Politecnico di Torino

Politecnico di Torino

Master in Data Science and Engineering

A.a. 2025/2026

Graduation Session March 2026

Data-Guided Benchmarking of Optimization Strategies

Supervisors:

Paolo Garza

Daniele Rege Cambrin

Candidate:

Filippo Struffi

Abstract

In Deep Learning, the choice of optimization algorithm significantly influences training dynamics, convergence behavior, and computational efficiency. This thesis presents a systematic benchmark of widely adopted first-order optimization methods, evaluated across eight heterogeneous tasks spanning computer vision and natural language processing, under a unified experimental framework designed to reflect realistic low-budget training conditions.

The results reveal several findings that challenge prevailing assumptions. RMSProp emerges as the most consistently strong optimizer, suggesting it is a robust and underutilized alternative to Adam. The classical SGD-vision / adaptive-NLP dichotomy receives only partial support: SGD with Momentum outperformed all adaptive methods on a structured NLP task, while architecture and loss structure proved more reliable predictors of optimizer suitability than task domain alone. Simpler optimizers frequently matched more complex recent variants, and sharpness-aware methods offered genuine advantages only in high-variance regimes where their computational overhead was justified. A systematic decoupling between epoch-level convergence and wall-clock efficiency further highlights that per-iteration cost must factor into optimizer selection.

Overall, the results demonstrate that optimizer effectiveness is inherently context-dependent. The findings provide empirically grounded guidance for selecting optimization strategies across diverse deep learning applications.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In today's highly digitalized society, a wide range of everyday activities, such as professional work, caring for family members, grocery shopping, and household maintenance, are increasingly mediated by technological systems. Although many of the devices embedded in our daily environment operate through relatively simple mechanisms, it is becoming increasingly common for users to interact with applications characterized by a higher degree of computational complexity. Despite their ubiquity, the internal functioning of these systems is often overlooked or deliberately avoided by users, either due to a lack of interest or to the perceived difficulty of understanding them.

Before being deployed within domestic environments, and in some cases even after their introduction, such applications undergo a crucial preparatory stage aimed at ensuring optimal user satisfaction. This stage typically consists of a learning phase, during which the system acquires the ability to identify patterns, preferences, and trends from furnished data. The knowledge extracted during this process enables the application to deliver services that are efficient, personalized, and aligned with user expectations. A representative example is provided by the voice assistants that have become widespread in households over the past decade, such as Alexa and Google Assistant. These systems rely on automatic speech recognition and natural language understanding techniques to interpret vocal commands, infer contextual information, and generate appropriate responses, forming the technological basis of modern conversational agents [**1**].

Similarly, autonomous robotic devices, such as robotic vacuum cleaners, can benefit from learned knowledge to improve navigation and obstacle avoidance. In this context, a variety of computer vision techniques, including image segmentation, object detection, and image classification, may be employed to analyze and interpret the surrounding environment. Likewise, learning-based approaches such as

reinforcement learning can be utilized to support the planning and optimization of cleaning trajectories over time. Furthermore, users can interact with learning-based systems even during leisure activities, for instance when receiving personalized recommendations for television programs or multimedia content. These recommendations are typically generated by adaptive algorithms that exploit users' past interactions and preferences to suggest relevant content, as illustrated in Figure **1.1**.

While the technologies mentioned above, as well as others, will be mostly discussed in greater detail in the subsequent chapters, it is essential at this stage to provide a general overview of how a system can process raw data and transform it into meaningful information that is both useful and actionable for the end user.

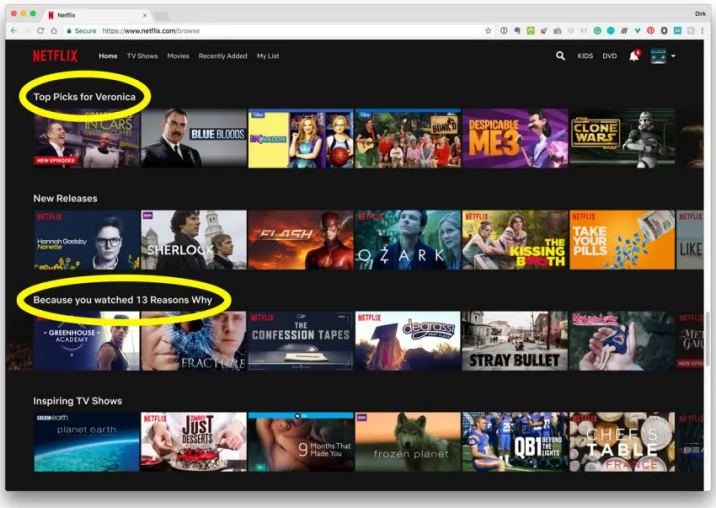

Figure 1.1: Netflix suggestion algorithm

This introductory chapter aims to provide the reader with a general and holistic overview of how learning occurs within a computational system, without reference to any specific learning paradigm, as for example supervised or unsupervised learning, nor to particular tasks or model architectures. A Deep Learning model operates by identifying and internalizing complex patterns from large-scale datasets.

The process begins with the collection and preparation of raw data, which may consist of images, textual information, or numerical values. This preliminary phase is essential to ensure that the inputs are clean, consistent, and suitable for subsequent analysis. Once pre-processed, the data is forwarded through a model composed of multiple layers, each responsible for progressively extracting features of increasing abstraction and semantic relevance. It is precisely this multilayered structure that

distinguishes deep learning from more traditional and widely adopted *Machine Learning* approaches, where feature extraction is typically shallow and often relies on manually engineered representations.

As information flows through the successive layers, the model incrementally builds a hierarchical representation of the underlying data structure. Learning takes place through repeated exposure to examples, during which the model iteratively adjusts its internal parameters in order to reduce the discrepancy between its predictions and the corresponding ground truth. These parameter updates are guided by a loss function and refined over multiple iterations through *optimization algorithms*. Over time, this iterative refinement leads to improved performance, enabling the model to make informed predictions or decisions when presented with previously unseen data. The final output thus reflects the system's ability to generalize from experience, shaped by both the data it has processed and the feedback received during training.

The sequence of steps described above is visually summarized in Figure **1.2**, which is intended to support comprehension through graphical representation. The figure introduces additional details, including more advanced and customized approaches to feature extraction, scaling, and selection. For the sake of clarity and simplicity, these aspects have been collectively referred to in the textual description as data preparation, or more appropriately, data pre-processing.

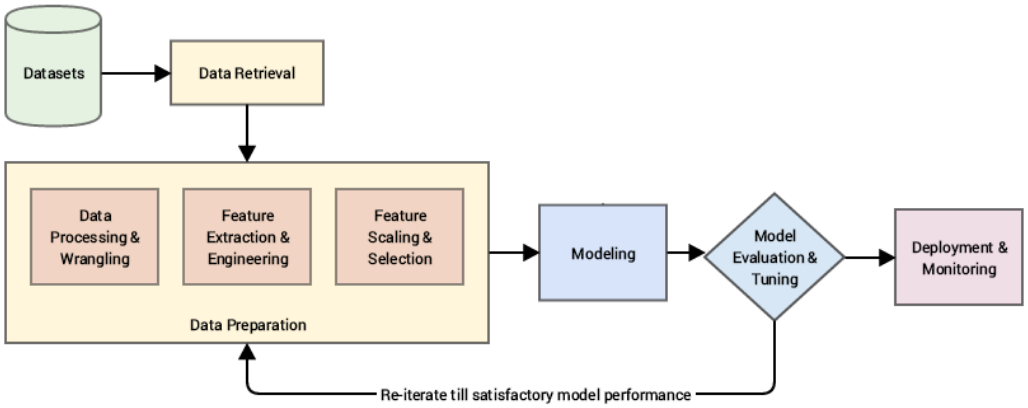

Figure 1.2: A Deep Learning process, from data to output for the user

The learning phase described at a finer level of granularity is commonly divided into training, an optional step for validation, and testing stages. These phases are fundamental to ensuring that the model is capable of generalizing to new data that share characteristics with the samples used during learning. Such generalization is crucial for real-world deployment, where systems must operate on inputs that differ from those encountered during training. As previously noted, during the training phase the model's internal parameters are optimized to minimize the error between the predicted outputs and the ground truth. This minimization process occurs over the surface of a loss function, which quantitatively represents the cost associated with prediction errors. The search for minima within this loss landscape is facilitated by *numerical optimization* methods, which constitute the central focus of the research presented in this thesis.

1.1 The Significance of the Research

As outlined in the previous section, the scope of this research is deeply embedded within the training and validation phases of a deep learning pipeline. Given the intrinsic complexity of the overall process, it is important to acknowledge that errors in model outputs may arise from a wide range of factors, ranging from improper data handling, such as data leakage between training and validation sets, distribution shifts, or inconsistent preprocessing pipelines, to suboptimal model design choices, including inadequate architectural capacity, poorly tuned hyperparameters, or ineffective optimization procedures.

However, when deep learning systems are inadequately trained, significant issues may emerge, particularly as a consequence of errors or limitations in optimization strategies. One of the most immediate and critical consequences is poor generalization performance. In such cases, the model may exhibit strong performance on training data, successfully exploiting learned features and patterns, yet fail to maintain comparable accuracy when evaluated on unseen data in the validation or test phase. This degradation in performance is commonly observed through declining evaluation metrics and unreliable predictions. A frequent cause of this behavior is overfitting, whereby the model memorizes noise or irrelevant details in the training data rather than capturing its underlying structure. Suboptimal choices in optimization methods and hyperparameter settings, or even the intrinsic characteristics of the loss function landscape can lead to slow convergence, convergence to suboptimal solutions, or, in extreme cases, failure to converge altogether.

Another critical concern involves the presence of biased or imbalanced data within the training set, as well as potential mis-specifications in the learning objective and

evaluation protocol. If such issues are not adequately addressed, the model may learn patterns that reflect and potentially amplify pre-existing biases, resulting in skewed or unfair predictions. While the optimizer is not a primary source of bias, it can interact with data imbalance and the chosen loss formulation by shaping the training dynamics. Finally, computational efficiency must also be considered, since poorly matched optimization strategies may lead to excessive training times with limited or no performance improvements.

To translate the concepts discussed thus far into the real-world domestic scenarios introduced at the beginning of this chapter, such errors could manifest, for example, in prolonged periods before a product is able to deliver tailored and effective solutions to the end user, or even in the production of inaccurate or suboptimal outcomes. Moreover, if critical issues are overlooked or implicitly assumed to be negligible, the overall model development cycle may be significantly extended due to poor generalization during testing phases and increased computational demands.

It is important to note that the companies owning the aforementioned processes are generally well aware of these risks and possess advanced technical resources to mitigate them. Consequently, this analogy should be regarded as a deliberate exaggeration, employed solely to maintain consistency with the illustrative examples provided. Nonetheless, the identification and discussion of the issues highlighted above remain highly relevant for individual developers, researchers, and stakeholders who are engaged in the design and deployment of personal or small-scale systems. Effective optimization plays a central role in the success of deep learning models, and neglecting its importance during the training process can severely compromise both model reliability and practical applicability.

In conclusion, optimization effectiveness is not solely determined by the choice of the optimization algorithm itself, but also by its implementation, the computational environment in which it operates, and the specific task it is intended to solve.

1.2 Outline of the Work

In this section, the methodological structure of the dissertation and the organization of the experimental framework are briefly outlined, excluding the present introductory chapter, which is intended to provide a general overview of the topic for non-expert readers. Given that the intended audience of this work includes not only domain experts but also readers who are developing an interest and foundational understanding of deep learning systems, a dedicated chapter is included to present a theoretical overview of the numerical optimization methods that will

later be selected for the benchmark. This chapter serves a dual purpose: on one hand, it facilitates a clearer comprehension of the concepts and results discussed in subsequent sections; on the other, it provides insight into the historical evolution of optimization techniques, situating them within the broader methodological and conceptual framework addressed in this study. Each optimizer will be introduced through its mathematical foundations, followed by a critical discussion of its advantages and limitations, highlighting potential challenges that users may encounter during implementation.

The optimization methods analyzed in this work are presented in a structured order, reflecting their core operational principles, which allow them to be grouped into three broader families of optimizers. This classification facilitates a clearer conceptual comparison by highlighting the fundamental mechanisms that distinguish each class. In particular, the families and corresponding algorithms considered in this study are

- **SGD-based methods:** a family of optimizers derived from stochastic gradient descent that rely on first-order gradient updates with relatively simple update rules and typically require careful learning rate scheduling. This category includes *Stochastic Gradient Descent* (SGD), *SGD with Momentum* (SGDM), and *Layer-wise Adaptive Rate Scaling* (LARS).

- **Adaptive methods:** optimizers that automatically adjust parameter-wise learning rates by leveraging running statistics of past gradients, allowing more flexible updates and often faster convergence in complex optimization landscapes. This group comprises *Root Mean Square Propagation* (RMSProp), *Adaptive Moment Estimation* (Adam), *Adaptive Gradient* (AdaGrad), AdaDelta, AdaFactor, AdaBelief, *Adam with Weight Decay* (AdamW), *Rectified Adam* (RAdam), *Evolved Sign Momentum* (Lion), Lamb, and Yogi.

- **Sharpness-aware methods:** optimization techniques designed to improve generalization by explicitly accounting for the local sharpness of the loss landscape, typically by performing updates that favor flatter minima. Representative methods include *Sharpness-Aware Minimization* (SAM) and *Gap-Guided Sharpness-Aware Minimization* (GSAM).

The selection of the optimization algorithms referenced in this work is guided by two primary considerations. First, the set of methods includes some of the most widely adopted optimization techniques in the field of deep learning. These algorithms are characterized not only by their well-established operational principles, but also by the extensive body of empirical research dedicated to evaluating and comparing their performance across a wide range of applications. In addition

to these established approaches, the study also considers a set of more recent, innovative, and promising optimization methods. The inclusion of this latter group represents a significant and deliberate focus of the present research, as several recent benchmark studies have not incorporated these newer techniques in their evaluations. As a result, this work aims to provide a relatively novel contribution to the literature by extending existing comparative analyses and offering insights into the performance and practical relevance of these emerging optimization strategies.

In order to ensure a rigorous and comprehensive evaluation of the selected optimization algorithms, the experimental design of this dissertation is structured around a task–dataset–model–metric framework. Rather than limiting the analysis to a single application domain, the benchmark encompasses a heterogeneous set of learning problems spanning both computer vision and natural language processing. The tasks considered include image recognition, semantic segmentation, sentiment analysis, text generation, named entity recognition, question answering, machine translation, and text summarization. For each task, two representative datasets have been selected, balancing historical relevance, widespread adoption in the literature, and diversity in scale and complexity. This multi-domain and multi-dataset configuration is intended to reduce task-specific bias and to assess the robustness and generalization behavior of the optimizers across fundamentally different learning paradigms, ranging from pixel-level prediction to sequence-to-sequence generation.

For every task–dataset pair, the comparison is further grounded on a controlled selection of model architectures, including both lightweight models implemented from scratch and more structured or pre-trained architectures commonly employed in contemporary research. This choice allows the analysis to capture the interaction between optimization dynamics and model capacity, architectural depth, and parameterization strategies. Performance evaluation is conducted through task-specific primary metrics complemented by secondary indicators that provide additional insight into qualitative and quantitative behavior. Beyond predictive performance, particular attention is devoted to convergence-related aspects, including training and validation loss trajectories, metric values at convergence, number of epochs required to reach stability, and computational efficiency measured in terms of total runtime, per-epoch runtime, and time-to-convergence. This multidimensional evaluation protocol is designed to provide a nuanced understanding of optimizer behavior, balancing effectiveness, stability, and computational cost. A detailed exposition of the experimental setup, including dataset pre-processing, model configurations, hyperparameter tuning strategy, and evaluation methodology, will be presented in Chapter 3. The empirical results, discussed in Chapters 4 and 5, reveal several findings that challenge prevailing assumptions: RMSProp emerges as the most

consistently strong optimizer across tasks; the classical association between SGD and vision or adaptive methods and NLP proves architecture-dependent rather than universal; and simpler optimizers frequently match more complex recent variants under realistic training budgets.

