these variations, ADMM's ability to ultimately achieve final loss values comparable to GD by epoch 14 demonstrates its capacity to find effective solutions, albeit through a different and potentially less stable path.

4.2.2 Accuracy performance and comparative analysis

Figure 4.3 presents the accuracy values across training epochs for the three optimization algorithms. The horizontal axis represents the training epochs (0-25), while the vertical axis shows the accuracy as a percentage.

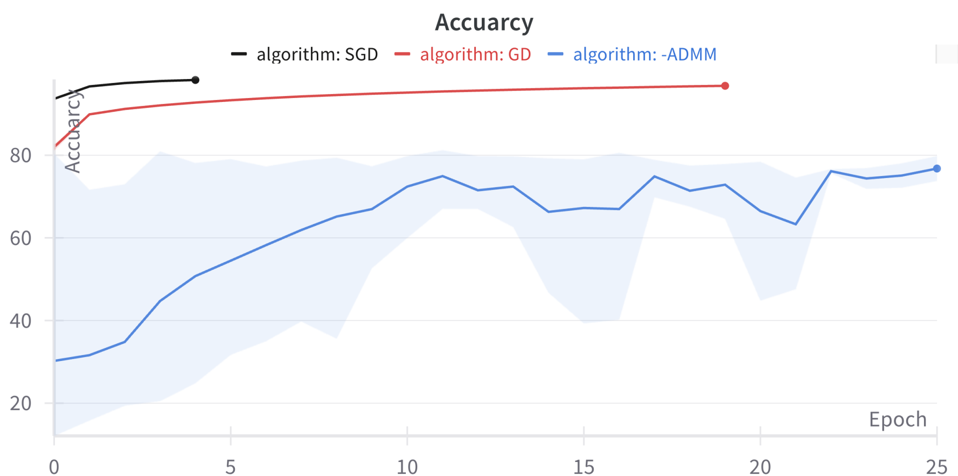


Figure 4.3: Accuracy values per epoch for ADMM, SGD, and GD optimization algorithms.

SGD achieved the highest overall accuracy, reaching approximately 90% within the first few epochs before plateauing. GD showed a steady increase in accuracy, starting at about 80% and gradually improving to roughly 95% by epoch 20.

ADMM showed a different trajectory, starting at around 30% accuracy and showing consistent improvement over the first 10 epochs. By epoch 10, ADMM reached approximately 70% accuracy and continued to fluctuate between 65% and 75% for the remainder of the training process. The final accuracy for ADMM at epoch 25 was approximately 75%.

The shaded region around the ADMM line indicates considerable variance in accuracy across experimental runs, suggesting less consistent performance compared to SGD and GD.

Implications for practical applications

The accuracy trajectories reveal a fundamental trade-off between computational efficiency and predictive performance. While ADMM demonstrates superior processing speed, its accuracy performance falls notably short of gradient-based methods on this standard classification task. This disparity highlights that ADMM, in its current implementation, may not be the optimal choice for applications where achieving state-of-the-art accuracy is the primary objective.

The 15-20% accuracy gap between ADMM and gradient-based methods is significant enough to go through a careful consideration when selecting optimization approaches. For applications where a 75-80% accuracy level is sufficient, ADMM's computational advantages may outweigh this performance gap. However, for tasks requiring higher accuracy, traditional methods like SGD and GD remain the preferred choice despite their computational costs.

ADMM's higher variance across runs also suggests that it may be less reliable in consistently producing high-quality models. This characteristic is particularly relevant for production environments where model performance stability is critical. The fluctuations observed in ADMM's accuracy during later training stages further indicate that it may benefit from additional stabilization mechanisms to maintain consistent performance throughout the training process.

4.2.3 Final test accuracy and optimal configurations

Figure 4.4 presents the final test accuracy for each optimization algorithm across different hyperparameter configurations. The horizontal axis represents the accuracy percentage, while each row displays the results for a specific algorithm configuration.

SGD and GD configurations remained consistent across all runs (5 epochs, learning rate of 0.01), producing stable accuracy results each time. For ADMM, three different parameter combinations were evaluated to determine the optimal configuration. The first configuration used 100 epochs with BETA=1, GAMMA=10, and WARMUP=10, while the second and third configurations

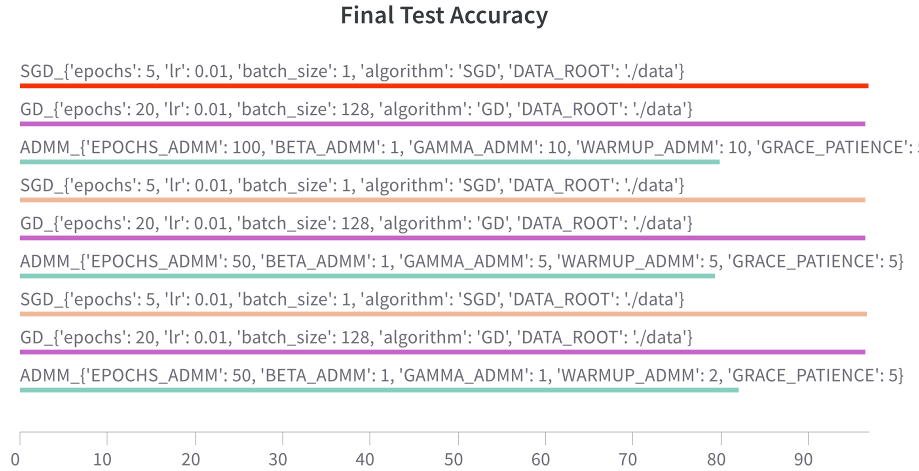


Figure 4.4: Final test accuracy for different configurations of ADMM, SGD, and GD optimization algorithms.

used 50 epochs with varying values for GAMMA and WARMUP. All ADMM configurations maintained BETA=1 and GRACE_PATIENCE=5 as constants.

To highlight the best performance achieved with each optimization algorithm, table 4.1 presents the optimal hyperparameter configurations and their corresponding accuracy values.

Algorithm	Epochs	Learning rate	Additional parameters	Accuracy (%)
SGD	5	0.01	-	**96.86**
GD	20	0.01	-	**96.51**
ADMM	50	-	BETA=1, GAMMA=1, WARMUP=2, GRACE=5	**81.98**

Table 4.1: Optimal hyperparameter configurations and corresponding test accuracy for each optimization algorithm

The test results showed that SGD achieved the highest final accuracy (96.86%) among the three algorithms, followed closely by GD (96.51%). ADMM achieved a lower accuracy of 81.98% with the optimal configuration of BETA=1, GAMMA=1, WARMUP=2, and GRACE=5.

Discussion of performance differences

The performance gap between ADMM and gradient-based methods reflects fundamental differences in how these algorithms approach optimization. The

nearly 15% difference in accuracy is substantial but should be interpreted within the context of the computational efficiency trade-off discussed earlier. ADMM's 82% accuracy, while lower than gradient-based methods, still represents effective learning on the MNIST dataset and may be sufficient for many applications where processing speed is prioritized over achieving the highest possible accuracy.