Ultimately, the objective of this dissertation is to deliver a systematic and empirically grounded benchmark of contemporary optimization algorithms across heterogeneous deep learning scenarios, with the explicit intention of supporting researchers and practitioners in making well-informed and context-aware methodological choices. The comparative study is conceived not merely as an evaluation of peak performance, but as an analysis of optimizer behavior under realistic and practically attainable experimental conditions. A core design principle of this work is therefore reproducibility under limited computational resources. The experimental protocol has been structured to ensure that the reported results can be replicated with minimal infrastructural requirements and without reliance on large-scale distributed systems. In particular, the selection of datasets, the controlled sizing of training subsets where appropriate, the predefined number of training epochs, and the constrained yet systematic hyperparameter exploration strategy have all been carefully calibrated to reflect a low-budget and time-constrained research setting. Rather than pursuing exhaustive hyperparameter searches or extremely long training schedules, the study emphasizes methodological transparency, comparability, and computational efficiency.

This deliberate choice enhances the practical relevance of the findings. The conclusions drawn from the benchmark are intended to be directly applicable not only to well-funded research laboratories, but also to individual researchers, small academic groups, and practitioners operating under limited hardware availability. By situating the experimental evaluation within a realistic resource-aware framework, this dissertation aspires to bridge the gap between theoretical optimizer analysis and the everyday constraints encountered in applied deep learning development.

Chapter 2

Background

Before delving into the main contributions of the thesis, it is beneficial to provide a general overview of the theoretical concepts underlying the tools analyzed in this work, as well as a discussion of the related literature. To this end, this chapter first presents a structured description of the numerical optimization methods that are benchmarked in the experimental analysis, followed by an overview of the relevant related work available to date.

2.1 Numerical Optimization

Prior to examining the theoretical foundations and practical advantages of each individual optimizer, it is necessary to define more precisely the role of numerical optimization within the context of deep learning. As introduced in the previous chapter, optimization in deep learning is intrinsically linked to the definition of an objective function $f(\theta)$, which is typically derived from a loss function that quantifies the empirical risk. The primary goal of the learning process is to minimize this objective function.

Rather than focusing on a specific optimization scenario within a particular computational setting, a more general formulation is adopted in this study. We assume the existence of a prediction function h with a fixed structure, and we consider optimization over its parameter vector $\theta \in \mathbb{R}^d$. Formally, given

$$h(\cdot;\cdot) : \mathbb{R}^{d_x} \times \mathbb{R}^d \to \mathbb{R}^{d_y},$$

Is defined the associated family of prediction functions as

$$H := \{h(\cdot;\theta) : \theta \in \mathbb{R}^d\}.$$

The objective of the optimization process is to identify the function within this family that minimizes the empirical risk. Given an input-output pair (x, y), a loss function ℓ is introduced to quantify the discrepancy between the predicted output and the corresponding ground truth. The incurred loss is expressed as $\ell(h(x;\theta), y)$, where $h(x;\theta)$ denotes the model prediction and y represents the true output [**2**]. The empirical risk can therefore be formulated as an objective function $f(\theta)$, which is minimized through the use of the numerical optimization methods described in subsequent sections.

To conclude this introductory theoretical background, it is useful to formally introduce one of the fundamental components underlying the majority of numerical optimization algorithms, namely the gradient. The gradient of the objective function $f(\theta)$ is defined as the vector composed of its first-order partial derivatives:

$$\nabla f(\theta) = \left(\frac{\partial f(\theta)}{\partial \theta_1}, \frac{\partial f(\theta)}{\partial \theta_2}, \ldots, \frac{\partial f(\theta)}{\partial \theta_d} \right)^\top \in \mathbb{R}^d.$$

The gradient identifies the direction of the steepest ascent of the function $f(\theta)$. In the context of optimization problems—and in particular within deep learning—the objective is instead to minimize $f(\theta)$. For this reason, the negative gradient $-\nabla f(\theta)$ assumes a central role, as it specifies the direction of steepest descent toward a local or global minimum.

Having formalized the learning objective and the role of the gradient, the remainder of this chapter focuses on the optimization strategies adopted in practice to minimize empirical risk. The discussion begins with gradient descent and its stochastic variants, and then moves to adaptive methods that rescale updates using gradient statistics, before concluding with sharpness-aware approaches that explicitly account for local loss geometry.

2.2 Gradient Descent Based Methods

Before describing the optimization methods analyzed in this section, it is useful to briefly review the fundamental principles of the basic Gradient Descent (GD) algorithm, which constitutes the core mechanism underlying all the approaches discussed here.

In this section, the emphasis is placed on methods whose update direction is directly derived from (stochastic) first-order information. These optimizers form the conceptual baseline against which more sophisticated adaptive and sharpness-aware methods will later be compared.

Gradient Descent, whose introduction is commonly attributed to Augustin-Louis Cauchy in 1847 [**3**], belongs to the class of unconstrained numerical optimization methods. It is a first-order iterative algorithm designed to minimize a differentiable, multivariate objective function $f(\theta)$. The method relies on the computation of first-order derivatives and updates the model parameters by moving iteratively in the direction opposite to the gradient of the objective function.

Given a learning rate $\alpha > 0$ and an initial parameter vector θ_0, the Gradient Descent procedure can be summarized as follows:

- Initialize the iteration counter $t = 0$.
- While a predefined convergence criterion is not satisfied:
  1. Increment the iteration index: $t \leftarrow t + 1$.
  2. Compute the gradient of the objective function: $g_t = \nabla_\theta f(\theta_{t-1})$.
  3. Update the parameters: $\theta_t \leftarrow \theta_{t-1} - \alpha g_t$.

The output of the iterative Gradient Descent algorithm is the parameter vector θ_t, which ideally corresponds to a local or global minimum of the objective function. However, this general optimization strategy presents notable limitations. In particular, convergence may require a large number of iterations due to the repeated computation of gradients over the entire dataset, resulting in high computational costs. Moreover, convergence is not always guaranteed, especially in the presence of non-convex objective functions, poorly chosen learning rates, or ill-conditioned loss landscapes. These limitations motivate the development of more advanced gradient-based optimization methods, which are discussed in the following subsections.

A natural first step in this direction is to reduce the per-iteration cost of gradient evaluations, which leads to stochastic and mini-batch formulations of gradient descent.

2.2.1 Stochastic Gradient Descent

One of the most widely adopted optimization algorithms in deep learning is *Stochastic Gradient Descent* (SGD). Originally introduced around 1951 [**4**], well before the advent of modern learning models and large-scale computational implementations, SGD is an iterative optimization method designed to improve upon classical Gradient Descent by introducing stochasticity into the parameter update process. Since its introduction, extensive efforts from both academia and industry have been

devoted to enhancing its runtime efficiency and to establishing a solid theoretical foundation that explains its strong empirical performance.

To address the cited limitations of classical Gradient Descent, particularly its high computational cost and slow convergence due to full-dataset gradient evaluations, stochasticity is introduced into the optimization process. In this formulation, as it can be observed in the algorithm below, the optimization begins with an initialization of the model parameters, after which each update step is performed using the gradient computed from a single randomly selected data sample. Unlike standard Gradient Descent, which requires evaluating the gradient over the entire dataset at each iteration, SGD updates its parameters based on one data point per iteration. This results in significantly reduced computational overhead and enables faster progress during training, especially in large-scale learning scenarios, albeit at the cost of increased variance in the optimization trajectory.

Algorithm 1 Stochastic Gradient Descent Algorithm. The function $f(\theta)$ denotes the objective function to be minimized, and θ_0 represents the initial parameter vector.

procedure STOCHASTIC GRADIENT DESCENT(α, $f(\theta)$, θ_0)
    $t \leftarrow 0$ ▷ Initialize iteration counter
    while convergence criterion is not met **do**
        $t \leftarrow t + 1$ ▷ Increment iteration counter
        Sample a random data point x_t from the dataset
        $g_t \leftarrow \nabla_\theta f(\theta_{t-1}; x_t)$ ▷ Compute stochastic gradient
        $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot g_t$ ▷ Update parameters
    end while
    return θ_t ▷ Return optimized parameters
end procedure

By computing the gradient at each iteration and updating the parameters in the opposite direction, the method aims to follow the path of steepest descent toward a minimum of the objective function. Operating on a single data instance at each update step—thus relying on a limited subset of the available data—significantly reduces the computational cost associated with gradient evaluation and enables efficient training on large-scale datasets.

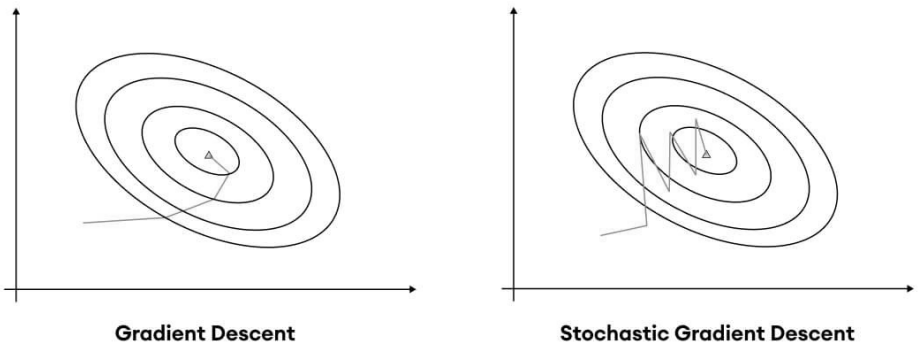

Figure 2.1: Comparison of optimization trajectories for Gradient Descent and Stochastic Gradient Descent

The representation above illustrates the optimization trajectories of the two methods discussed thus far: standard Gradient Descent on the left and Stochastic Gradient Descent on the right. From the figure and the distinct update paths followed by the two iterative methods, it can be inferred that SGD typically performs faster individual updates than standard Gradient Descent, as it processes one data point at a time. This property makes each update computationally cheaper, but it also introduces a higher variance in the estimated gradient. Consequently, the update direction at each iteration represents a noisy approximation of the true gradient of the full objective function. As a result, SGD often requires a larger number of iterations to approach the same region of the parameter space that could be reached with full-batch Gradient Descent. Nevertheless, due to the significantly lower computational cost per update, the overall training time may still be reduced in practice. As a result, while SGD enables rapid and computationally efficient updates, it often requires a larger number of iterations to converge. In summary, SGD offers fast per-iteration computations at the expense of slower and less stable convergence.

Despite the advantages discussed above, it is necessary to address the limitations of the stochastic approach in order to motivate the development of more advanced optimization algorithms. The inherent randomness in sample selection can help prevent the optimization process from becoming trapped in local minima; however, it can also increase the difficulty of converging to a global minimum. To mitigate this issue, learning rate scheduling strategies are often employed. These strategies are conceptually analogous to the annealing process in metallurgy, in which molten metal is gradually cooled [**5**]. Similarly, the learning rate is initially set to a relatively large value and progressively reduced over time, allowing the

optimization process to explore the parameter space early on and later stabilize around a minimum.

Another notable variant of Gradient Descent is the *Mini-batch Gradient Descent* method. While standard Gradient Descent computes gradients using the entire dataset and SGD relies on a single data point per iteration, the mini-batch approach estimates gradients using a small subset of samples. This compromise offers improved stability over SGD while maintaining computational efficiency. Moreover, mini-batch processing enables effective exploitation of hardware acceleration, particularly when leveraging matrix operations on GPUs.

In conclusion, the experimental analysis conducted in this thesis employs the Stochastic Gradient Descent optimization method. Several software libraries provide robust support for SGD-based training; in particular, this work utilizes the `torch.optim` module.

2.2.2 SGD with Momentum

Stochastic Gradient Descent with Momentum (SGD with Momentum) represents an extension of the standard SGD algorithm, originally inspired by the concept of momentum introduced by Boris Teodorovich Polyak in 1964 [**6**]. This refinement is specifically designed to mitigate some of the well-known limitations of vanilla SGD. Although SGD has demonstrated strong empirical performance across a wide range of optimization problems, it often suffers from slow convergence rates and may experience stagnation in regions of the loss landscape characterized by plateaus or highly inconsistent gradients.

The introduction of momentum augments the optimization process with a mechanism that can be interpreted as a form of memory. Rather than relying exclusively on the current gradient information, the optimizer incorporates a weighted accumulation of past gradients into each parameter update. This accumulated information allows the optimization trajectory to preserve a consistent direction of descent, thereby reducing sensitivity to noisy or erratic gradient estimates and promoting more stable convergence behavior.

A useful intuition for understanding the momentum mechanism can be drawn from classical physics. Consider a ball rolling down a sloped surface: its motion at any given point is not determined solely by the local slope, but also by the velocity accumulated from its previous trajectory. Analogously, in SGD with Momentum, parameter updates are influenced not only by the current stochastic gradient but also by the history of past gradients. This effect enables the optimizer

to maintain progress along directions that consistently reduce the objective function.

The benefits of this approach are particularly evident in regions of the loss surface exhibiting narrow ravines, where curvature differs significantly across dimensions. In such scenarios, standard SGD tends to oscillate across steep directions, resulting in inefficient progress along the flatter axis. Momentum alleviates this issue by dampening oscillations and accelerating movement along the relevant descent direction, leading to faster and more reliable convergence.

To further enter in detail, we now analyze the iterative implementation of this optimizer. The main idea is to have at each step the contribution of the before cited velocity vector: this is the combination of the current gradient with a potentially discounted history of former gradients. The algorithm proceeds as follows:

Algorithm 2 The momentum coefficient β controls the contribution of past gradients, while α is the learning rate

procedure SGD WITH MOMENTUM(α, β, $f(\theta)$, θ_0)
    $t \leftarrow 0$ ▷ Initialize timestep
    $v_0 \leftarrow 0$ ▷ Initialize velocity vector
    while not converged **do**
        $t \leftarrow t + 1$ ▷ Update timestep
        Sample a random data point x_t from the dataset
        $g_t \leftarrow \nabla_\theta f(\theta_{t-1}; x_t)$ ▷ Compute stochastic gradient
        $v_t \leftarrow \beta \cdot v_{t-1} + (1 - \beta) \cdot g_t$ ▷ Update velocity
        $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot v_t$ ▷ Update parameters
    end while
    return θ_t ▷ Return optimized parameters
end procedure

The momentum mechanism can be interpreted as a form of moving average applied to the gradient updates, providing a smoothed estimate of the most suitable descent direction. By aggregating information from multiple past gradients, this approach reduces the impact of noisy or misleading individual estimates, allowing the optimization process to maintain consistent and meaningful progress across iterations.

One of the primary advantages of momentum lies in its ability to mitigate the tendency of standard SGD to exhibit oscillatory or meandering behavior when successive gradient estimates point in heterogeneous or conflicting directions. The maintenance of a running average effectively acts as a noise-filtering mechanism,

attenuating high-frequency fluctuations while reinforcing persistent descent directions. This smoothing effect enhances the stability of the optimization trajectory and can significantly accelerate convergence, particularly in challenging scenarios where the loss surface is irregular or highly non-uniform.

In addition, the incorporation of past velocity information enables the optimizer to escape shallow local minima or extended flat regions of the loss landscape that often hinder vanilla SGD. The reliance on accumulated gradient information allows the algorithm to traverse such regions more effectively, increasing the likelihood of reaching solutions of higher quality. This advantage becomes increasingly relevant as model architectures grow in complexity and their associated loss landscapes become more intricate.

Despite these benefits, the use of momentum introduces additional hyperparameter sensitivity. The momentum coefficient, denoted by β, determines the extent to which past gradients influence the current update. If β is set too low, the optimizer may fail to fully exploit the advantages of momentum; conversely, excessively high values can lead to overshooting optimal points and unstable convergence behavior. In practice, commonly adopted values for β range between 0.8 and 0.99, with 0.9 often serving as a reliable default.