The optimal configuration findings indicate that ADMM is quite sensitive to hyperparameter selection. The identification of BETA=1, GAMMA=1, and WARMUP=2 as the optimal parameter combination provides practical guidance for practitioners implementing ADMM for neural network training. This sensitivity contrasts with gradient-based methods, which performed well with standard configurations, suggesting that ADMM requires more careful tuning to achieve its best performance.

The necessity of 50 epochs for ADMM to achieve its optimal accuracy, compared to just 5 epochs for SGD, further emphasizes the different convergence characteristics of these approaches. While ADMM requires more iterations, its per-epoch efficiency means that the total training time can still be competitive or even advantageous compared to gradient-based methods, depending on the specific application requirements.

4.3 Scalability analysis and parameter sensitivity

This section investigates the performance characteristics of ADMM under various configuration scenarios, focusing on its scalability in parallel computing environments and sensitivity to key hyperparameters. Understanding these relationships is crucial for optimizing ADMM implementations in practical neural network training applications and identifying the most efficient parameter settings for different computational constraints.

4.3.1 ADMM scalability with parallel processing

The scalability of ADMM with increasing numbers of parallel processes was examined to evaluate its potential for distributed computing environments. Figures 4.5, 4.6, 4.7, and 4.8 present the training and test accuracy of ADMM when distributed across different numbers of parallel processes.

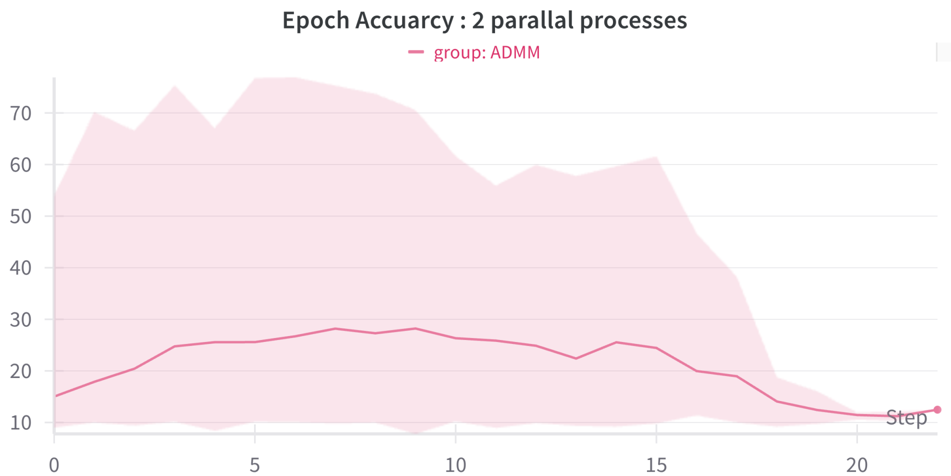


Figure 4.5: Training accuracy of ADMM with 2 parallel processes across epochs.

Figure 4.5 shows the training accuracy with 2 parallel processes. The accuracy initially increased to approximately 25-28% by epoch 7, then remained relatively stable before gradually declining after epoch 15 to about 12% by epoch 20. The shaded region indicates considerable variance across experimental runs.

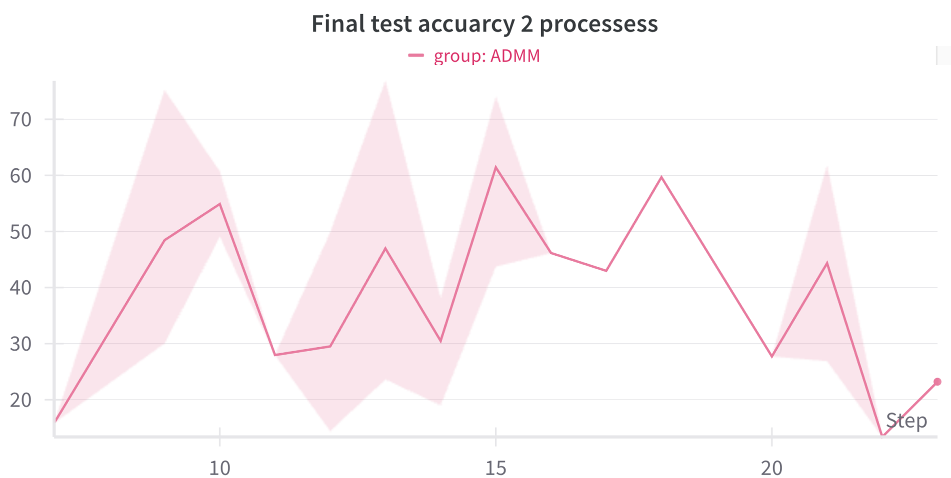


Figure 4.6: Test accuracy of ADMM with 2 parallel processes across steps.

Figure 4.6 displays the test accuracy with 2 parallel processes, which showed high variability, fluctuating between approximately 15% and 60% throughout the testing steps. The pattern exhibited multiple sharp drops and recoveries, indicating instability in the distributed optimization process.

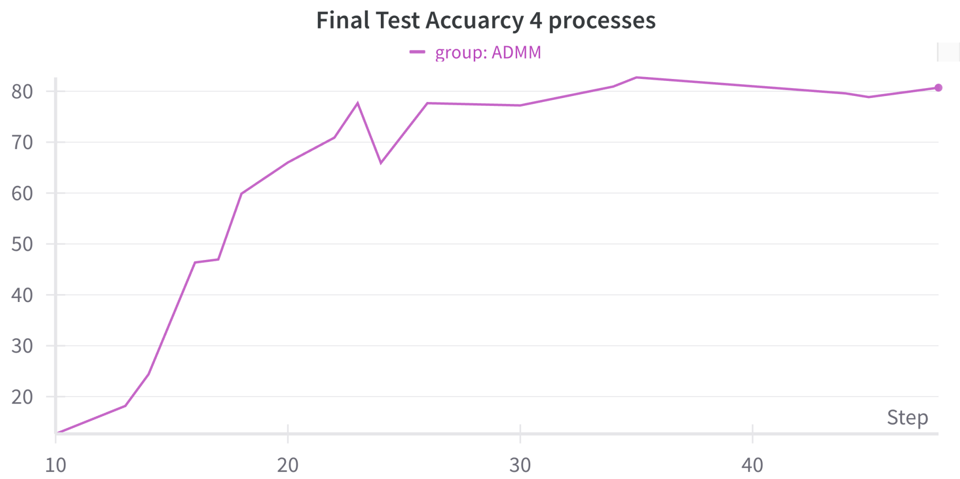


Figure 4.7: Test accuracy of ADMM with 4 parallel processes across steps.

When scaling to 4 parallel processes, as shown in Figure 4.7, the test accuracy showed a more stable upward trend. Starting at approximately 12% at step 10, the accuracy improved consistently, reaching over 80% by step 35, and maintained this level through the remainder of the testing period. The convergence pattern was smoother with fewer fluctuations compared to the 2-process implementation.