The learning rate, α, also plays a critical role in conjunction with momentum. Although the stabilizing effect of momentum may permit the use of larger learning rates, combining a high learning rate with strong momentum can result in overly aggressive updates and convergence issues. For this reason, momentum is frequently employed alongside learning rate scheduling strategies, which progressively reduce the step size over the course of optimization to ensure stable and effective convergence.

Finally, it is worth noting that, in practical implementations, incorporating momentum into SGD requires minimal additional effort, as it is typically achieved by specifying an extra hyperparameter. Analogously to the standard case discussed previously, the momentum-based variant employed in our benchmark experiments was implemented using the `torch.optim` module.

While momentum substantially improves optimization stability, applying a single global learning rate can remain problematic when training with very large batch sizes, where update magnitudes may become imbalanced across layers. This observation motivates layer-wise scaling strategies, discussed next through the LARS optimizer.

2.2.3 Layer-wise Adaptive Rate Scaling

Layer-wise Adaptive Rate Scaling (LARS) is a gradient-based optimization algorithm specifically designed to address the numerical instability and performance degradation that commonly arise when training deep neural networks with very large batch sizes. The method was introduced by You et al.[7] in 2017 within the context of large-scale distributed learning, where conventional Stochastic Gradient Descent (SGD), even when augmented with momentum, linear learning-rate scaling, and warm-up strategies, was shown to either diverge or converge to suboptimal solutions.

The key motivation behind LARS stems from an empirical observation regarding the heterogeneous behavior of different layers in deep architectures. In particular, during training, the ratio between the norm of the parameters and the norm of their corresponding gradients can vary significantly across layers. When a single global learning rate is applied uniformly, as is standard practice in SGD and its momentum-based variants, this disparity may result in disproportionately large parameter updates for some layers and overly conservative updates for others. Such imbalances are especially detrimental during the initial training phase and become increasingly severe as the batch size grows.

To alleviate this issue, LARS employs *layer-wise adaptive* learning rates that explicitly account for the scale of both parameters and gradients at each layer. Let w_l denote the parameter vector associated with layer l, and let $g_l = \nabla_{w_l} f(w)$ represent the corresponding gradient. The local learning rate for layer l is defined as

$$\lambda_l = \eta \cdot \frac{\|w_l\|}{\|g_l\|},$$

where $\eta \in (0,1)$ is a *trust coefficient* that regulates the magnitude of the update relative to the parameter norm. This layer-specific scaling is combined with a global learning rate α to compute the final update. By normalizing the update magnitude with respect to the scale of each layer's parameters, LARS effectively harmonizes learning dynamics across the network, thereby enhancing training stability and enabling efficient optimization in large-batch regimes.

Empirical results reported by You et al. [7] demonstrate that LARS enables the successful training of deep convolutional architectures such as AlexNet and ResNet–50 using batch sizes of up to 32K without incurring any loss in final accuracy, provided that a sufficiently large number of optimization steps is performed. These findings indicate that, in large-batch training regimes, optimization challenges are primarily attributable to improper scaling of parameter updates rather than to intrinsic limitations in the generalization capability of the models.

The principal advantage of LARS lies in its capacity to stabilize and scale the optimization process when operating with very large batch sizes. By normalizing parameter updates according to layer-wise weight norms, LARS allows for the use of substantially larger global learning rates while mitigating the risk of divergence, thereby improving computational efficiency in distributed and high-throughput training settings. Furthermore, LARS integrates seamlessly with SGD with Momentum and avoids per-parameter adaptivity, thus preserving the favorable generalization properties traditionally associated with SGD-based optimization methods. Despite these advantages, LARS presents limitations: firstly, the computation of parameter and gradient norms at each iteration introduces a modest additional computational overhead. Moreover, the benefits of LARS tend to diminish in small- or moderate-batch regimes, where simpler optimization algorithms often achieve comparable performance with reduced algorithmic complexity.

Unlike SGD and SGD with Momentum, LARS is not provided as a native optimizer within the `torch.optim` module. As a result, for the experimental evaluation conducted in this thesis, LARS was implemented from scratch by extending PyTorch's `optimizer` base class. The implementation adheres closely to the original formulation proposed by You et al., while adopting a minimal and pragmatic design compatible with PyTorch's optimization framework. This approach preserves the fundamental principles of LARS and introduces only minor practical adaptations to ensure interoperability with PyTorch and to enable a fair comparison with the other optimizers considered in the experimental benchmark.

The layer-wise viewpoint adopted by LARS highlights that optimization difficulties may arise not only from stochasticity, but also from heterogeneous parameter scales. A complementary line of work addresses this heterogeneity by adapting the learning rate at the level of individual parameters, which is the focus of the following section on adaptive methods.

2.3 Adaptive Methods

In order to overcome the limitations inherent to classical Gradient Descent and its stochastic variants, a wide range of *adaptive gradient-based optimization methods* has been proposed in the literature. These methods modify the basic GD update rule by introducing mechanisms that adapt the learning rate throughout the training process, often at the level of individual parameters. The primary objective is to improve convergence properties in the presence of ill-conditioned optimization landscapes, non-stationary gradients, and heterogeneous parameter sensitivities,

which are commonly encountered in large-scale and deep learning models.

In the experimental benchmark developed in this thesis, adaptive optimizers are considered both in their classical forms (e.g., RMSProp and Adam) and in more recent variants designed to improve stability, memory efficiency, or large-batch behavior (e.g., Adafactor, RAdam, Lion, and LAMB). The goal of the present section is to introduce their defining mechanisms and clarify their practical trade-offs.

Adaptive optimizers typically leverage statistics of past gradients, such as their first-order moments and second-order moments, to rescale parameter updates. By normalizing gradient magnitudes, these methods reduce sensitivity to the initial learning rate and enable more stable optimization across different layers and parameter groups. A detailed analysis of these methods is therefore essential to understand their practical and theoretical trade-offs, and is provided in the following subsections.

2.3.1 Root Mean Square Propagation

Root Mean Square Propagation (RMSProp) is an adaptive optimization algorithm originally introduced by Geoffrey Hinton in 2012 during the Coursera course *Neural Networks for Machine Learning* [**8**]. Although it was not formally published at the time, RMSProp has since become a widely adopted method, designed to overcome several limitations of earlier gradient-based optimization techniques.

RMSProp builds upon the intuition behind Stochastic Gradient Descent (SGD) with Momentum, which incorporates a memory term to smooth parameter updates and reduce sensitivity to noisy gradients. In contrast, the key idea underlying RMSProp is the dynamic adaptation of the learning rate on a per-parameter basis. This is achieved by exploiting information about the magnitude of recently observed gradients, allowing the optimizer to better handle scenarios in which gradient scales vary significantly across dimensions.

More specifically, RMSProp maintains an exponentially decaying moving average of the squared gradients for each parameter. This quantity is subsequently used to normalize the current gradient, effectively scaling down updates associated with large gradients while amplifying those corresponding to smaller ones. Such normalization mitigates unstable learning dynamics and is particularly beneficial in loss landscapes characterized by anisotropic curvature, where different parameters may require updates of substantially different magnitudes. The update rules of RMSProp are summarized in the algorithm presented in the following page.

Algorithm 3 RMSProp

procedure RMSProp(α, β, ϵ, $f(\theta)$, θ_0)
  $t \leftarrow 0$ ▷ Initialize timestep
  $s_0 \leftarrow 0$ ▷ Initialize accumulation vector
  while not converged **do**
    $t \leftarrow t + 1$ ▷ Update timestep
    Sample a random data point x_t from the dataset
    $g_t \leftarrow \nabla_\theta f(\theta_{t-1}; x_t)$ ▷ Compute stochastic gradient
    $s_t \leftarrow \beta \cdot s_{t-1} + (1 - \beta) \cdot g_t^2$ ▷ Update squared gradient accumulation
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \frac{g_t}{\sqrt{s_t + \epsilon}}$ ▷ Update parameters
  end while
  return θ_t ▷ Return optimized parameters
end procedure

In this formulation, s_t represents the exponentially decaying average of past squared gradients. The constant ϵ is introduced to ensure numerical stability and to prevent division by zero. By dividing the gradient by $\sqrt{s_t + \epsilon}$, RMSProp effectively normalizes each update with respect to the recent gradient history.

This normalization mechanism enables RMSProp to perform robustly in regions of the loss surface where the steepness varies considerably across different directions. Compared to standard SGD, which applies a uniform learning rate to all parameters, RMSProp dynamically adjusts the effective step size in each dimension. As a result, it reduces oscillations along directions of high curvature while accelerating progress along flatter directions, leading to more stable and efficient convergence behavior.

The principal advantages of RMSProp stem from its adaptive nature. First, the algorithm assigns an individual learning rate to each parameter, which is particularly advantageous in high-dimensional optimization problems where parameters may exhibit heterogeneous sensitivities. This adaptability often translates into faster convergence than vanilla SGD, especially in the presence of ill-conditioned or highly non-convex loss landscapes. Second, RMSProp is well suited for scenarios involving sparse or intermittent gradients, such as those commonly encountered in natural language processing or recommendation systems. The moving average of squared gradients ensures that learning rates remain well-behaved even when gradients are infrequently non-zero.

Despite these advantages, RMSProp also presents certain limitations, primarily related to hyperparameter selection. The decay rate β plays a critical role in determining the responsiveness of the moving average: values close to 1 may

cause overly slow adaptation, whereas values too close to 0 can make the algorithm excessively sensitive to recent gradients. In practice, β is typically set to 0.9. Similarly, the learning rate α must be carefully tuned, as the adaptive normalization may otherwise result in updates that are either too aggressive or overly conservative.

From an implementation standpoint, RMSProp is relatively simple and is natively supported by modern deep learning frameworks. In this work, the optimizer is implemented using the `torch.optim` module provided by PyTorch. Leveraging a built-in implementation ensures computational efficiency and consistency with the standard RMSProp formulation as established in the literature.

2.3.2 Adaptive Moment Estimation

Adam (Adaptive Moment Estimation) constitutes a significant milestone in the development of optimization algorithms for machine learning. The fundamental motivation behind Adam is the integration of the most effective characteristics of RMSProp and momentum-based Stochastic Gradient Descent (SGD) within a single, unified optimization framework. The algorithm was introduced in 2015 by Diederik P. Kingma and Jimmy Ba at the *International Conference on Learning Representations* [**9**]. Since its introduction, Adam has rapidly gained widespread adoption and has become one of the most commonly employed optimization methods in deep learning, owing to its consistently robust performance across a broad range of architectures and application domains. This effectiveness stems from its ability to simultaneously incorporate momentum-like update dynamics and adaptive, parameter-specific learning rates. Furthermore, Adam introduces dedicated mechanisms to improve numerical stability during the early stages of training through the application of bias-correction techniques.

As discussed previously, the Adam optimizer is founded on two key principles inherited from earlier optimization methods. From momentum-based approaches, Adam adopts the concept of maintaining an exponentially decaying moving average of past gradients, commonly referred to as the first moment estimate. This mechanism facilitates accelerated convergence, particularly in regions of the optimization landscape characterized by consistent gradient directions. In parallel, Adam incorporates from RMSProp the use of an exponentially decaying moving average of past squared gradients, denoted as the second moment estimate, which enables adaptive learning rate scaling on a per-parameter basis. What distinguishes Adam from these precursor methods is the synergistic combination of first and second moment estimates, further enhanced by the application of bias-correction procedures that mitigate the effects of initialization-induced estimation errors, thereby improving convergence behavior and overall optimization stability.

The mathematical formulation of the Adam optimizer, as outlined above, involves maintaining two distinct state vectors for each trainable parameter. This design enables the simultaneous estimation of first- and second-order moments of the gradient, resulting in an efficient and scalable optimization procedure, which can be formally expressed as follows:

Algorithm 4 Adam: β_1 and β_2 control the decay rates of the moment estimates, α is the learning rate, and ϵ ensures numerical stability. The bias correction terms ($\hat{m}_t$ and $\hat{v}_t$) are crucial for initial training stability.

procedure ADAM(α, β_1, β_2, ϵ, $f(\theta)$, θ_0)
  $t \leftarrow 0$ ▷ Initialize timestep
  $m_0 \leftarrow 0$ ▷ Initialize 1st moment vector
  $v_0 \leftarrow 0$ ▷ Initialize 2nd moment vector
  while not converged **do**
    $t \leftarrow t + 1$
    Sample minibatch B_t from training data
    $g_t \leftarrow \nabla_\theta \frac{1}{|B_t|} \sum_{x \in B_t} f(\theta_{t-1}; x)$ ▷ Compute minibatch gradient
    $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ ▷ Update biased 1st moment
    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ ▷ Update biased 2nd moment
    $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ ▷ Correct 1st moment bias
    $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ ▷ Correct 2nd moment bias
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ ▷ Update parameters
  end while
  return θ_t
end procedure

For readers without a strong technical background, it may be beneficial to employ the previously introduced analogy, particularly given that the complexity of the Adam optimizer exceeds that of all optimization methods discussed thus far. The underlying intuition of Adam can be illustrated through the metaphor of a hypothetical spacecraft navigating through an asteroid field. In order to successfully traverse this environment and maintain its trajectory, the spacecraft's onboard navigation system must simultaneously account for two critical factors: the velocity accumulated from past movements up to a given time instant, which can be interpreted as momentum, and the spatial density of surrounding matter in different directions. This density may vary significantly across directions, analogous to the heterogeneous magnitudes observed in gradient components.

Conventional optimization algorithms typically address these two aspects in isolation. In contrast, Adam continuously refines its update direction by synthesizing

both sources of information, while also compensating for initial calibration inaccuracies within the navigation system. In the context of optimization, this compensation corresponds to the bias-correction mechanism applied to the moment estimates.

It is particularly important to emphasize that the bias-correction procedure represents one of the most significant innovations introduced by the Adam optimizer. During the early stages of training, when the iteration index t is small, the correction terms $(1 - \beta_1^t)$ and $(1 - \beta_2^t)$ play a crucial role in adjusting the estimates of the first and second moments of the gradient, respectively, compensating for their initialization at zero. As training progresses and t increases, both correction factors asymptotically converge to unity, thereby ensuring a smooth transition toward the uncorrected moment estimates.

Furthermore, the adaptive learning rate mechanism proves highly effective in normalizing parameter updates based solely on the historical magnitudes of past gradients. This property prevents excessive and unnecessary amplification of updates for parameters associated with large gradients, while allowing proportionally larger updates for parameters characterized by smaller gradient magnitudes. Concurrently, the momentum component contributes to maintaining a stable progression along consistent and, presumably, favorable descent directions, thereby enhancing both convergence stability and optimization efficiency.

Continuing the analysis of the advantages offered by the Adam optimizer, it can be observed that the algorithm exhibits a high degree of robustness with respect to hyperparameter selection. Owing to its adaptive design, Adam proves particularly effective in addressing challenging optimization scenarios characterized by the following conditions:

- Loss landscapes dominated by sparse or noisy gradients, a situation commonly encountered in natural language processing tasks;
- High-dimensional parameter spaces, in which individual parameters may require substantially different learning rates;
- Strongly heterogeneous curvature across different dimensions of the optimization surface;
- Training regimes constrained to the use of small batch sizes.

Based on the considerations presented in this section, Adam emerges as a practical and reliable optimization choice, offering both efficiency and versatility across a wide range of heterogeneous learning tasks. Nevertheless, it is important to acknowledge that recent theoretical analyses have identified specific scenarios in

which Adam may converge to suboptimal solutions, particularly within certain convex [**10**] and non-convex [**11**] optimization settings. This behavior can be attributed to the algorithm's adaptive nature, which may lead to an excessive reliance on early gradient information, potentially hindering exploration of more favorable regions of the optimization landscape that would require temporary deviation from the prevailing descent direction.