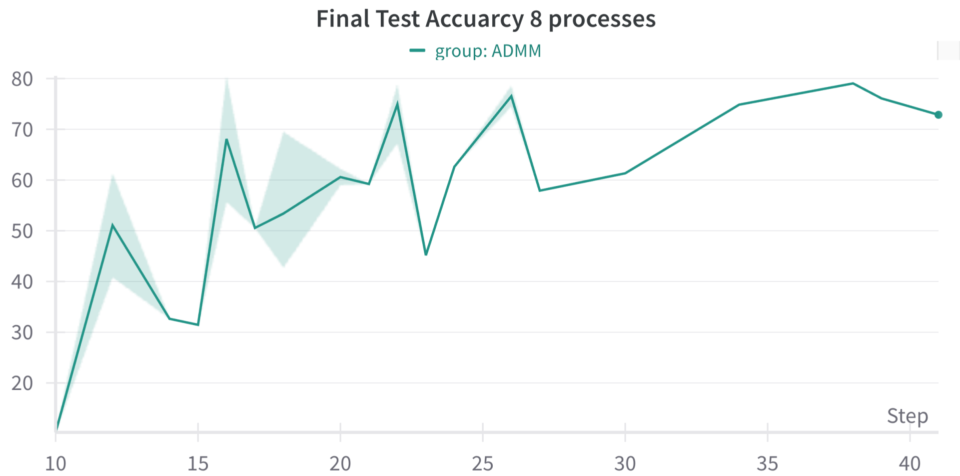


Figure 4.8: Test accuracy of ADMM with 8 parallel processes across steps.

Figure 4.8 presents the results with 8 parallel processes. The accuracy showed an initial rapid increase from 10% to approximately 50% by step 12, followed by several significant fluctuations. Despite these variations, the general trend was

positive, with accuracy improving to approximately 75-80% by the end of the testing period. Some instability remained, as evidenced by the drops at steps 16, 22, and 26.

These results demonstrate ADMM's promising but nuanced potential for distributed neural network training. The significant improvement in both stability and final accuracy when scaling from 2 to 4 processes suggests that ADMM can effectively leverage parallel computing resources, aligning with Taylor et al.'s [45] findings regarding ADMM's scaling properties.

The optimal performance observed with 4 parallel processes indicates that there exists a "sweet spot" in the parallelization strategy for ADMM. This finding is particularly valuable for practical implementations, as it suggests that modest parallel computing resources can be sufficient to achieve optimal performance with ADMM, potentially reducing infrastructure requirements compared to more resource-intensive distributed training approaches.

However, the diminishing returns and increased instability observed when scaling to 8 processes highlight important limitations in ADMM's scalability. This behavior contrasts with the theoretical linear scaling properties of ADMM and suggests that in practice, communication overhead or synchronization issues may impact performance at higher levels of parallelization. The fluctuations observed in the 8-process implementation indicate that additional stabilization mechanisms may be necessary to fully realize ADMM's potential in large-scale distributed environments.

These findings have significant implications for the design of distributed neural network training systems. Rather than maximizing the number of parallel processes, implementations may benefit from finding the optimal balance between parallelization and stability, potentially through adaptive approaches that adjust the degree of parallelism based on observed convergence behavior.

4.3.2 Parameter sensitivity analysis

After establishing optimal parallel processing configurations, we investigated how key ADMM parameters affect performance, convergence behavior, and resource utilization. This section analyzes four critical parameters: BETA, GAMMA, WARMUP, and initialization methods.

Effect of BETA parameter

The effect of the BETA parameter on ADMM's performance was investigated using the optimal 4-process configuration. Figure 4.9 and Figure 4.10 present the epoch processing time and final test accuracy for three different BETA values (1, 5, and 10).

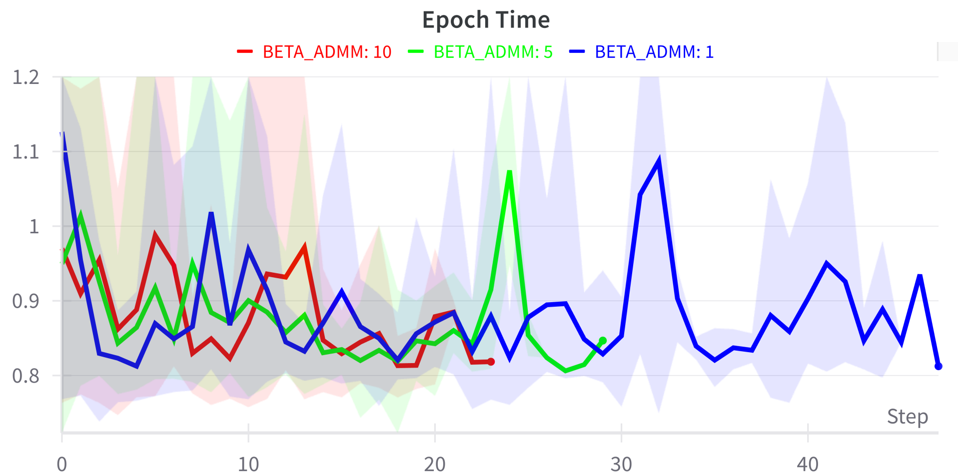


Figure 4.9: Epoch processing time for ADMM with different BETA values (1, 5, and 10) using 4 parallel processes.

Figure 4.9 shows that all three BETA configurations exhibited similar epoch processing times, averaging approximately 0.8-1.0 time units per epoch. Some variability was observed, particularly with BETA=1 showing a spike around epoch 30, but overall the processing time remained relatively stable regardless of the BETA value. The shaded regions indicate the variance across experimental runs.

The test accuracy results in Figure 4.10 reveal substantial differences between the three BETA configurations. BETA=1 (orange line) achieved the highest and most stable accuracy, maintaining approximately 80-82% throughout the testing period. BETA=5 (teal line) showed good performance initially, reaching nearly 80% accuracy around step 20, but gradually declined to about 72% by step 30. BETA=10 (purple line) demonstrated the lowest overall performance, with accuracy fluctuating between 45-60% and finishing at approximately 58% by the end of testing.

The BETA parameter's influence on accuracy without significantly affecting processing time reveals important insights into ADMM's optimization dynam-

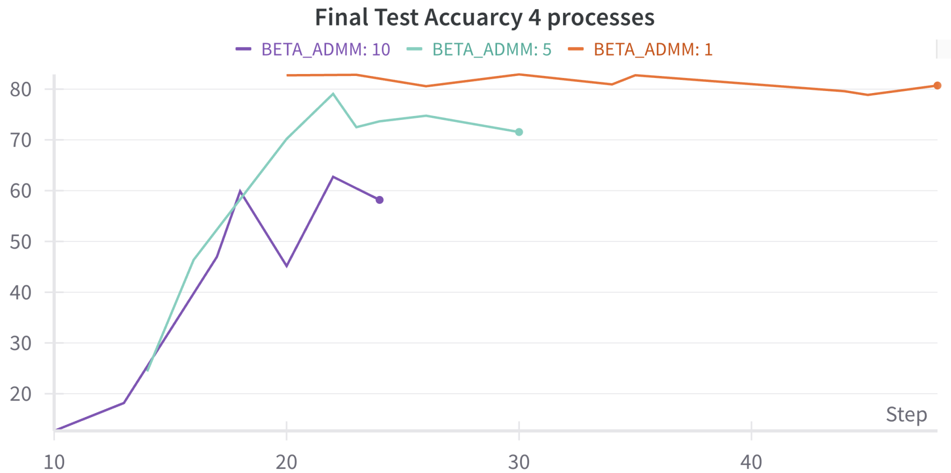


Figure 4.10: Test accuracy for ADMM with different BETA values (1, 5, and 10) using 4 parallel processes.

ics. BETA controls the penalty weight for the constraint between weight matrices and their corresponding auxiliary variables, essentially determining how strictly these constraints are enforced during optimization.