Additionally, due to the requirement of maintaining both first- and second-moment estimates for each parameter, Adam incurs a higher memory footprint and slightly increased computational overhead when compared to standard stochastic gradient descent (SGD). While this cost is often negligible in practice, it nevertheless represents a trade-off to be considered when selecting an optimization strategy.

To conclude, from an implementation perspective, Adam is relatively straightforward to employ and is natively supported by modern deep learning frameworks. In this work, the optimizer is implemented using the `torch.optim` module provided by PyTorch. Relying on the built-in implementation ensures computational efficiency, numerical stability, and full adherence to the standard Adam formulation as defined in the original literature, including the correct handling of moment estimation and bias-correction mechanisms.

2.3.3 Adaptive Gradient

AdaGrad (Adaptive Gradient) is among the earliest optimization algorithms to introduce adaptive, parameter-specific learning rates. The method was originally proposed in 2011 [**12**], preceding the development of Adam by several years. AdaGrad is founded on the key insight that learning rates should be adjusted individually for each parameter based on the historical behavior of its gradients. Specifically, the algorithm adapts the learning rate by scaling it with the inverse square root of the accumulated sum of past squared gradients, as formalized below [**13**]:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{G_t + \epsilon}} \cdot g_t \tag{2.1}$$

In this formulation, α denotes the initial learning rate, while ϵ is a small constant introduced to prevent numerical instability arising from division by zero. The term $\sqrt{G_t + \epsilon}$ acts as an adaptive scaling factor applied independently to each parameter, and $G_t := \sum_{k=1}^{t} g_k(\theta_{k-1}) g_k(\theta_{k-1})^T$ represents the cumulative sum of squared gradients, effectively serving as a memory of all past gradient information.

In contrast to Adam, which relies on exponentially decaying moving averages

to estimate gradient moments, AdaGrad employs a cumulative aggregation strategy. As a consequence, the effective learning rates decrease monotonically over time. This property makes AdaGrad particularly well-suited for optimization problems involving sparse data, where certain features occur infrequently yet carry significant informational value, as the algorithm naturally assigns relatively larger learning rates to such parameters.

However, the unbounded accumulation of squared gradients may lead to an overly aggressive decay of the learning rate, potentially resulting in premature convergence or stagnation. This limitation can hinder performance in long training regimes or in scenarios where continued parameter updates are necessary to reach an optimal solution.

To integrate AdaGrad within our proposed method, the optimizer is instantiated using the `torch.optim` module provided by PyTorch. Leveraging the built-in implementation ensures computational efficiency and consistency with the canonical AdaGrad formulation as previously established.

2.3.4 AdaDelta

AdaDelta was introduced in 2012 [**14**] as a refinement of AdaGrad, explicitly addressing the latter's progressive decay of the effective learning rate. The key contribution of AdaDelta consists in replacing the unbounded accumulation of squared gradients with an exponentially decaying average, thereby preventing the learning rate from vanishing over time. This mechanism is conceptually analogous to the second-moment estimation later adopted by Adam. In addition, AdaDelta explicitly adapts the magnitude of parameter updates, as reflected in the update rule:

$$\Delta\theta_t = -\frac{\sqrt{E[\Delta\theta^2]_{t-1} + \epsilon}}{\sqrt{E[g^2]_t + \epsilon}}\, g_t. \tag{2.2}$$

The terms involved in this formulation are defined as follows:

- $E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2$ denotes the exponentially weighted moving average of the squared gradients. Here, g_t represents the gradient evaluated at iteration t, while ρ is a decay coefficient (typically close to one, e.g., 0.95 or 0.99) that determines the relative influence of historical versus current gradient information.
- $E[\Delta\theta^2]_t = \rho E[\Delta\theta^2]_{t-1} + (1-\rho)\Delta\theta_t^2$ represents the exponentially weighted moving average of squared parameter updates, thereby capturing the typical scale of past updates applied to the parameters.

From this formulation, AdaDelta can be regarded as a distinctive optimization method in that it does not rely on an explicit global learning rate. Instead, the update magnitude is entirely governed by the ratio between the accumulated update and gradient statistics. While this property removes the need for manual tuning of a learning rate parameter α, it may also lead to reduced controllability of the optimization dynamics when compared to methods that retain an explicit learning rate.

Within the proposed methodology, AdaDelta is incorporated by instantiating the corresponding optimizer from the `torch.optim` module in PyTorch. The use of the framework's native `Adadelta` implementation provides an efficient and reliable realization of the algorithm, while remaining faithful to its standard formulation.

2.3.5 AdaFactor

Adafactor is an adaptive optimization algorithm introduced as a memory-efficient alternative to Adam, particularly designed for large-scale neural networks with high-dimensional parameter spaces [**15**]. While retaining the core principle of parameter-wise adaptive learning rates, Adafactor addresses one of Adam's primary limitations, namely the quadratic memory cost associated with storing second-order moment estimates for each parameter.

The key innovation of Adafactor lies in its factorized approximation of the second-moment accumulator. For matrix-valued parameters, instead of maintaining a full second-moment matrix, Adafactor decomposes it into the outer product of row-wise and column-wise statistics, thereby reducing memory consumption from $\mathcal{O}(nm)$ to $\mathcal{O}(n+m)$. The parameter update rule can be expressed as:

$$\theta_{t+1} = \theta_t - \alpha_t \frac{g_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.3}$$

In the equation, g_t denotes the gradient at iteration t, ϵ is a small constant for numerical stability, and $\hat{v}_t$ represents the factorized approximation of the second moment of the gradients. Unlike Adam, Adafactor typically omits the explicit first-moment (momentum) estimate, relying instead on the variance normalization alone.

Furthermore, Adafactor introduces a learning-rate scaling strategy that is often proportional to the root mean square of the parameters themselves, enabling the use of relative step sizes. This design choice allows the optimizer to operate effectively without a manually tuned global learning rate, improving robustness across a wide range of training configurations.

By significantly reducing memory overhead while preserving adaptive behavior, Adafactor is particularly well-suited for training very large models, such as transformer-based architectures, where optimizer state size can otherwise become a dominant bottleneck. However, the absence of explicit momentum and the approximated nature of the second-moment estimates may lead to less stable updates in certain optimization landscapes, requiring careful consideration depending on the task and model architecture.

In this work, Adafactor was implemented by adhering to its fundamental design principles, namely factored second-moment estimation for matrix-valued parameters and RMS-based update clipping, while employing a non-factored variant for lower-dimensional tensors. To reduce implementation complexity, elements of the original formulation were simplified, including the full parameter-scale relative update mechanism and the complete learning-rate scheduling heuristics proposed in the original work. This resulted in a streamlined optimizer that preserves the core adaptive behavior of Adafactor while remaining computationally efficient and suitable for the scope of the present experiments.

To conclude, it is to note that, at the time the experimental campaign was conducted, an official Adafactor implementation was not yet available in the `torch.optim` library. Consequently, the experiments were completed using the custom implementation described above, and a direct comparison with the subsequently released reference implementation is deferred to future work.

2.3.6 AdaBelief

AdaBelief is an adaptive gradient-based optimization algorithm proposed as a refinement of Adam, with the objective of improving generalization and training stability by modifying the manner in which second-order information is estimated [**16**]. While preserving Adam's core structure, including momentum-based first-order updates and parameter-wise adaptive learning rates, AdaBelief introduces a principled reinterpretation of the second-moment accumulator based on the notion of gradient "belief".

The central idea behind AdaBelief is that the variance of the gradient should be measured relative to its exponential moving average rather than its raw magnitude. Instead of accumulating the squared gradients directly, AdaBelief tracks the squared deviation of the current gradient from its first-moment estimate. This adjustment allows the optimizer to adapt its step size based on how predictable or surprising the gradient is, assigning larger learning rates to parameters with consistent gradients and smaller updates to those exhibiting high uncertainty. The

update equations are given by:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{2.4}$$

$$s_t = \beta_2 s_{t-1} + (1 - \beta_2)(g_t - m_t)^2 \tag{2.5}$$

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{s}_t} + \epsilon} \tag{2.6}$$

Here, m_t denotes the first-moment estimate, s_t represents the exponential moving average of the squared gradient residuals, β_1 and β_2 are decay coefficients, and $\hat{m}_t$ and $\hat{s}_t$ are bias-corrected versions of the corresponding moments. The term ϵ is introduced to ensure numerical stability.

By modeling the second moment as the variance around the predicted gradient direction, AdaBelief effectively bridges the behavior of adaptive optimizers and stochastic gradient descent with momentum. In regions of the loss landscape where gradients are stable, the algorithm behaves similarly to SGD, promoting better generalization, while in regions of high gradient variability it retains the robustness of adaptive methods such as Adam.

Empirical results reported in the original work demonstrate that AdaBelief consistently matches or outperforms Adam across a wide range of tasks, including image classification and language modeling, often achieving improved convergence speed and reduced test error without additional hyperparameter tuning. Importantly, AdaBelief maintains the same computational and memory complexity as Adam, making it a drop-in replacement in most training pipelines.

In this work, AdaBelief was implemented in close alignment with its original formulation, employing bias-corrected exponential moving averages for both the first-order gradients and the gradient residuals that define the belief-based second moment. In addition, a decoupled weight decay scheme was adopted to mirror the core update mechanism proposed by Zhuang et al. In conclusion, minor simplifications were introduced for practicality, resulting in a streamlined yet faithful implementation that preserves the essential behavior of AdaBelief while ensuring comparability with other adaptive optimizers considered in the experimental evaluation.

2.3.7 Adam with Weight decay

Adam with Weight decay(AdamW) was introduced in 2017 by Loshchilov and Hutter [**17**] as a principled modification of the Adam optimizer, aimed at correcting

the interaction between adaptive learning rates and L_2 regularization. While Adam incorporates weight decay implicitly through the gradient of the regularization term, AdamW decouples weight decay from the adaptive gradient update, thereby restoring its intended behavior as a true regularization mechanism. This reformulation improves both theoretical consistency and empirical generalization performance, particularly in deep learning settings.

The AdamW update rule can be expressed as follows:

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} - \alpha \lambda \theta_t, \tag{2.7}$$

In the formulation, α denotes the learning rate, λ is the weight decay coefficient, and ϵ is a small constant introduced for numerical stability. The terms $\hat{m}_t$ and $\hat{v}_t$ correspond to bias-corrected first- and second-moment estimates of the gradients, inherited from the original Adam formulation.

More specifically, the moment estimates are defined as:

- $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$ represents the exponentially weighted moving average of past gradients, capturing first-order moment information.
- $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$ denotes the exponentially weighted moving average of squared gradients, providing an estimate of second-order moment statistics.

Bias correction is applied to compensate for initialization effects at early iterations, yielding $\hat{m}_t = m_t/(1-\beta_1^t)$ and $\hat{v}_t = v_t/(1-\beta_2^t)$. In contrast to Adam, the weight decay term $-\alpha\lambda\theta_t$ is applied directly to the parameters rather than being absorbed into the gradient computation. This decoupling ensures that the regularization strength remains independent of the adaptive learning rate dynamics.

As a result, AdamW can be interpreted as preserving the advantages of Adam—namely fast convergence and robustness to gradient scaling—while providing a more faithful implementation of weight decay. This property has made AdamW a widely adopted optimizer in modern deep learning architectures.

Within the proposed methodology, AdamW is employed through the corresponding implementation available in the `torch.optim` module of PyTorch. Relying on the framework's native `AdamW` optimizer ensures computational efficiency and adherence to the standard algorithmic definition presented in the original work.

2.3.8 Rectified Adam

Rectified Adam (RAdam) was proposed in 2019 by Liu et al. [**18**] with the objective of improving the stability of Adam during the early stages of training. Although

Adam exhibits fast convergence and robustness to gradient scaling, empirical evidence shows that its adaptive learning rate can suffer from excessive variance in the initial iterations, especially when the second-moment estimate is still unreliable. RAdam addresses this issue by introducing a rectification mechanism that adaptively controls the variance of the learning rate, effectively bridging the gap between Adam and stochastic gradient descent with momentum.

RAdam builds upon the standard Adam framework by retaining the exponentially decaying estimates of the first and second moments of the gradients,

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2,$$

together with their bias-corrected counterparts. The key distinction lies in the introduction of a variance rectification term derived from the analytical variance of the adaptive learning rate. Specifically, RAdam computes the length of the approximated simple moving average of the second moment,

$$\rho_t = \rho_\infty - \frac{2t\beta_2^t}{1-\beta_2^t}, \qquad \rho_\infty = \frac{2}{1-\beta_2} - 1,$$

which quantifies the reliability of the variance estimate at iteration t.

When ρ_t exceeds a predefined threshold, the adaptive update is rectified by a factor that corrects for variance inflation, yielding a stable Adam-like update. Conversely, when ρ_t is small and the variance estimate is deemed unreliable, RAdam falls back to a momentum-based update resembling SGD with momentum. The resulting parameter update can therefore be expressed as

$$\theta_{t+1} = \theta_t - \alpha\, r_t \, \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon},$$

where r_t denotes the rectification factor, which smoothly transitions between the two regimes.

The primary advantage of RAdam lies in its ability to stabilize training in the early optimization phase without requiring manual warm-up schedules, which are often employed with Adam to mitigate convergence issues. By adaptively controlling the variance of the learning rate, RAdam reduces sensitivity to the choice of hyperparameters and improves robustness across a wide range of tasks. Furthermore, as training progresses and the variance estimate becomes reliable, RAdam naturally recovers the behavior of standard Adam, preserving its fast convergence properties. On the other hand, the rectification mechanism introduces additional computations and hyperparameters, marginally increasing algorithmic complexity. While this overhead is typically negligible in practice, the benefits of RAdam over Adam may

diminish in settings where explicit learning rate warm-up is already carefully tuned.

Within the proposed methodology, RAdam is employed via the native implementation provided in the previously mentioned `torch.optim` module. The use of the built-in `RAdam` optimizer ensures an efficient and numerically stable realization of the algorithm, faithfully reproducing the rectification strategy described in the original work while seamlessly integrating into the overall training pipeline.

2.3.9 Evolved Sign Momentum

Evolved Sign Momentum (Lion) is a recently proposed optimization algorithm introduced by Chen et al. in 2023 [**19**]. The method was discovered via symbolic program search and is designed as a lightweight alternative to Adam-like optimizers: it is more memory-efficient because it only stores a momentum buffer, while parameter updates are computed through a *sign* operation, yielding uniform update magnitudes across coordinates.

Conceptually, Lion combines (i) a first-order exponential moving average (EMA) of gradients and (ii) sign-based descent. At each step, an interpolated gradient and momentum direction is formed and then reduced to its sign, so the update direction depends on the alignment of gradients over time rather than on adaptive per-parameter scaling. In the same spirit as AdamW, Lion is typically used with decoupled weight decay, applied directly to the parameters instead of being mixed into the gradient.

procedure LION(α, β_1, β_2, λ, $f(\theta)$, θ_0)
  $t \leftarrow 0$
  $m_0 \leftarrow 0$ ▷ Initialize momentum
  while not converged **do**
    $t \leftarrow t + 1$
    Sample a random data point x_t
    $g_t \leftarrow \nabla_\theta f(\theta_{t-1}; x_t)$
    $u_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ ▷ Update direction (interp)
    $\theta_{t-1} \leftarrow (1 - \alpha\lambda)\, \theta_{t-1}$ ▷ Decoupled weight decay
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \mathrm{sign}(u_t)$
    $m_t \leftarrow \beta_2 m_{t-1} + (1 - \beta_2) g_t$ ▷ Momentum update (interp)
  end while
  return θ_t
end procedure

A concise formulation of Lion can be observed in the previous page. Given a stochastic gradient g_t, Lion first computes an intermediate direction u_t as an interpolation between g_t and the previous momentum m_{t-1}, controlled by β_1. The parameters are then updated by stepping along $\text{sign}(u_t)$, and the momentum is refreshed using a second interpolation coefficient β_2.