The superior performance with BETA=1 suggests that a balanced approach to constraint enforcement is optimal for neural network training with ADMM. This finding aligns with optimization theory, which indicates that overly strict constraint enforcement (higher BETA values) can restrict exploration of the parameter space, while too lenient enforcement may lead to solutions that violate the network's structural requirements.

The dramatic performance degradation observed with increasing BETA values (nearly 25% accuracy reduction from BETA=1 to BETA=10) highlights the critical importance of this parameter in practical implementations. Unlike many hyperparameters in machine learning that offer relatively flat performance curves around optimal values, ADMM's BETA parameter appears to have a sharp sensitivity profile that demands precise tuning.

Effect of GAMMA parameter

After establishing the optimal parallel processing configuration with 4 processes and BETA=1, the influence of the GAMMA parameter on ADMM's performance was examined. Three different GAMMA values (1, 5, and 10) were evaluated, with results presented in Figures 4.11, 4.12, and 4.13.

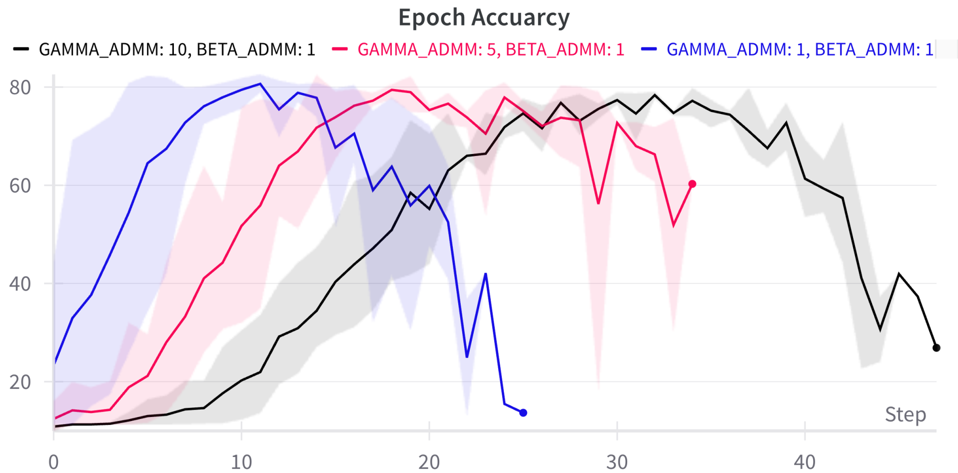


Figure 4.11: Training accuracy for ADMM with different GAMMA values (1, 5, and 10) using 4 parallel processes and BETA=1.

Figure 4.11 shows the training accuracy evolution over epochs for different GAMMA values. GAMMA=1 (purple line) exhibited the fastest initial convergence, reaching approximately 80% accuracy by epoch 10, but then experienced a sharp drop after epoch 20. GAMMA=5 (pink line) demonstrated a more moderate but steadier increase in accuracy, reaching its peak of about 80% by epoch 15. GAMMA=10 (black line) showed the slowest initial progress but eventually achieved the most stable performance after epoch 20, maintaining accuracy between 70-80% for the remainder of training.

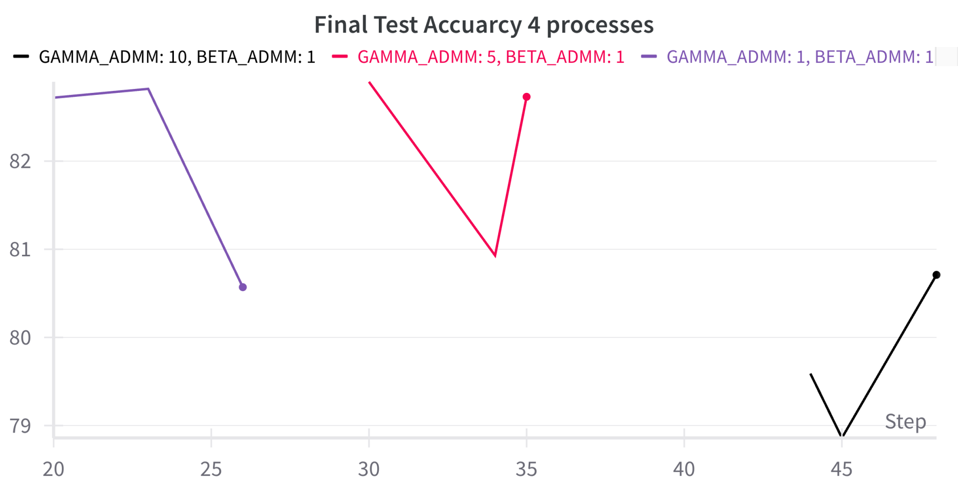


Figure 4.12: Final test accuracy for ADMM with different GAMMA values (1, 5, and 10) using 4 parallel processes and BETA=1.

Figure 4.12 presents a zoomed-in view of the final test accuracy during the later stages of training. All three configurations achieved high accuracy above 80%. GAMMA=1 showed the highest initial accuracy (approximately 82.5%) at step 20 but was only measured until step 25. GAMMA=5 reached a peak of approximately 82.5% at step 35. GAMMA=10 demonstrated more stable performance with slight fluctuations and achieved approximately 80.7% accuracy by the end of training.

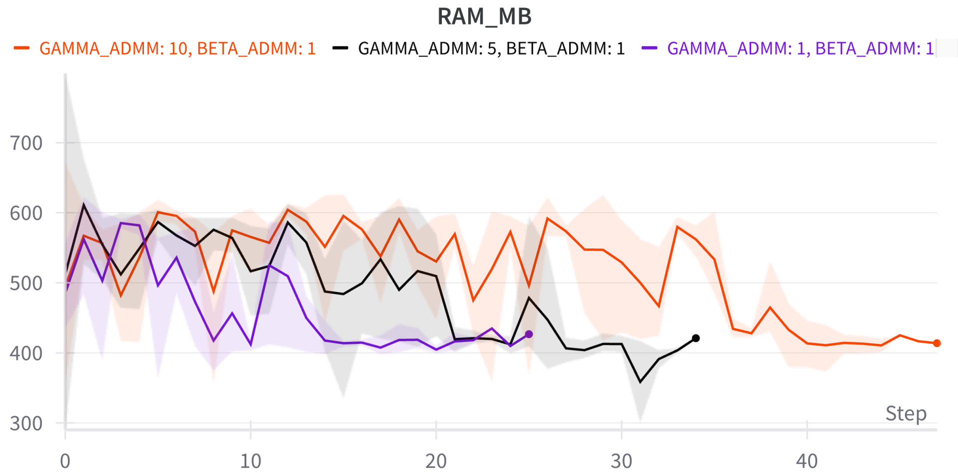


Figure 4.13: Memory usage (RAM in MB) for ADMM with different GAMMA values (1, 5, and 10) using 4 parallel processes and BETA=1.