Lion's main practical implications follow directly from this design. Since the update magnitude is set by the sign operator, Lion typically requires smaller learning rates than adaptive methods such as AdamW, and its performance gains are often more pronounced at large batch sizes. At the same time, the absence of second-moment tracking reduces memory overhead and can improve efficiency in large-scale training settings.

From an implementation perspective, `torch.optim` does not natively provide Lion; therefore, in this work the optimizer is implemented as a custom subclass. The implementation follows the commonly used Lion update pattern and applies decoupled weight decay multiplicatively to the parameters. In addition, it performs two EMA operations on the same momentum buffer within each step: the first EMA with β_1 is used to compute the sign update direction, while a second EMA with β_2 updates the stored momentum for the next iteration. This matches the reference pseudocode structure in the original paper, to ensure comparability in the proposed experiments.

2.3.10 Layer-wise Adaptive Moments for Batch Training

Layer-wise Adaptive Moments for Batch Training (LAMB) was proposed by You et al. in 2019 as an optimization algorithm specifically designed to maintain stable and efficient convergence in large-batch training regimes **[20]**. Motivated by the limitations observed when scaling adaptive methods such as Adam to very large batch sizes, LAMB introduces a layer-wise normalization mechanism that enables consistent optimization behavior across parameter groups, particularly in deep architectures employed for large-scale representation learning.

LAMB extends the Adam optimizer by retaining its bias-corrected first- and second-order moment estimates, while modifying the parameter update through a layer-wise adaptive scaling factor, commonly referred to as the *trust ratio*. Given an Adam-style update direction u_t, the parameters are updated according to

$$r_t = \frac{\|w_t\|_2}{\|u_t\|_2}, \qquad w_{t+1} = w_t - \alpha\, r_t\, u_t,$$

where w_t denotes the current parameter vector and α the learning rate. When either norm is ill-defined, the trust ratio is conventionally set to unity. In line with

modern best practices, LAMB is typically combined with decoupled weight decay, ensuring that regularization is applied independently of the gradient-based update and improving stability under large learning rates [**20**].

The introduction of the trust ratio constitutes the central contribution of LAMB. By explicitly relating the update magnitude to the scale of the parameters themselves, the algorithm mitigates the risk of disproportionately large or small updates across layers, a phenomenon that becomes increasingly problematic as batch size grows. This mechanism allows LAMB to preserve the favorable convergence properties of adaptive optimizers while enabling effective large-batch scaling without extensive retuning of hyperparameters. The principal advantages of LAMB therefore lie in its robustness and scalability in large-batch training scenarios, as well as its ability to harmonize update magnitudes across layers in deep networks. Nonetheless, the method introduces additional sensitivity to design choices related to the computation and regulation of the trust ratio. In practice, several implementations incorporate auxiliary heuristics, such as trust-ratio clipping, to prevent extreme scaling effects, potentially complicating direct comparisons across implementations.

From an implementation perspective, LAMB is not included among the standard optimizers provided by the `torch.optim` module. Consequently, the experimental results presented in this work rely on a custom PyTorch implementation that adheres to the core algorithmic principles described in the original formulation. Specifically, the implementation incorporates bias-corrected Adam moments, decoupled weight decay, and layer-wise trust-ratio scaling. Certain auxiliary features found in reference implementations, such as windowing strategies, were intentionally omitted in order to preserve conceptual clarity while maintaining alignment with the defining characteristics of LAMB.

2.3.11 Yogi

Yogi is an adaptive first-order optimization algorithm introduced by Zaheer et al. in 2018 [**21**] with the explicit objective of improving the robustness and theoretical behavior of adaptive gradient methods in stochastic and non-convex settings. The method was developed in response to certain limitations observed in Adam, particularly the tendency of its second-moment estimate to grow monotonically due to the cumulative effect of squared gradients. Such growth may lead to excessively small effective learning rates and, in some cases, suboptimal generalization performance. Yogi addresses this issue by introducing a controlled update mechanism for the variance estimate, thereby regulating its evolution over time.

The algorithm preserves the exponential moving average of first-order gradients,

similarly to Adam. Formally, the first-moment estimate is defined as

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \tag{2.8}$$

where $g_t = \nabla_\theta f(\theta_{t-1})$ denotes the stochastic gradient at iteration t, and $\beta_1 \in (0,1)$ governs the momentum decay. The distinctive contribution of Yogi lies in its second-moment update rule. Instead of accumulating squared gradients additively, Yogi employs a sign-adjusted correction:

$$v_t = v_{t-1} - (1 - \beta_2)\,\mathrm{sign}(v_{t-1} - g_t^2)\, g_t^2, \tag{2.9}$$

with $\beta_2 \in (0,1)$ controlling the decay rate. This formulation ensures that the second-moment estimate increases only when the instantaneous squared gradient exceeds the current estimate and decreases otherwise. Consequently, the variance term is prevented from growing unboundedly, enabling a more balanced and stable adaptation of the effective step sizes.

As in other adaptive moment methods, bias correction is applied to both moment estimates to counteract initialization effects:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}. \tag{2.10}$$

The parameter update is then computed as

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}, \tag{2.11}$$

where α denotes the learning rate and ϵ is a small positive constant introduced for numerical stability.

From a theoretical perspective, the controlled variance dynamics of Yogi contribute to improved convergence properties in stochastic non-convex optimization. By allowing the second-moment estimate to both increase and decrease in response to gradient behavior, the algorithm avoids the systematic overestimation of curvature that may occur in purely additive schemes. This property yields a more stable effective learning rate and can enhance generalization in large-scale deep learning tasks characterized by noisy or highly variable gradients.

The principal strengths of Yogi therefore stem from its self-regulating second-moment mechanism and its capacity to maintain adaptive learning rates without inducing excessive attenuation of parameter updates. Nevertheless, the method remains sensitive to hyperparameter selection, particularly with respect to β_1, β_2, and α, which must be carefully calibrated to the problem at hand. As with other

adaptive optimizers, inappropriate configurations may compromise convergence speed or final performance.

With regard to implementation, Yogi is not currently included in the standard `torch.optim` module. For this reason, a dedicated implementation was developed within the experimental framework of this work. The implementation adheres closely to the original formulation, incorporating bias-corrected first and second moments together with the sign-based variance update rule. The design choices ensure conceptual fidelity to the original algorithm while maintaining compatibility with the PyTorch-based training pipeline adopted throughout the experimental evaluation.

2.4 Sharpness-Aware Methods

While adaptive gradient methods primarily focus on rescaling parameter updates based on historical gradient statistics, a different line of research addresses another fundamental aspect of optimization in deep learning: the geometry of the loss landscape. In particular, recent studies have emphasized the relationship between the sharpness of local minima and the generalization performance of over-parameterized models. This observation has motivated the development of sharpness-aware optimization methods, which explicitly incorporate local curvature information into the training objective.

The central idea underlying these approaches is to favor parameter configurations that lie in flat regions of the loss surface, rather than merely minimizing the training loss at a single point. Instead of performing updates solely based on the gradient evaluated at the current parameters, sharpness-aware methods approximate a local worst-case perturbation within a predefined neighborhood and optimize the model with respect to this adversarially perturbed configuration. In doing so, they implicitly penalize sharp minima and encourage solutions that are more robust to small parameter perturbations.

As illustrated in **Figure 2.2**, two minima may exhibit comparable training loss values while differing significantly in local curvature. Solutions located in flatter regions tend to be more robust to parameter perturbations and are empirically associated with improved generalization performance. Among the most representative methods in this category are *Sharpness-Aware Minimization* (SAM) and its *generalized* extension (GSAM). The following subsections examine their theoretical foundations and practical implementation details.

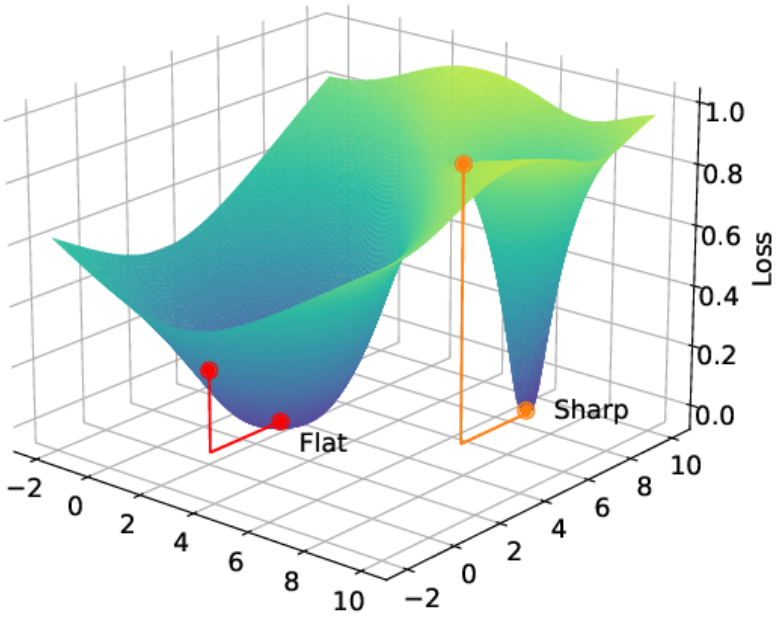

Figure 2.2: Conceptual illustration of sharp versus flat minima in a loss landscape

2.4.1 Sharpness-Aware Minimization

Sharpness-Aware Minimization (SAM) was introduced by Foret et al. in 2021 [**22**] as a general-purpose optimization framework explicitly designed to improve generalization by favoring flat minima. Unlike classical first-order methods, which minimize the loss evaluated at the current parameter vector, SAM reformulates training as a local min–max problem. The objective is not only to reduce the empirical risk, but also to ensure that the solution remains robust under small perturbations of the model parameters.

Formally, SAM considers the following optimization problem:

$$\min_{\theta} \max_{\|\epsilon\|_2 \leq \rho} \mathcal{L}(\theta + \epsilon), \tag{2.12}$$

where $\rho > 0$ defines the radius of a neighborhood around the current parameters. The inner maximization identifies the worst-case perturbation within an ℓ_2-ball, thereby measuring local sharpness, while the outer minimization updates the parameters so as to reduce this locally perturbed loss. In practice, the inner

problem is approximated using a first-order expansion, yielding a perturbation in the direction of the normalized gradient:

$$\epsilon_t = \rho \frac{g_t}{\|g_t\|_2}, \tag{2.13}$$

where $g_t = \nabla_\theta \mathcal{L}(\theta_t)$. The parameters are temporarily perturbed by ϵ_t, a new gradient is computed at $\theta_t + \epsilon_t$, and a standard optimizer step is then applied using this adversarially perturbed gradient.

Algorithmically, SAM can therefore be interpreted as a two-step procedure at each iteration: (i) ascent in parameter space along the normalized gradient to approximate the sharpest local direction, and (ii) descent using a base optimizer evaluated at the perturbed point. A concise description is reported below.

procedure SAM(α, ρ, base optimizer, $f(\theta)$, θ_0)
  $t \leftarrow 0$
  while not converged **do**
    $t \leftarrow t + 1$
    Sample a mini-batch x_t
    $g_t \leftarrow \nabla_\theta f(\theta_{t-1}; x_t)$
    $\epsilon_t \leftarrow \rho \frac{g_t}{\|g_t\|_2}$ ▷ Worst-case perturbation
    $\tilde{\theta} \leftarrow \theta_{t-1} + \epsilon_t$
    $\tilde{g}_t \leftarrow \nabla_\theta f(\tilde{\theta}; x_t)$
    $\theta_t \leftarrow \text{BaseOptimizerStep}(\theta_{t-1}, \tilde{g}_t)$
  end while
  return θ_t
end procedure

The practical implications of this formulation are twofold. First, SAM introduces an explicit bias toward flat regions of the loss landscape, which empirically improves generalization across a wide range of architectures and datasets. Second, the method requires two forward–backward passes per iteration, increasing computational cost compared to standard single-step optimizers. Nevertheless, SAM remains modular, as it can be combined with virtually any first-order method, such as SGD or AdamW.

From an implementation perspective, `torch.optim` does not natively provide SAM. In this work, the optimizer is implemented as a custom subclass that wraps a generic base optimizer. The implementation follows the canonical two-step strategy through a closure-based design: during the first step, the ℓ_2 norm of all gradients is computed globally, parameters are perturbed along the normalized gradient direction scaled by a single global radius ρ, and the original weights are temporarily

stored. During the second step, parameters are restored to their pre-perturbation values and the base optimizer performs its update using the gradient evaluated at the perturbed point. The adopted implementation adheres closely to the core algorithmic structure proposed in the original paper, while introducing controlled simplifications. This design ensures clarity, reproducibility, and comparability within the experimental framework developed in this thesis.

2.4.2 Gap Guided Sharpness-Aware Minimization

Gap-Guided Sharpness-Aware Minimization (GSAM), proposed by Zhuang et al. in 2022 **[23]**, extends the Sharpness-Aware Minimization (SAM) framework by explicitly addressing the trade-off between loss minimization and sharpness reduction. While SAM formulates training as a local min-max problem, GSAM observes that the gradient computed at perturbed parameters may not be well aligned with the descent direction that minimizes the empirical loss. This misalignment can lead to suboptimal updates and reduced training efficiency.

The central idea of GSAM is to decompose the gradient evaluated at the perturbed point into components that are respectively aligned and misaligned with the original gradient direction. By attenuating the component responsible for sharpness increase while preserving the descent-oriented component, GSAM improves the alignment between curvature control and optimization progress. Formally, the method introduces a correction term regulated by a hyperparameter γ, which balances the influence of sharpness-aware modification against standard gradient descent. This mechanism enables GSAM to reduce the generalization gap while maintaining stable convergence behavior.

From an algorithmic perspective, GSAM retains the two-step structure characteristic of SAM: (i) computation of a perturbation within a radius ρ, and (ii) parameter update using a modified gradient evaluated at the perturbed point. However, the effective update direction is adjusted through the gap-guided correction, ensuring that the optimization trajectory does not excessively prioritize flatness at the expense of descent efficiency.

It is worth noting that GSAM is not currently available in the `torch.optim` module. Consequently, within the scope of this thesis, GSAM has been implemented as a custom extension derived from the previously defined SAM optimizer. For reasons of computational efficiency and experimental coherence, a simplified approximation has been adopted in which the perturbation radius is scaled according to

$$\rho_{\text{eff}} = \rho(1 + \gamma)$$

This approximation captures the increased emphasis on flatness induced by GSAM through an amplified perturbation magnitude, although it does not reproduce the full gradient decomposition and alignment strategy described in the original formulation. Therefore, the adopted implementation should be interpreted as a controlled and computationally efficient approximation of GSAM: it preserves the fundamental intuition of gap-guided sharpness modulation and ensures methodological consistency within the experimental framework developed in this work.

In summary, the optimization strategies examined in this chapter reflect a systematic evolution of first-order methods along three principal axes: variance mitigation and acceleration through stochastic approximations and momentum mechanisms; parameter and layer-wise adaptivity aimed at stabilizing updates in heterogeneous and ill-conditioned settings; and, more recently, explicit incorporation of local loss geometry through sharpness-aware formulations. Each class of methods embodies distinct theoretical assumptions and practical trade-offs, particularly with respect to convergence behavior, computational complexity, memory requirements, and sensitivity to hyperparameter configuration. Within the scope of this thesis, these approaches are analyzed under a unified experimental framework to enable a principled and comparative evaluation.

Having established the theoretical foundations and clarified the algorithmic design choices adopted in the implementation, the next section situates the present study within the broader research landscape by reviewing the most relevant related work.

2.5 Research Related to the Studied Topic

Before introducing the methodological framework adopted in this thesis, it is appropriate to situate the present work within the broader landscape of existing research. This section provides a concise yet structured overview of the state of the art concerning the comparative analysis of optimization algorithms for deep learning. Given the central role of optimization in training neural networks, benchmarking different optimizers has become a topic of substantial interest. A large number of online technical articles and practitioner-oriented reports compare optimization methods across popular deep learning frameworks. Although such contributions often provide valuable practical insights for developers, they typically lack the methodological rigor, reproducibility standards, and theoretical grounding required for a formal academic literature review. For this reason, while these sources may serve as complementary material for implementation-oriented discussions, they are not considered authoritative references within the scientific context of this dissertation.