Figure 4.13 illustrates the memory consumption for each GAMMA configuration. GAMMA=10 (orange line) consistently required the most memory, averaging around 500-600 MB throughout most of the training process before gradually decreasing to approximately 410 MB by the end of training. GAMMA=5 (black line) showed moderate memory usage that decreased from initial peaks of about 600 MB to around 420 MB in later stages. GAMMA=1 (purple line) demonstrated the most efficient memory utilization, stabilizing at approximately 410-430 MB after epoch 15.

The GAMMA parameter results reveal a fundamental trade-off between convergence speed, stability, and resource utilization in ADMM optimization. GAMMA controls the penalty weight for the nonlinear activation constraints, influencing how strictly the algorithm enforces the relationship between pre-activation and post-activation variables.

The faster initial convergence observed with GAMMA=1 suggests that a

lighter penalty on activation constraints enables more rapid exploration of the parameter space early in training. However, the subsequent performance drop indicates potential instability with this approach. Conversely, GAMMA=10 demonstrated slower initial progress but greater stability throughout the training process, suggesting that stricter enforcement of activation constraints leads to more reliable, though initially slower, convergence.

The memory usage patterns provide additional insight into this trade-off. Higher GAMMA values required consistently greater memory resources, indicating that stricter constraint enforcement demands more computational overhead. This observation complements the processing time findings and highlights that parameter choices in ADMM affect not only optimization quality but also resource utilization.

Effect of WARMUP parameter and initialization methods

The WARMUP parameter, which controls the number of epochs before ADMM's full optimization strategy is applied, was investigated alongside different initialization strategies. These aspects together determine the early-stage behavior of the ADMM optimization process.

WARMUP parameter analysis. Figure 4.14 presents the test accuracy for ADMM with three different WARMUP values (0, 2, and 5) using 4 parallel processes.

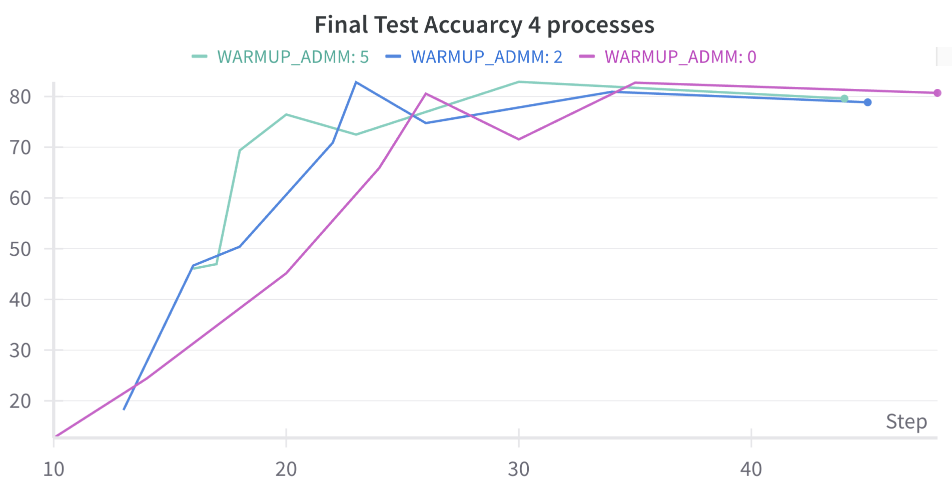


Figure 4.14: Test accuracy for ADMM with different WARMUP values (0, 2, and 5) using 4 parallel processes.

Figure 4.14 illustrates the impact of the WARMUP parameter on the convergence rate and final accuracy. WARMUP=5 (teal line) showed the fastest initial convergence, reaching approximately 70% accuracy by step 18, but experienced some fluctuations thereafter. WARMUP=2 (blue line) demonstrated a slightly slower initial convergence but achieved the highest intermediate accuracy of approximately 82% around step 22 before stabilizing around 80%. WARMUP=0 (purple line) exhibited the slowest initial convergence but eventually reached the highest final accuracy of approximately 80.5% by step 45.

The WARMUP parameter findings offer valuable insights into initialization strategies for ADMM optimization. This parameter's role in controlling when Lagrangian multiplier updates begin has subtle but significant effects on convergence dynamics.

Higher WARMUP values (such as 5) allow the optimization process to establish reasonable initial values for auxiliary variables before enforcing constraints exactly through Lagrangian updates. This approach facilitates faster early convergence by enabling the algorithm to explore the parameter space more freely in initial epochs. However, the subsequent fluctuations observed with WARMUP=5 suggest that this early exploration may lead to less stable trajectories once exact constraint enforcement begins.

Conversely, lower WARMUP values (such as 0) enforce constraints from the beginning, leading to slower but potentially more stable convergence paths. The highest final accuracy achieved with WARMUP=0 indicates that early constraint enforcement may ultimately lead to better solutions, albeit requiring more steps to reach them.

Initialization methods analysis. The effect of weight initialization strategies on ADMM's performance was evaluated across training accuracy and final test accuracy. Four initialization methods were compared: Normalized, He, Xavier, and Zero initialization.

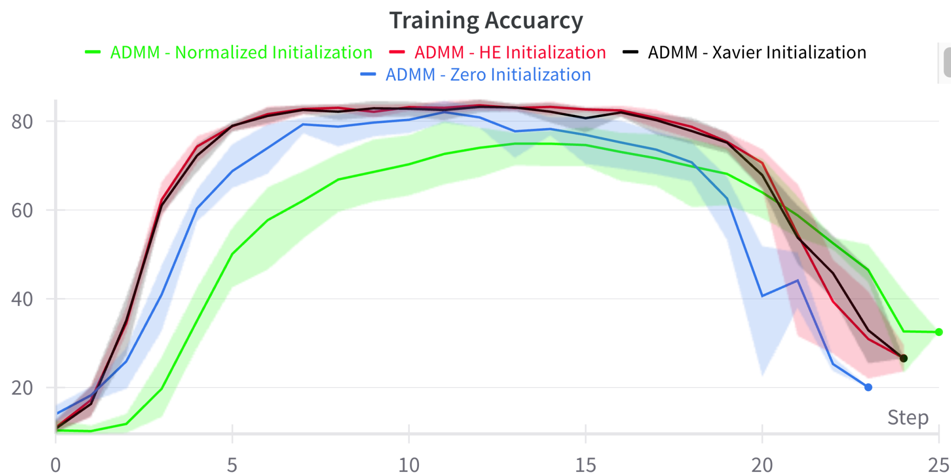


Figure 4.15: Training accuracy for ADMM with different initialization methods: Normalized, He, Xavier, and Zero.

Figure 4.15 reveals distinct patterns in how each initialization method affected accuracy development during training. He initialization (red line) and Xavier initialization (black line) demonstrated nearly identical performance, with the fastest initial convergence, reaching approximately 80% accuracy by step 5 and maintaining this level until around step 18 before gradually declining to about 28% by the end of training.

Zero initialization (blue line) showed moderately fast initial convergence, reaching approximately 80% accuracy by step 11, maintaining this level until step 16, then experiencing a sharp decline to approximately 20% by step 25.

Normalized initialization (green line) exhibited the slowest initial convergence, requiring approximately 12 steps to reach 70% accuracy and never exceeding 75%. However, it demonstrated the most gradual decline in later stages and maintained the highest final accuracy of approximately 32% by step 25.

Figure 4.16 presents the final test accuracy achieved by ADMM with each initialization method. While all methods achieved high accuracy values around 80%, there are subtle but noteworthy differences. He initialization (red bar) and Xavier initialization (black bar) appear to achieve marginally higher test accuracy compared to Normalized initialization (green bar) and Zero initialization (blue bar).

Combined implications for early-stage optimization. Both WARMUP parameter settings and initialization methods significantly impact ADMM's early op-

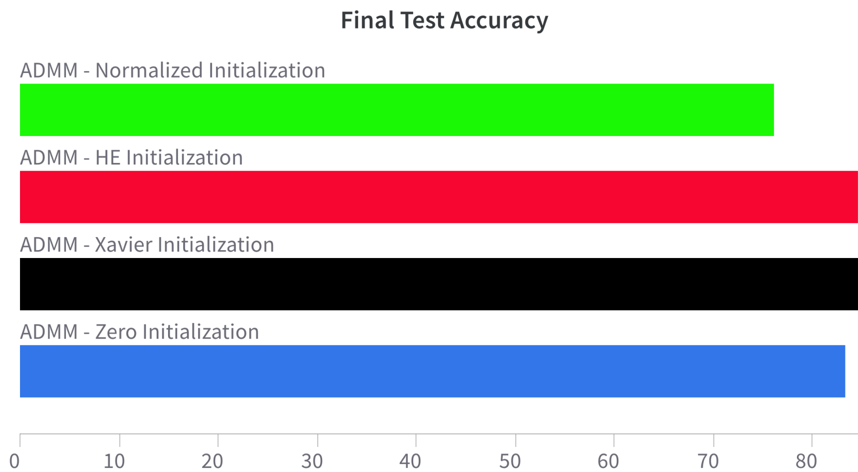


Figure 4.16: Final test accuracy for ADMM with different initialization methods.

timization behavior. The superior performance of He and Xavier initialization methods aligns with theoretical principles from deep learning literature, where these approaches were designed to maintain appropriate variance scales across network layers, facilitating efficient gradient flow.

The nearly identical performance of He and Xavier initialization suggests that ADMM benefits from initialization strategies that place parameters in favorable regions of the parameter space, similar to gradient-based methods. This finding indicates that despite ADMM's fundamentally different optimization approach, the initial positioning within the parameter space remains crucial for effective convergence.

When combined with appropriate WARMUP settings, particularly the balanced approach of WARMUP=2, these initialization strategies can significantly enhance ADMM's efficiency. The intermediate WARMUP value allows some initial flexibility while preventing the algorithm from moving too far from constraint-satisfying regions of the parameter space, complementing the beneficial starting positions provided by proper initialization methods.

These observations have practical implications for ADMM implementation strategies. For applications requiring rapid preliminary results, higher WARMUP values combined with He or Xavier initialization may be advantageous, while those prioritizing solution quality and stability would benefit from lower WARMUP values. The similar final accuracy levels achieved by most configurations indicates that these parameters primarily affect convergence dynamics rather than

ultimate performance, providing flexibility to adapt these parameters to specific application requirements.

5 Conclusions and future work

This chapter synthesises the findings from our investigation into the Alternating Direction Method of Multipliers (ADMM) for neural network optimization. We examine the implications of our results, consider the limitations of our approach, and propose directions for future research that could extend this work.

5.1 Summary of key findings

Our systematic evaluation of ADMM compared to traditional gradient-based methods has revealed several significant insights:

- **Computational efficiency:** ADMM demonstrated remarkable efficiency, processing epochs approximately 3 times faster than Gradient Descent (GD) and 20 times faster than Stochastic Gradient Descent (SGD). This efficiency derives from ADMM's fundamental approach of decomposing complex optimization problems into simpler subproblems that can be solved in closed form without requiring expensive gradient calculations. The consistent performance observed across multiple epochs indicates reliable scalability for extended training regimes.

- **Accuracy performance:** Despite its computational advantages, ADMM achieved a maximum accuracy of approximately 82% compared to the 96% attained by gradient-based methods. This 15% accuracy differential represents the primary trade-off when considering ADMM as an optimization alternative. The performance gap is substantial but not prohibitive for applications where computational efficiency may take precedence over maximum accuracy.

- **Convergence behaviour:** ADMM ultimately reached loss values comparable to gradient-based methods but exhibited less stability during the convergence process. The algorithm demonstrated occasional spikes in the loss function and wider variance across experimental runs. This behaviour reveals both strengths and limitations of ADMM's optimization dynamics and suggests that its decomposition approach, while efficient, may occasionally produce subproblems that temporarily divert the solution before reconverging.

- **Parallel processing capabilities:** Performance improved substantially when scaling from 2 to 4 parallel processes but showed diminishing returns and increased instability with 8 processes. This finding indicates that moderate parallelisation represents an optimal strategy for ADMM implementations. The identification of 4 processes as an optimal configuration provides valuable guidance for practical implementations and suggests that modest parallel computing resources can effectively leverage ADMM's distributed computing potential.

- **Parameter sensitivity:** ADMM's performance was significantly influenced by several key parameters:

  - BETA=1 consistently outperformed higher values, suggesting that a balanced approach to constraint enforcement is optimal for neural network training with ADMM
  - Lower GAMMA values facilitated faster initial convergence but potentially reduced stability, presenting a trade-off between convergence speed and training reliability
  - The WARMUP parameter substantially affected convergence dynamics, with higher values accelerating initial progress but sometimes leading to fluctuations in later training stages
  - He and Xavier initialisation methods demonstrated clear advantages over simpler approaches, indicating that ADMM benefits from established techniques that maintain appropriate variance scales across network layers

5.2 Limitations

While our investigation provides valuable insights into ADMM's potential for neural network optimization, several limitations of the current study should be acknowledged:

- The experiments focused primarily on standard neural network architectures applied to the MNIST dataset. This scope, while appropriate

for initial evaluation, leaves open questions about ADMM's performance on more complex architectures (such as deep convolutional networks or transformers) and larger, more diverse datasets where the computational patterns observed might not scale linearly.

- Our hyperparameter exploration, while informative, covered only discrete values within constrained ranges. A more exhaustive grid search or advanced hyperparameter optimization techniques might uncover parameter combinations that yield better performance than those identified in our study.
- The ADMM implementation represents one specific approach to adapting this algorithm for neural networks. Alternative formulations of the constraints, different update rules, or specialized implementations for specific network architectures might yield different or improved performance profiles.
- Theoretical guarantees for ADMM's convergence in non-convex neural network optimization remain less developed than for gradient-based methods. The occasional instability observed in our experiments highlights this theoretical gap and suggests that further mathematical analysis is needed to provide stronger convergence assurances.
- Our comparative analysis focused primarily on standard SGD and GD implementations. Evaluations against more sophisticated approaches (such as Adam, RMSProp, or more recent methods) would provide a more comprehensive positioning of ADMM within the broader optimization landscape.