In contrast, academically grounded comparative studies on optimization algorithms are comparatively fewer but significantly more rigorous. Among the most relevant works is the comprehensive analysis by Sebastian Ruder [**24**], which systematically reviews gradient descent–based optimization methods. Ruder's contribution is particularly valuable for its clear exposition of the mathematical foundations underlying each algorithm, as well as its discussion of practical considerations affecting empirical performance.

Equally relevant is the dissertation by Thomas Frerix (TUM, 2022) [**25**], which offers an in-depth examination of numerical optimization techniques within modern deep learning pipelines. In addition, the study by Schmidt et al. [**26**] provides a rigorous empirical evaluation of different optimization methods under controlled experimental settings, with particular emphasis on reproducibility and fair comparison protocols. Their methodological framework, especially regarding standardized benchmarking procedures and more traditional optimizers selection, has informed the experimental design adopted in this thesis. Furthermore, the work of Thomas P. Minka [**27**] on numerical optimization for logistic regression is noteworthy. Although the focus of that contribution lies primarily on second-order methods, which are generally impractical in large-scale deep learning due to their computational cost, the methodological rigor and experimental protocol proposed therein have inspired aspects of the comparative strategy employed in this project.

Finally, no literature review on this topic would be complete without referencing the work of Dami Choi et al. [**28**], *On Empirical Comparisons of Optimizers for Deep Learning*. Their study constitutes one of the most comprehensive empirical benchmarks in the field, integrating both theoretical considerations and large-scale experimental evaluations. The questions raised and partial limitations identified in that work served as a primary motivation for the present dissertation, which aims to further investigate and clarify specific aspects of optimizer comparison within contemporary deep learning settings.

The present thesis differs from these prior works along three concrete dimensions. First, whereas Schmidt et al.[**26**] restrict their evaluation to a relatively small set of established optimizers and well-studied vision and language modeling tasks, this benchmark explicitly incorporates more recently proposed methods, including Lion, AdaBelief, Yogi, and sharpness-aware optimizers such as SAM and GSAM, whose behavior under controlled comparison conditions has received limited systematic attention. Second, the task coverage of this work is substantially broader, spanning eight distinct domains from pixel-level segmentation to abstractive summarization, enabling cross-domain conclusions that single-domain or narrow benchmarks cannot

support. Third, and in direct response to the methodological concerns raised by Choi et al. [**28**], the experimental protocol adopted here evaluates optimizers under their publication-recommended default configurations rather than independently tuned hyperparameters, deliberately benchmarking the out-of-the-box behavior that practitioners most commonly encounter. Together, these design choices position the present study as a complement to existing benchmarks rather than a replication, extending the comparative landscape to newer methods, broader tasks, and more ecologically valid experimental conditions.

Chapter 3

Methodology

Having established the theoretical foundations of numerical optimization methods in the previous chapter, the present chapter defines the experimental methodology adopted to conduct the empirical benchmark. While Chapter 2 examined the mathematical principles and algorithmic structures underlying the optimizers considered in this study, the objective here is to describe how these methods are evaluated under a unified and reproducible framework across heterogeneous deep learning scenarios.

The reliability, validity, and interpretability of any comparative analysis in optimization are inherently dependent on the rigor of its experimental design. Variations in dataset preprocessing, model architecture, metric definition, training schedules, and hyperparameter selection may substantially influence observed performance, potentially obscuring the intrinsic characteristics of the optimization algorithms themselves. To mitigate such confounding factors, the benchmark developed in this thesis is structured around a controlled and unified *task* - *dataset* - *model* - *metric* framework. This design ensures methodological consistency across experiments while preserving sufficient diversity to assess optimizer robustness under substantially different learning paradigms.

The experimental campaign encompasses eight distinct tasks spanning both computer vision and natural language processing domains. The inclusion of this heterogeneous set of problems is deliberately intended to avoid domain-specific bias and to evaluate whether optimizer performance generalizes across fundamentally different objective functions, output structures, and data modalities. By situating the analysis within multiple application contexts, the study aims to derive conclusions that are not restricted to a single class of learning problems.

The selection of tasks, datasets, and model architectures is guided by three main

criteria: relevance in the literature, diversity in scale and difficulty, and compatibility with a constrained computational budget. Accordingly, each task is associated with two datasets and two model architectures, typically combining one lightweight model implemented from scratch with one more structured or pre-trained architecture commonly adopted in contemporary research. This design balances empirical breadth with methodological control and enables the investigation of how optimization dynamics interact with model capacity, architectural complexity, and parameterization.

Evaluation is conducted through task-specific primary metrics, complemented by secondary indicators where appropriate. Beyond predictive performance, particular attention is devoted to convergence behavior, training and validation loss trajectories, runtime efficiency, and stability across epochs. By integrating both performance-oriented and computational criteria, the study provides a multidimensional characterization of each optimizer, thereby supporting a more comprehensive comparative analysis.

The structure of this chapter reflects the organization of the experimental framework. The following sections describe the benchmark design for each selected task, including the chosen datasets, model architectures, evaluation metrics, and the main preprocessing and implementation choices relevant to fair comparison. Special emphasis is placed on maintaining consistent training protocols across optimizers to ensure fairness, reproducibility, and methodological transparency.

The final section of this chapter presents the overarching experimental methodology, including the hardware environment, training configuration, optimizer selection strategy, and reproducibility considerations that govern the entire benchmark. This methodological synthesis establishes the foundation upon which the empirical results discussed in the subsequent chapter are interpreted. Through this structured and transparent design, the present chapter bridges the theoretical analysis of optimization methods with their practical evaluation, thereby defining the operational context required for a rigorous and meaningful comparative study.

3.1 Image Classification

Image classification is included in the benchmark as a controlled computer vision setting for evaluating optimization stability, convergence dynamics, and generalization performance. Its standardized formulation and broad use in the literature make it a suitable testbed for comparing first-order optimization algorithms under comparable conditions.

Let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ denote a dataset of labeled images, where $x_i \in \mathbb{R}^{C\times H\times W}$ represents an input image with C channels and spatial resolution $H \times W$, and $y_i \in \{1, \ldots, K\}$ denotes the corresponding class label among K possible categories. The objective of the learning process is to determine parameters θ of a model f_θ that minimize the empirical risk

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \ell\Big(f_\theta(x_i), y_i\Big),$$

where ℓ corresponds to the multiclass cross-entropy loss. Within the implemented framework, the model outputs logits $z = f_\theta(x) \in \mathbb{R}^K$, and predictions are obtained through the decision rule

$$\hat{y} = \arg\max_{k\in\{1,\ldots,K\}} z_k.$$

From the perspective of the benchmark, image classification provides a representative non-convex learning problem in which preprocessing, normalization, and batch size selection directly affect gradient distributions and training stability. It therefore serves as a controlled setting for comparing optimizer behavior under realistic deep learning conditions.

3.1.1 MNIST Dataset

MNIST is a grayscale handwritten digit dataset composed of 28×28 single-channel images and $K = 10$ classes. Originally introduced for document recognition research, it remains a standard reference benchmark in representation learning [**29**]. The dataset is partitioned into 60,000 training samples and 10,000 test samples; in the present experimental protocol, the test split is employed as validation.

The preprocessing pipeline includes random rotations within $\pm 10^\circ$ applied during training, followed by tensor conversion and normalization using dataset-specific statistics. The validation split undergoes normalization without augmentation. The introduction of controlled geometric variability enhances invariance to minor rotations while preserving semantic consistency.

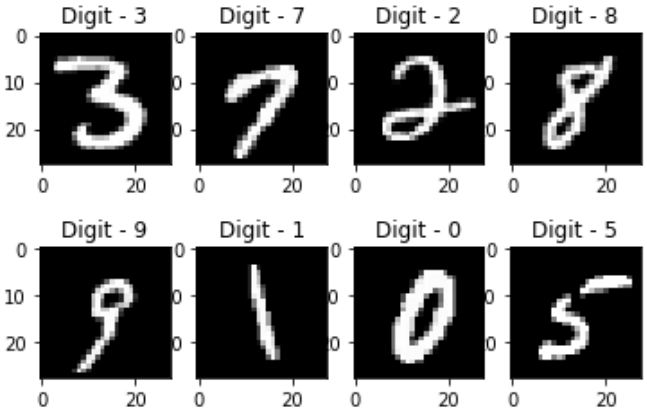

Figure 3.1: Samples from the MNIST dataset

Due to its reduced dimensionality and relatively simple structure, MNIST is computationally efficient and typically exhibits rapid convergence. This makes it particularly suitable for controlled comparisons of optimizer stability and early training behavior. However, its low intrinsic complexity may lead to early saturation of training accuracy, especially in small-sample regimes. In such scenarios, insufficient normalization or augmentation may obscure meaningful differences among optimization strategies. Furthermore, the single-channel nature of the dataset necessitates architectural adaptation in models originally designed for RGB inputs.

3.1.2 CIFAR-10 Dataset

CIFAR-10 consists of 32×32 RGB images distributed across 10 semantic classes and was introduced as a benchmark for small-scale object recognition [**30**]. The dataset contains 50,000 training samples and 10,000 test samples, the latter used here as validation.

The training preprocessing pipeline incorporates random cropping with four-pixel padding and random horizontal flipping, followed by channel-wise normalization. The validation split is normalized without augmentation. Proper normalization is essential for stabilizing optimization dynamics, as inappropriate input scaling may significantly affect gradient magnitudes and convergence behavior. Data augmentation introduces invariances that act as implicit regularization, mitigating overfitting and increasing the discriminative power of optimizer comparisons.

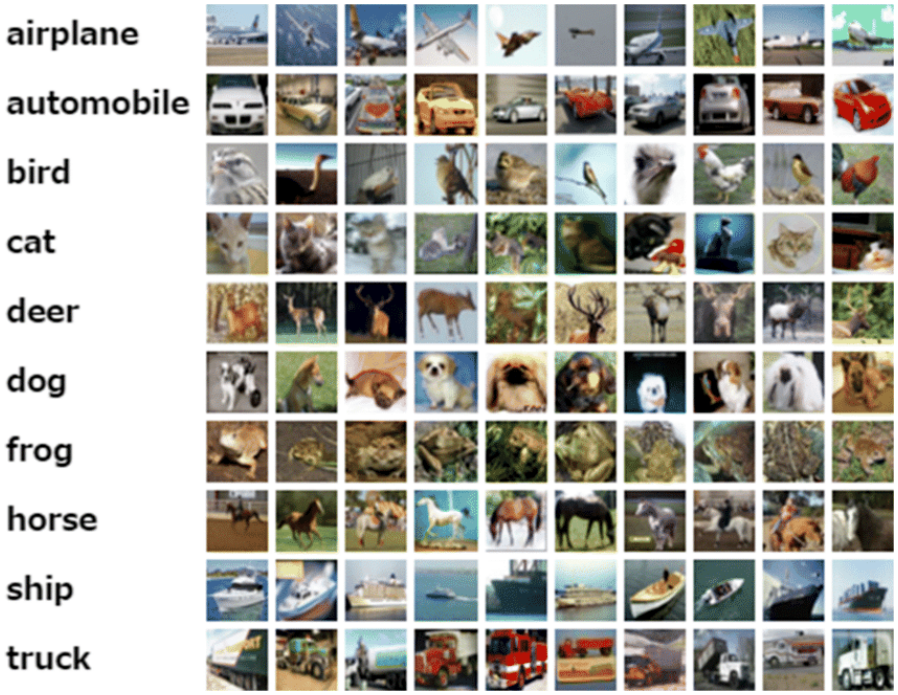

Figure 3.2: Samples from the CIFAR-10 dataset

Compared to MNIST, CIFAR-10 presents a higher degree of visual variability and structural complexity, thereby inducing a more intricate loss landscape and slower convergence. The combined use of both datasets ensures heterogeneity in input modality and task difficulty.

3.1.3 CNN Model

The `CNN` model is a compact convolutional neural network trained from random initialization. Architecturally, it comprises stacked convolution–ReLU–pooling blocks followed by a fully connected classification head:

$$\begin{aligned} &\text{Conv} \rightarrow \text{ReLU} \rightarrow \text{MaxPool} \\ &\quad \rightarrow \text{Conv} \rightarrow \text{ReLU} \rightarrow \text{MaxPool} \\ &\quad \rightarrow \text{Flatten} \rightarrow \text{Linear} \rightarrow \text{ReLU} \\ &\quad \rightarrow \text{Linear} \end{aligned} \tag{3.1}$$

The network contains approximately 1–2 million parameters, depending on the input channel configuration. The final classification layer is automatically adapted to the number of classes inferred from the dataset.

This architecture deliberately excludes Batch Normalization layers and pretraining, thereby increasing sensitivity to hyperparameters such as learning rate and weight decay. As a result, it provides a suitable testbed for highlighting optimizer-dependent differences in convergence and stability. Owing to its limited depth, the architecture is especially useful for isolating the effects of update dynamics and implicit regularization.

3.1.4 ResNet18 Model

The second model is a randomly initialized ResNet-18, a deep residual architecture originally proposed by He et al. [**31**]. The network is composed of residual blocks with identity skip connections and Batch Normalization layers, and contains approximately 11 million parameters.

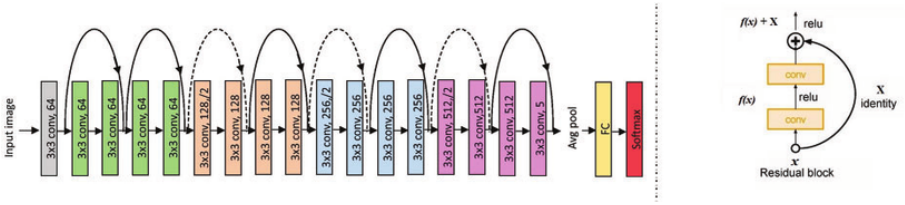

Figure 3.3: Schematic representation of the ResNet-18 architecture

To ensure compatibility with small-resolution inputs and non-RGB datasets, the initial convolutional layer is replaced by a 3×3 convolution with stride 1 and padding 1 whenever the number of input channels differs from three. This modification preserves spatial resolution in early layers and aligns the architecture with CIFAR-style configurations.

The contrast between the shallow `CNN` architecture and the deeper residual `ResNet18` enables the evaluation of optimization methods across substantially different model capacities and conditioning properties, thereby strengthening the robustness of the comparative analysis.

3.1.5 Accuracy and Task-Specific Metrics

The evaluation protocol focuses on predictive performance indicators that quantify both global correctness and error distribution across classes. This combination ensures a rigorous and multidimensional assessment of model behavior.

Accuracy is defined as the proportion of correctly classified samples over the total number of evaluated instances. Let

$$\delta(\hat{y}_i, y_i) = \begin{cases} 1 & \text{if } \hat{y}_i = y_i, \\ 0 & \text{otherwise,} \end{cases}$$

then accuracy is computed as

$$\text{Accuracy} = \frac{1}{N} \sum_{i=1}^{N} \delta(\hat{y}_i, y_i).$$

Accuracy constitutes the primary evaluation metric due to its interpretability and direct relationship with empirical risk minimization under 0–1 loss.

In addition, metrics derived from true positives, false positives, and false negatives are computed, in particular *precision*, *recall*, and *F1 score*. In the micro-averaged formulation, these quantities are aggregated globally across all classes prior to ratio computation, thereby assigning proportional weight to each sample.

Within the broader experimental framework, a cosine learning-rate scheduler is adopted by default, introducing a smooth decay of the step size across epochs. The validity of optimizer comparisons in this task depends critically on controlling input normalization, data augmentation, and batch size selection so that observed differences can be attributed primarily to optimization behavior rather than to inconsistent experimental conditions. Insufficient control of these factors may lead to unstable dynamics or premature saturation, potentially confounding the interpretation of optimization performance.