5.3 Practical implications

Based on our empirical findings, we offer the following recommendations for practitioners considering ADMM for neural network optimization:

- Consider ADMM for applications where processing speed and computational efficiency are prioritized over achieving the highest possible accuracy. The substantial reduction in processing time could translate to significant practical advantages in resource-constrained environments or applications requiring rapid model training.
- For optimal results with ADMM, initialize with BETA=1, GAMMA values between 1-5, WARMUP=2, and either He or Xavier initialization methods. This configuration was found to balance convergence speed, stability, and final accuracy performance.
- When implementing distributed ADMM, aim for 4 parallel processes as an initial configuration. This "sweet spot" in parallelization strategy suggests that modest parallel computing resources can be sufficient to achieve optimal performance, potentially reducing infrastructure requirements.

- Carefully evaluate whether the approximately 15% accuracy gap compared to gradient-based methods is acceptable for specific application requirements. For many practical use cases, the substantially reduced computational demands may justify this trade-off.

- Consider hybrid strategies that leverage ADMM's efficiency for initial training phases before switching to gradient-based methods for fine-tuning when accuracy is critical. Such approaches could potentially combine the advantages of both optimization paradigms.

5.4 Future work

Our investigation reveals several promising avenues for future research that could address current limitations and expand ADMM's utility for neural network optimization:

- **Stability enhancement:** Develop specialized techniques to address the convergence instabilities observed with ADMM. Potential approaches include adaptive parameter adjustment mechanisms, momentum-inspired modifications, or constraint relaxation strategies during volatile convergence phases. These enhancements could improve ADMM's reliability while maintaining its computational advantages.

- **Architecture-specific adaptations:** Explore tailored ADMM formulations for specific neural network architectures such as CNNs, RNNs, or Transformers. Research into how network structure can inform constraint formulation and decomposition strategies may uncover more effective optimization approaches that leverage architectural characteristics.

- **Advanced distributed implementations:** Investigate techniques such as asynchronous updates, adaptive consensus constraints, or hierarchical decomposition strategies to enable better scaling to larger numbers of parallel processes. These approaches could potentially overcome the diminishing returns observed in our study when scaling beyond 4 processes.

- **Hybrid optimization approaches:** Develop systematic frameworks that combine ADMM's computational efficiency with the accuracy advantages of gradient-based methods. Strategies could include using ADMM for initial large-step exploration of the parameter space before switching to gradient-based fine-tuning, or specialized combinations of both approaches for different parts of the network.

- **Theoretical foundations:** Advance the mathematical understanding of ADMM's convergence properties in non-convex neural network optimization. Bridging the gap between ADMM's strong theoretical guarantees in convex settings and its empirical performance in neural networks could

yield insights that inform improved implementations and provide more robust performance guarantees.

In conclusion, this thesis has demonstrated that ADMM represents a viable alternative to gradient-based optimization for neural networks, offering distinct advantages in computational efficiency despite current limitations in accuracy performance. The algorithm's ability to achieve 82% accuracy with significantly reduced processing time positions it as a valuable addition to the neural network optimization toolkit, particularly for applications where computational constraints are significant. The parameter sensitivity and parallelization findings provide practical guidance for implementations, while the identified limitations and future research directions outline a path toward addressing current challenges.

As these challenges are addressed through further research, ADMM's role in neural network optimization is likely to expand, potentially narrowing the gap between computational efficiency and accuracy performance. This work contributes to the broader endeavour of developing diverse optimization approaches that can address the varying requirements of modern machine learning applications, particularly in resource-constrained environments where the trade-off between accuracy and computational efficiency must be carefully balanced.

References

[1] Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. "A Convergence Theory for Deep Learning via Over-Parameterization". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, Sept. 2019, pp. 242–252. URL: https://proceedings.mlr.press/v97/allen-zhu19a.html.

[2] Rayna Andreeva et al. *Topological Generalization Bounds for Discrete-Time Stochastic Optimization Algorithms*. 2024. arXiv: 2407.08723 [cs.LG]. URL: https://arxiv.org/abs/2407.08723.

[3] Sanjeev Arora et al. *Fine-Grained Analysis of Optimization and Generalization for Overparameterized Two-Layer Neural Networks*. 2019. arXiv: 1901.08584 [cs.LG]. URL: https://arxiv.org/abs/1901.08584.

[4] Tal Ben-Nun and Torsten Hoefler. *Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis*. 2018. arXiv: 1802.09941 [cs.LG]. URL: https://arxiv.org/abs/1802.09941.

[5] Y. Bengio, P. Simard, and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166. DOI: 10.1109/72.279181.

[6] Léon Bottou. "Stochastic Gradient Descent Tricks". In: *Neural Networks*. 2012. URL: https://api.semanticscholar.org/CorpusID:121049.

[7] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[8] Stephen Boyd et al. "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers". In: *Foundations and Trends in Machine Learning* 3 (Jan. 2011), pp. 1–122. DOI: 10.1561/2200000016.

[9] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165.

[10] Feiyang Chen et al. *Comprehensive Survey of Model Compression and Speed up for Vision Transformers*. 2024. arXiv: 2404.10407 [cs.CV]. URL: https://arxiv.org/abs/2404.10407.

[11] Gregory Cohen et al. "EMNIST: Extending MNIST to handwritten letters". In: *2017 International Joint Conference on Neural Networks (IJCNN)*. 2017, pp. 2921–2926. DOI: 10.1109/IJCNN.2017.7966217.

[12] Yann Dauphin et al. *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization*. 2014. arXiv: 1406.2572 [cs.LG]. URL: https://arxiv.org/abs/1406.2572.

[13] Li Deng. "The mnist database of handwritten digit images for machine learning research". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[14] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Ed. by Jill Burstein, Christy Doran, and Thamar Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://aclanthology.org/N19-1423/.

[15] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12.61 (2011), pp. 2121–2159. URL: http://jmlr.org/papers/v12/duchi11a.html.

[16] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12 (July 2011), pp. 2121–2159.

[17] Euhanna Ghadimi et al. "Optimal Parameter Selection for the Alternating Direction Method of Multipliers (ADMM): Quadratic Problems". In: *IEEE Transactions on Automatic Control* 60.3 (2015), pp. 644–658. DOI: 10.1109/TAC.2014.2354892.

[18] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[20] Robert M. Gower et al. *Variance-Reduced Methods for Machine Learning*. 2020. arXiv: 2010.00892 [cs.LG]. URL: https://arxiv.org/abs/2010.00892.

[21] Lei Guan et al. "An Efficient ADMM-Based Algorithm to Nonconvex Penalized Support Vector Machines". In: *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*. 2018, pp. 1209–1216. DOI: 10.1109/ICDMW.2018.00173.