3.2 Semantic Segmentation

Semantic segmentation is included in the benchmark as a dense prediction task in which a semantic label is assigned to each pixel of an input image. Compared to image classification, it introduces a substantially more demanding optimization setting due to the high-dimensional structured output space.

Let $\mathcal{D} = \{(x_i, Y_i)\}_{i=1}^{N}$ denote a dataset of images $x_i \in \mathbb{R}^{C \times H \times W}$ and corresponding segmentation masks $Y_i \in \{0, \ldots, K-1\}^{H \times W} \cup \{255\}$, where K represents the number of semantic classes and 255 denotes an ignore index. A parametric model f_θ is trained to produce dense logits

$$f_\theta : \mathbb{R}^{C \times H \times W} \to \mathbb{R}^{K \times H \times W},$$

so that each spatial location (h, w) is associated with a class score vector. The empirical objective aggregates pixel-wise losses while excluding ambiguous or unlabeled pixels:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{|\Omega_i|} \sum_{(h,w)\in\Omega_i} \ell\Big(f_\theta(x_i)_{:,h,w}, Y_i(h,w)\Big),$$

where $\Omega_i = \{(h, w) \mid Y_i(h, w) \neq 255\}$.

From the perspective of optimizer benchmarking, semantic segmentation constitutes a particularly demanding learning problem. The dimensionality of the output space scales with the number of pixels, and the loss aggregates thousands of local classification terms per image. Spatial correlations and class imbalance induce highly structured gradient distributions, making convergence especially sensitive to learning-rate scheduling, normalization, and batch statistics. Dense prediction tasks therefore provide a stringent setting for assessing optimizer stability and robustness.

3.2.1 Oxford-IIIT Pet Dataset

The Oxford-IIIT Pet dataset, introduced by Parkhi et al. **[32]**, is employed in its segmentation configuration. The dataset comprises approximately 3,700 training images and 3,700 validation images, each accompanied by a trimap annotation identifying pet, outline, and background regions.

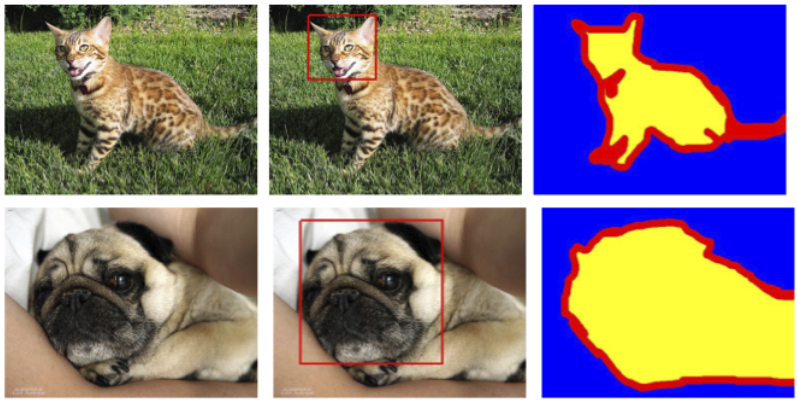

Figure 3.4: Example images and corresponding segmentation masks from the Oxford-IIIT Pet dataset.

Within the implemented preprocessing pipeline, trimaps are converted into semantic masks and subsequently remapped to a binary formulation. Specifically, foreground (pet) and background are retained as valid classes, whereas the thin outline region is assigned the ignore index 255. This design choice reduces label ambiguity and avoids disproportionately weighting narrow boundary regions that may introduce high-frequency noise into the optimization process, especially in limited-data regimes.

All images are resized to 256×256 using bilinear interpolation and normalized according to ImageNet channel statistics. Segmentation masks are resized using nearest-neighbor interpolation in order to preserve discrete class indices and prevent interpolation artifacts. The adoption of ImageNet normalization ensures compatibility with architectures originally developed for large-scale visual recognition and contributes to consistent gradient scaling across experiments.

Despite the reduced binary structure, the dataset remains non-trivial due to the predominance of background pixels. Such imbalance may bias optimization toward degenerate solutions that favor the majority class. To mitigate this effect, dynamically computed class-balanced weights are incorporated into the cross-entropy term during training, thereby rebalancing gradient contributions at batch level and promoting more stable learning dynamics.

3.2.2 Pascal VOC2012 Dataset

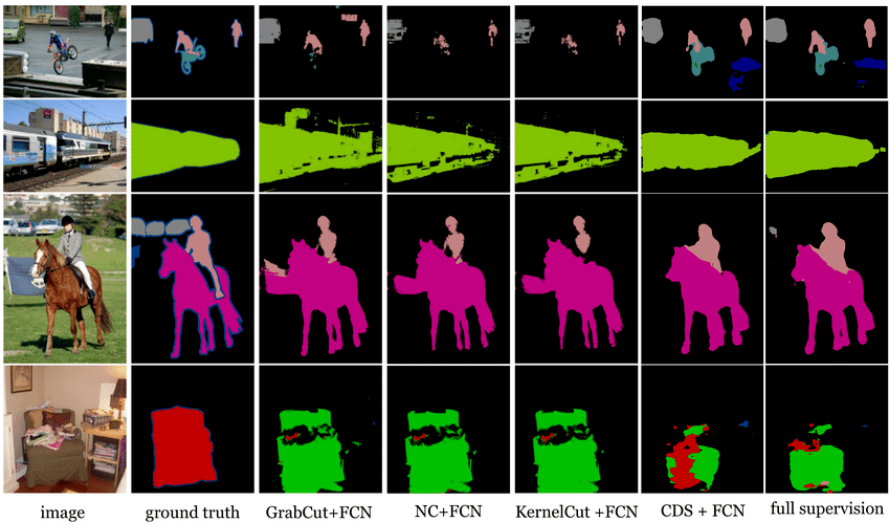

Figure 3.5: Representative segmentation examples from the Pascal VOC2012 dataset.

The Pascal VOC2012 segmentation benchmark, comprehensively reviewed by Everingham et al. [**33**], represents one of the most influential reference datasets for semantic segmentation. It provides dense annotations for 21 semantic categories, including background, and defines official train and validation splits that are adopted in the present framework. Segmentation masks encode class indices in $\{0, \ldots, 20\}$, with 255 reserved for ambiguous pixels excluded from loss computation. Images are resized to 256×256, converted to tensors, and normalized using ImageNet statistics. As in the previous dataset, masks are resized using nearest-neighbor interpolation to preserve label integrity. Automatic dataset download and directory organization ensure reproducibility and consistency of the experimental setup.

Compared to the binary Oxford-IIIT Pet configuration, Pascal VOC2012 introduces substantially greater semantic diversity and more intricate inter-class relationships. The multi-class setting increases gradient variance and results in a more complex loss landscape, thereby accentuating differences in optimizer behavior. Its established role as a canonical segmentation benchmark further enhances the interpretability and external validity of the empirical analysis.

3.2.3 DeepLabV3 with ResNet-50 Backbone Model

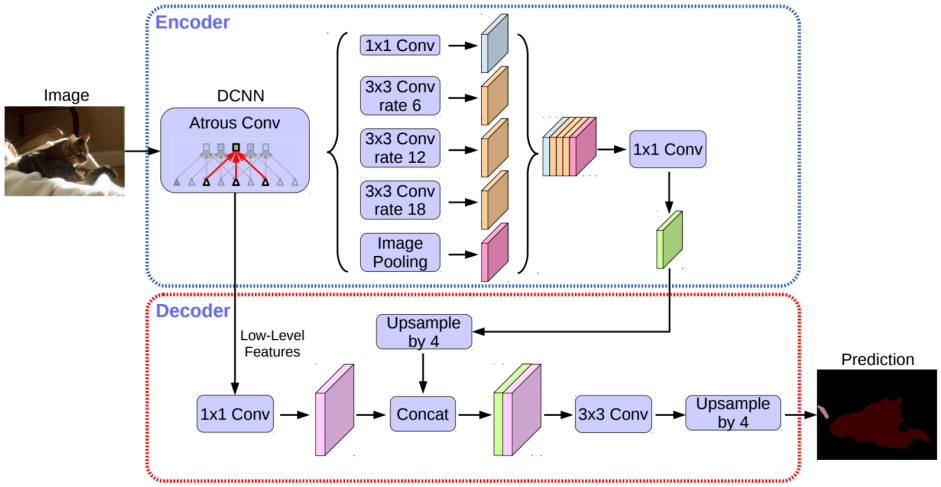

Figure 3.6: Schematic representation of the DeepLabV3 architecture with ASPP module.

The first model considered is DeepLabV3 with a ResNet-50 backbone, proposed by Chen et al. [**34**] in 2017. The architecture integrates deep residual feature extraction with an Atrous Spatial Pyramid Pooling (ASPP) module to capture

multi-scale contextual information while maintaining adequate spatial resolution.

The number of output channels is dynamically inferred from the dataset, either through an explicit `num_classes` attribute or via inspection of mask values excluding the ignore index. Residual connections facilitate gradient propagation and improve conditioning in deep architectures, while atrous convolutions enlarge the receptive field without excessive downsampling, thereby enhancing boundary delineation.

In the present framework, the model is trained from random initialization in order to preserve comparability with the other architectures included in the benchmark.

3.2.4 Vanilla U-Net Model

The second architecture is a lightweight U-Net [**35**], introduced by Ronneberger et al. in 2015. U-Net follows a symmetric encoder–decoder structure with skip connections linking corresponding resolution levels. Convolutional blocks are organized along a contracting path that captures contextual information, followed by an expanding path that restores spatial resolution through transposed convolutions and feature concatenation.

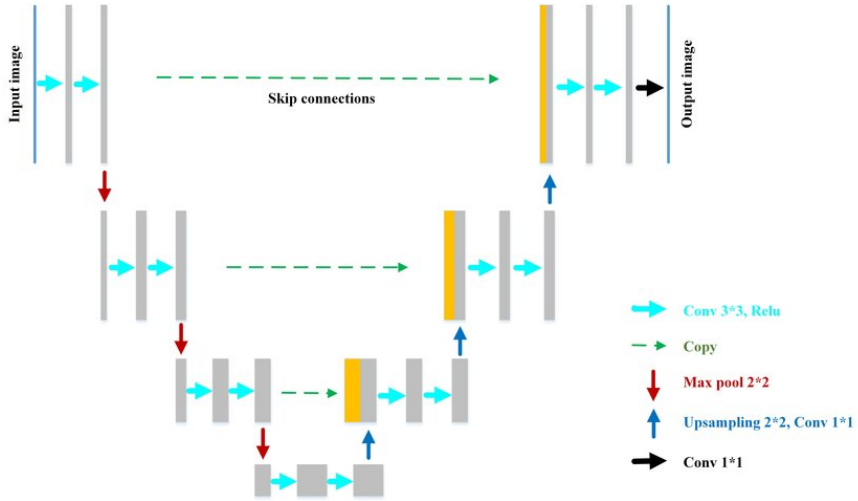

Figure 3.7: Encoder–decoder structure of the U-Net architecture with skip connections.

The final 1×1 convolution maps feature representations to K output channels inferred from the dataset. In contrast to DeepLabV3, this implementation does not include Batch Normalization layers, thereby increasing sensitivity to learning-rate

magnitude and batch size. Such sensitivity makes the architecture particularly suitable for highlighting optimizer-dependent regularization effects.

3.2.5 Dice and Task-Specific Metrics

Model predictions are obtained by assigning to each spatial location the class index associated with the maximum logit:

$$\hat{Y}(h, w) = \arg \max_{c \in \{0,\dots,K-1\}} f_\theta(x)_{c,h,w}.$$

All reported metrics are computed after excluding pixels labeled with the ignore index 255, consistently with the training objective. In practice, evaluation proceeds by accumulating a confusion matrix $C \in \mathbb{N}^{K \times K}$ over the validation set, where $C_{i,j}$ counts the number of pixels whose ground-truth class is i and whose predicted class is j. In addition, the number of correctly classified pixels and the total number of valid pixels are accumulated to compute pixel-level accuracy. To ensure robustness, any out-of-range predictions or labels (with respect to $\{0, \dots, K-1\}$) are excluded from the confusion-matrix update.

The first reported indicator is the *pixel accuracy*, defined as the fraction of correctly classified pixels among those not ignored:

$$\text{PixelAcc} = \frac{\sum_{c=0}^{K-1} C_{c,c}}{\sum_{i=0}^{K-1} \sum_{j=0}^{K-1} C_{i,j}}.$$

While pixel accuracy is intuitive, it can be dominated by majority classes in imbalanced scenarios. For this reason, the evaluation also reports the *mean pixel accuracy*, i.e., the mean of per-class accuracies, where for each class c the accuracy is computed as $C_{c,c} / \sum_j C_{c,j}$, and then averaged across classes.

The primary evaluation criterion adopted in this task is the *mean Dice coefficient*, computed from the confusion matrix as an average of class-wise Dice scores. For each class c, the Dice coefficient is defined as

$$\text{Dice}_c = \frac{2C_{c,c}}{\sum_{j=0}^{K-1} C_{c,j} + \sum_{i=0}^{K-1} C_{i,c} + \varepsilon}, \qquad \text{Dice} = \frac{1}{K} \sum_{c=0}^{K-1} \text{Dice}_c,$$

where ε is a small constant to avoid numerical instabilities. This formulation directly measures overlap between predicted and ground-truth regions by balancing false positives and false negatives, and is therefore particularly suitable in the presence of class imbalance and small foreground regions.

3.3 Sentiment Analysis

Sentiment analysis is included in the benchmark as a text classification task that allows optimizer behavior to be studied in both recurrent and transformer-based architectures. In its binary formulation, the task consists of assigning to a sentence or document a label $y \in \{0,1\}$, typically corresponding to negative and positive sentiment, respectively. Unlike vision-based classification, textual inputs are inherently sequential and variable in length, requiring tokenization, numerical encoding, and explicit handling of padding and truncation.

Formally, let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ denote a dataset of text samples x_i, where each x_i is a finite sequence of tokens drawn from a vocabulary $\mathcal{V}$, and $y_i \in \{0,1\}$. A parametric model f_θ defines a mapping

$$f_\theta : \mathcal{V}^{\leq L} \to \mathbb{R}^2,$$

where L denotes a maximum sequence length imposed for computational tractability. The model outputs logits $z_i = f_\theta(x_i)$, and predictions are obtained as $\hat{y}_i = \arg\max_c z_{i,c}$. The empirical objective corresponds to the average cross-entropy loss, optionally incorporating label smoothing.

From the perspective of optimizer benchmarking, sentiment analysis is informative because tokenized inputs induce highly non-uniform gradient statistics, sequence truncation affects gradient propagation, and recurrent and transformer-based architectures exhibit markedly different conditioning properties. These factors make the task suitable for comparing optimizer robustness across heterogeneous NLP model families.

3.3.1 IMDB Dataset

The first dataset employed is the IMDB movie review corpus, originally introduced by Maas et al. [**36**] and accessed in this framework via the HuggingFace `imdb` interface. It consists of 25,000 training samples and 25,000 test samples, each labeled with binary sentiment. Reviews are typically long-form documents, often spanning several hundred tokens, which makes the dataset particularly suitable for analyzing truncation effects and long-range dependency modeling.

Within the implemented pipeline, each sample is retrieved as a raw text string paired with an integer label. Two alternative preprocessing paths are supported. In the recurrent setting, tokenization is performed via simple whitespace splitting, and tokens are mapped to integer identifiers through a vocabulary constructed from a subset of the training data. Sequences are truncated to a fixed maximum

length of $L = 128$ tokens and padded with a dedicated `<pad>` symbol. In the transformer-based configuration, a pretrained tokenizer performs subword segmentation, padding, and truncation to the same maximum length.