[22] Saad Hikmat Haji and Adnan Mohsin Abdulazeez. "Comparison of Optimization Techniques Based on Gradient Descent Algorithm: A Review". In: *PalArch's Journal of Archaeology of Egypt/Egyptology* 18.4 (2021), pp. 2715–2743. URL: https://archives.palarch.nl/index.php/jae/article/view/6705.

[23] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[24] Kaiming He et al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. arXiv: 1502.01852 [cs.CV]. URL: https://arxiv.org/abs/1502.01852.

[25] Sepp Hochreiter. "Untersuchungen zu dynamischen neuronalen Netzen". PhD thesis. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München, 1991.

[26] Yanping Huang et al. *GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism*. 2019. arXiv: 1811.06965 [cs.CV]. URL: https://arxiv.org/abs/1811.06965.

[27] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG]. URL: https://arxiv.org/abs/1502.03167.

[28] Stanisaw Jastrzbski et al. *Three Factors Influencing Minima in SGD*. 2018. arXiv: 1711.04623 [cs.LG]. URL: https://arxiv.org/abs/1711.04623.

[29] Nitish Shirish Keskar et al. *On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima*. 2017. arXiv: 1609.04836 [cs.LG]. URL: https://arxiv.org/abs/1609.04836.

[30] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Commun. ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386. URL: https://doi.org/10.1145/3065386.

[32] Yunwen Lei. "Stability and Generalization of Stochastic Optimization with Nonconvex and Nonsmooth Problems". In: *Proceedings of Thirty Sixth Conference on Learning Theory*. Ed. by Gergely Neu and Lorenzo Rosasco. Vol. 195. Proceedings of Machine Learning Research. PMLR, Dec. 2023, pp. 191–227. URL: https://proceedings.mlr.press/v195/lei23a.html.

[33] Hao Li et al. *Visualizing the Loss Landscape of Neural Nets*. 2018. arXiv: 1712.09913 [cs.LG]. URL: https://arxiv.org/abs/1712.09913.

[34] Yujun Lin et al. *Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training*. 2020. arXiv: 1712.01887 [cs.CV]. URL: https://arxiv.org/abs/1712.01887.

[35] Ruben Mayer and Hans-Arno Jacobsen. *Scalable Deep Learning on Distributed Infrastructures: Challenges, Techniques and Tools*. 2019. arXiv: 1903.11314 [cs.DC]. URL: https://arxiv.org/abs/1903.11314.

[36] Neal Parikh and Stephen Boyd. "Proximal Algorithms". In: *Foundations and Trendső in Optimization* 1.3 (2014), pp. 127–239. DOI: 10.1561/2400000003. URL: http://dx.doi.org/10.1561/2400000003.

[37] Guan-Ju Peng. "Adaptive ADMM for Dictionary Learning in Convolutional Sparse Representation". In: *IEEE Transactions on Image Processing* 28.7 (2019), pp. 3408–3422. DOI: 10.1109/TIP.2019.2896541.

[38] Xin Qian and Diego Klabjan. *The Impact of the Mini-batch Size on the Variance of Gradients in Stochastic Gradient Descent*. 2020. arXiv: 2004.13146 [math.OC]. URL: https://arxiv.org/abs/2004.13146.

[39] Samyam Rajbhandari et al. *ZeRO: Memory Optimizations Toward Training Trillion Parameter Models*. 2020. arXiv: 1910.02054 [cs.LG]. URL: https://arxiv.org/abs/1910.02054.

[40] Herbert Robbins and Sutton Monro. "A Stochastic Approximation Method". In: *Annals of Mathematical Statistics* 22.3 (Sept. 1951), pp. 400–407. DOI: 10.1214/aoms/1177729586. URL: https://doi.org/10.1214/aoms/1177729586.

[41] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2017. arXiv: 1609.04747 [cs.LG]. URL: https://arxiv.org/abs/1609.04747.

[42] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550.7676 (Oct. 2017), pp. 354–359. DOI: 10.1038/nature24270.

[43] Leslie N. Smith. *A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay*. 2018. arXiv: 1803.09820 [cs.LG]. URL: https://arxiv.org/abs/1803.09820.

[44] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, 17–19 Jun 2013, pp. 1139–1147. URL: https://proceedings.mlr.press/v28/sutskever13.html.

[45] Gavin Taylor et al. *Training Neural Networks Without Gradients: A Scalable ADMM Approach*. 2016. arXiv: 1605.02026 [cs.LG]. URL: https://arxiv.org/abs/1605.02026.

[46] André Teixeira et al. "The ADMM Algorithm for Distributed Quadratic Problems: Parameter Selection and Constraint Preconditioning". In: *IEEE Transactions on Signal Processing* 64.2 (2016), pp. 290–305. DOI: 10.1109/TSP.2015.2480041.

[47] Neil C. Thompson et al. *The Computational Limits of Deep Learning*. 2022. arXiv: 2007.05558 [cs.LG]. URL: https://arxiv.org/abs/2007.05558.

[48] Bo Wahlberg et al. *An ADMM Algorithm for a Class of Total Variation Regularized Estimation Problems*. 2012. arXiv: 1203.1828 [stat.ML]. URL: https://arxiv.org/abs/1203.1828.

[49] Ashia C. Wilson et al. *The Marginal Value of Adaptive Gradient Methods in Machine Learning*. 2018. arXiv: 1705.08292 [stat.ML]. URL: https://arxiv.org/abs/1705.08292.

[50] Chen Xing et al. *A Walk with SGD*. 2018. arXiv: 1802.08770 [stat.ML]. URL: https://arxiv.org/abs/1802.08770.

[51] Jinshan Zeng et al. "On ADMM in Deep Learning: Convergence and Saturation-Avoidance". In: *Journal of Machine Learning Research* 22.199 (2021), pp. 1–67. URL: http://jmlr.org/papers/v22/20-1006.html.

[52] Chiyuan Zhang et al. *Understanding deep learning requires rethinking generalization*. 2017. arXiv: 1611.03530 [cs.LG]. URL: https://arxiv.org/abs/1611.03530.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor, professor Jacopo, for his guidance, support, and expertise throughout this journey.

To my family, my brother, my mother, and my father, thank you for your love, support, and belief in me. I could not have done this without you.

To all my friends, thank you for your support and good humour over the years. A special thanks to Nazo for your help during this experience. I'm also really grateful to the two new great friends I made during this master's - Zhaku and Andrea. Your support and notes meant a lot, even though we didn't get much time together in person.

A huge thank you to my former employer and mentor, Eduart, for encouraging me to take this leap and pursue my studies abroad.

I'm also incredibly grateful to my current employers for their flexibility and understanding, which allowed me to balance my professional and academic life.

To my sunflower, words cannot express how much your love and support mean to me. I could not have done this without you, and I would not want to do anything in life without you. And, by the way...more. Game over.

Finally, a note of recognition for my own journey in balancing this master's degree in a new country while maintaining full-time work.

This thesis stands as a testament to all who have contributed to my journey, directly and indirectly. Thank you.

Declaration

I hereby declare that all contents and analyses presented in this thesis are my original work. However, to enhance the clarity and quality of the text, I employed OpenAI's ChatGPT for text restyling and proofreading. This support was limited to improving the linguistic presentation, and it did not influence the originality or integrity of the technical and scientific content.