The choice of IMDB is motivated by its long textual sequences, which amplify the interaction between tokenization strategy, sequence truncation, and optimization dynamics. Very long reviews may contain sentiment cues located far from the beginning of the text; truncation at 128 tokens may therefore discard informative content.

3.3.2 SST-2 Dataset

The second dataset is the Stanford Sentiment Treebank binary subset (SST-2), introduced by Socher et al. [**37**] and accessed through the HuggingFace `glue` interface with configuration `sst2`. It provides approximately 67,000 training samples and 1,800 validation samples, each consisting of a short sentence annotated with a binary sentiment label.

Compared to IMDB, SST-2 contains substantially shorter inputs, typically limited to a single sentence. This structural difference results in reduced truncation effects and faster iteration times, making SST-2 particularly suitable for rapid benchmarking under constrained computational budgets. The preprocessing strategy mirrors that of IMDB: either whitespace-based tokenization with vocabulary lookup for recurrent models, or subword tokenization via a pretrained transformer tokenizer.

3.3.3 LSTM-Based Sentiment Classifier

The first architecture considered is a recurrent neural network based on the Long Short-Term Memory (LSTM) formulation proposed by Hochreiter and Schmidhuber [**38**]. LSTMs were originally designed to mitigate vanishing and exploding gradient phenomena in recurrent networks by introducing gated memory cells, thereby enabling more stable learning over long sequences.

The implemented model comprises an embedding layer of dimension $d_e = 128$, a single-layer LSTM with hidden dimension $d_h = 128$, a dropout layer with probability $p = 0.5$, and a final linear classifier mapping the last hidden state to two output logits.

Given an input sequence of token identifiers $x = (x_1, \ldots, x_T)$, the embedding layer produces vectors $e_t \in \mathbb{R}^{d_e}$. The LSTM processes the packed sequence to accommodate variable lengths, and the final hidden state h_T is extracted. After

dropout regularization, a linear transformation yields the logits:

$$z = Wh_T + b.$$

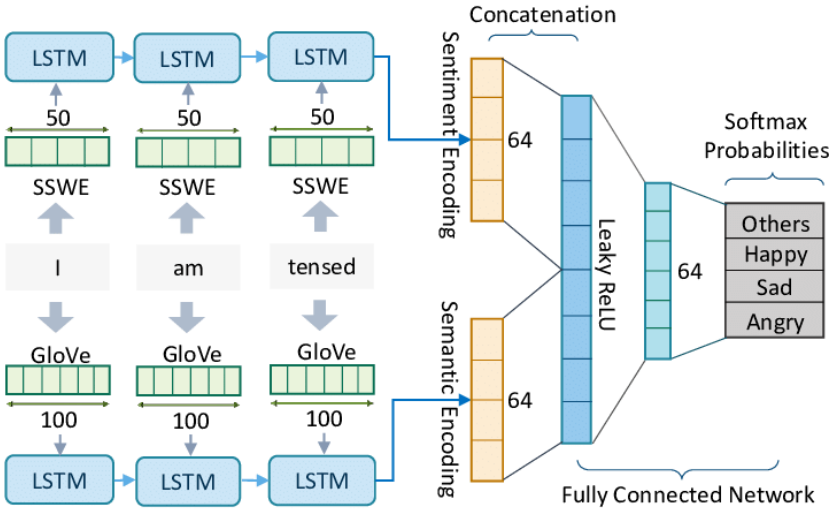

Figure 3.8: Schematic representation of the LSTM-based sentiment classifier

The vocabulary is constructed from up to 10,000 training samples, with reserved indices for padding and unknown tokens.

3.3.4 DistilBERT-Based Sentiment Classifier

The second model is a transformer-based classifier built upon the DistilBERT architecture proposed by Sanh et al. [**39**], itself derived from the BERT model introduced by Devlin et al. [**40**]. DistilBERT retains the bidirectional self-attention mechanism of BERT while reducing model size through knowledge distillation, resulting in approximately 66 million parameters.

Input texts are tokenized into subword units, augmented with attention masks, and truncated or padded to length 128. The transformer encoder produces contextualized representations, and the representation associated with the special classification token is fed to a linear head producing logits in $\mathbb{R}^2$.

The implementation attempts to load pretrained weights, falling back to locally cached versions or random initialization if necessary. In the fine-tuning scenario, optimization typically requires smaller learning rates and may benefit from weight decay regularization in an AdamW-style formulation.

3.3.5 Accuracy and Task-Specific Metrics

Evaluation metrics are computed through an accumulator that aggregates counts of correct predictions, total samples, true positives, false positives, and false negatives. Given predicted labels $\hat{y}_i$ and ground-truth labels y_i, the primary metric is *accuracy*, defined as

$$\text{Accuracy} = \frac{\sum_{i=1}^{N} \mathbf{1}[\hat{y}_i = y_i]}{N}.$$

In addition to accuracy, binary classification indicators such as precision, recall, and F1 score are computed, following the same counting formulation adopted for the image classification task. Therefore, accuracy is designated as the primary selection criterion. Precision, recall, and F1 score provide complementary information in scenarios where class imbalance or asymmetric misclassification costs are relevant. The reported indicators are strictly aligned with the implemented metric computation and exclude loss-based or convergence-related quantities, which are treated uniformly across tasks.

3.4 Machine Translation

Machine translation is included in the benchmark as a sequence-to-sequence task that exposes optimizer behavior in long-range structured prediction settings with large output spaces. Unlike fixed-dimensional classification tasks, both input and output are variable-length token sequences, and the prediction space grows combinatorially with sequence length and vocabulary size.

Formally, let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ denote a parallel corpus, where $x_i = (x_{i,1}, \ldots, x_{i,S_i})$ is a source-language sequence and $y_i = (y_{i,1}, \ldots, y_{i,T_i})$ is the corresponding target-language sequence. After tokenization and numerical encoding through vocabularies $\mathcal{V}_s$ and $\mathcal{V}_t$, a parametric model f_θ implements a mapping

$$f_\theta : \mathcal{V}_s^{\leq S} \times \mathcal{V}_t^{\leq T} \to \mathbb{R}^{T \times |\mathcal{V}_t|},$$

producing, at each decoding step, logits over the target vocabulary. Training is performed via teacher forcing: the decoder receives the shifted ground-truth sequence $y_{i,1:T-1}$ and is trained to predict $y_{i,2:T}$. The empirical objective corresponds to token-level cross-entropy with padding ignored and label smoothing coefficient $\epsilon = 0.1$ when supported by the backend implementation. Label smoothing acts as a regularizer that mitigates overconfidence and can stabilize optimization in sequence generation tasks.

From the perspective of optimizer benchmarking, machine translation presents large vocabularies that induce sparse and heavy-tailed gradient distributions in

embedding layers. Together with the structural differences between the selected sequence-to-sequence architectures, this creates a demanding and heterogeneous environment for comparative evaluation of first-order optimization algorithms.

3.4.1 Europarl Bilingual (en–de)

The first dataset employed is the Europarl bilingual English–German corpus, accessed through the HuggingFace `europarl_bilingual` interface with configuration `de-en`. The dataset consists of aligned parliamentary proceedings translated between English and German, providing relatively clean and well-structured parallel sentences.

In the implemented pipeline, tokenization is performed through lowercasing and whitespace splitting. A vocabulary is constructed from a subset of up to 10,000 training samples using frequency-based truncation, with the final vocabulary capped at 30,000 entries. Reserved symbols include `<pad>`, `<unk>`, `<bos>`, and `<eos>`. Sequences are encoded by prefixing `<bos>`, truncating to a maximum length of 64 tokens, and appending `<eos>`. Padding to the maximum batch length is applied dynamically during collation.

The choice of Europarl is motivated by its linguistic consistency and relatively low noise, which reduce confounding variability and facilitate clearer attribution of performance differences to optimization dynamics.

3.4.2 IWSLT14 en–de (Opus Books Proxy)

The second dataset corresponds to a lightweight English–German parallel corpus loaded via the HuggingFace `opus_books` interface, used as a proxy for small-scale IWSLT-style benchmarks. Compared to Europarl, this corpus is smaller and exhibits different domain characteristics.

The preprocessing strategy mirrors that of Europarl: whitespace tokenization, vocabulary construction with maximum size 40,000, insertion of boundary tokens, truncation at length 64, and dynamic padding. If no explicit validation split is available, a portion of the training data is held out to serve as validation.

The reduced scale of this dataset enables rapid experimentation under constrained computational budgets. However, domain differences relative to larger corpora can affect lexical diversity and n-gram overlap statistics, thereby influencing evaluation metrics.

3.4.3 LSTM-Based Sequence-to-Sequence Model

The first architecture considered is a recurrent encoder–decoder model based on Long Short-Term Memory (LSTM) units, following the general paradigm introduced in [**41**] and extended with attention mechanisms as in [**42**]. The encoder consists of a bidirectional LSTM with embedding dimension 256, hidden dimension 256, a single recurrent layer, and dropout probability 0.1. The decoder is a unidirectional LSTM with dot-product attention over encoder outputs.

Let $h_S^{\rightarrow}$ and $h_1^{\leftarrow}$ denote the final forward and backward encoder states. These are concatenated and linearly projected to initialize the decoder hidden and cell states through learned bridge transformations. At each decoding step, attention weights are computed via a masked dot-product between the decoder query and encoder keys, followed by softmax normalization. The resulting context vector is concatenated with the embedded input token and processed by the decoder LSTM. Output logits are produced through a linear projection to the target vocabulary.

3.4.4 Transformer-Based Sequence-to-Sequence Model

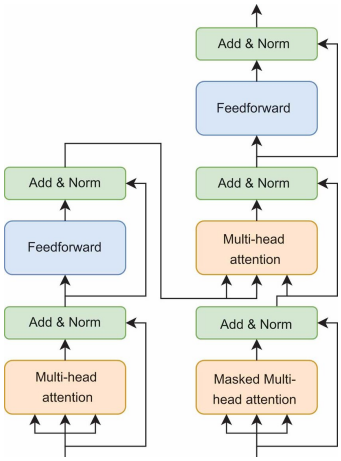

Figure 3.9: Transformer encoder-decoder architecture

The second architecture is a transformer encoder–decoder model implemented via `nn.Transformer`, directly inspired by the architecture proposed in [**43**]. Both source and target tokens are embedded into a $d_{\text{model}} = 256$ dimensional space and augmented with sinusoidal positional encodings. The encoder and decoder each comprise two stacked layers with multi-head self-attention ($n_{\text{head}} = 4$) and position-wise feed-forward networks.

Given embedded source sequence X and target prefix Y, the transformer computes contextualized representations through stacked self-attention in the encoder and masked self-attention combined with cross-attention in the decoder. The final decoder representations are projected to vocabulary logits through a linear layer.

3.4.5 BLEU and Task-Specific Metrics

Evaluation metrics are computed by aggregating predicted and reference token identifiers and subsequently detokenizing them by removing special symbols (`<pad>`, `<bos>`, `<eos>`). The primary metric is BLEU (Bilingual Evaluation Understudy), originally introduced by Papineni et al. [**44**]. BLEU measures modified n-gram precision combined with a brevity penalty to discourage excessively short translations.

For a candidate sequence c and reference sequence r, the modified n-gram precision for $n \in \{1, \ldots, 4\}$ is computed as

$$p_n = \frac{\text{match}_n + 1}{\text{total}_n + 1},$$

where add-one smoothing is applied to avoid numerical instability. BLEU is then defined as

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{4} \frac{1}{4} \log p_n\right),$$

where the brevity penalty $\text{BP} = \exp\left(1 - \frac{|r|}{|c|}\right)$ if $|c| < |r|$ and 1 otherwise. Reported BLEU corresponds to the score computed over the full evaluation corpus.

In addition to BLEU, ROUGE-L is computed via the longest common subsequence between candidate and reference sequences, yielding an F_1-style overlap score. A proxy for Translation Error Rate (TER) is defined as $1 - \text{ROUGE-L}$. These auxiliary metrics provide complementary perspectives on lexical overlap and structural similarity while remaining strictly aligned with the implemented evaluation procedure.

3.5 Question Answering

Extractive question answering is included in the benchmark as a span prediction task in which the model must identify the start and end positions of an answer within a context passage. Unlike generative sequence-to-sequence formulations, the output space is constrained to positions inside the provided context, yielding a structured prediction problem over token indices.

Formally, let $\mathcal{D} = \{(c_i, q_i, a_i)\}_{i=1}^{N}$ denote a dataset where $c_i = (c_{i,1}, \ldots, c_{i,T_i})$ is a context sequence, $q_i = (q_{i,1}, \ldots, q_{i,S_i})$ is a question sequence, and $a_i = (s_i, e_i)$ identifies the start and end indices of the answer span within the context. After tokenization and numerical encoding through a shared vocabulary $\mathcal{V}$, a parametric model f_θ produces two sequences of logits

$$f_\theta(c_i, q_i) = \left(z_i^{\text{start}}, z_i^{\text{end}}\right), \qquad z_i^{\text{start}}, z_i^{\text{end}} \in \mathbb{R}^{T_i},$$

corresponding to categorical distributions over possible start and end positions. Training is performed by minimizing the sum of two cross-entropy terms, one for the start index and one for the end index.

From the perspective of optimizer benchmarking, extractive QA combines long contextual dependencies, attention-based interactions between question and context, and sparse supervision confined to two token positions per sample. These characteristics produce non-uniform gradient signals and sharp probability distributions over token positions, thereby offering a demanding environment for evaluating convergence stability and generalization across optimization strategies.

3.5.1 SQuAD v1 Dataset

The first dataset employed is the Stanford Question Answering Dataset v1 [**45**], accessed through the HuggingFace `squad` interface. The dataset consists of Wikipedia passages paired with crowd-sourced questions and corresponding answer spans. Official splits comprise training and validation sets.

In the implemented pipeline, tokenization is based on whitespace segmentation. Character-level answer offsets are mapped to token indices through explicit span tracking derived from regular-expression matching over non-whitespace substrings. Context tokens are truncated to a maximum length of 384, while question tokens are truncated to 64. Both sequences are converted to lowercase prior to vocabulary lookup in order to improve lexical coverage and reduce sparsity.

SQuAD v1 provides clean supervision and well-aligned span annotations, reducing ambiguity in evaluation and enabling clearer attribution of performance differences to architectural and optimization factors. Alternative preprocessing strategies consistent with the task include subword tokenization or sliding-window approaches for long contexts. While such extensions can improve representational fidelity, they also alter sequence length distributions and gradient characteristics, thereby influencing optimizer dynamics.

3.5.2 TweetQA Dataset

The second implemented dataset is TweetQA [**46**], accessed through the HuggingFace `tweet_qa` interface. It consists of question-answer pairs over short and informal social media posts, thereby introducing domain variability relative to encyclopedic text.

Consistent with the previous dataset, tokenization relies on whitespace segmentation. Context sequences are truncated to 256 tokens and questions to 64 tokens. Answer span identification is performed by exact token-level subsequence matching between tokenized context and tokenized answer text. If no match is found, a default span is assigned.

TweetQA complements SQuAD by introducing noisier language, informal syntax, and higher lexical variability. These characteristics increase the likelihood of partial overlaps rather than exact span matches, potentially reducing exact match scores and increasing variance in small-sample regimes. From an optimization standpoint, shorter contexts and noisier supervision may lead to different gradient dispersion properties compared to SQuAD, offering a complementary stress test for adaptive and non-adaptive optimizers.

3.5.3 BiLSTM with Attention Model

The first architecture considered is a bidirectional LSTM-based model with dot-product attention over the question. Recurrent neural networks equipped with Long Short-Term Memory (LSTM) units were originally introduced by Hochreiter and Schmidhuber [**38**] to address vanishing gradient issues in sequence modeling. Attention mechanisms enable dynamic alignment between sequences and are central to many QA systems.

In the implemented model, context and question tokens are embedded into a shared embedding space of dimension 256 and processed by independent bidirectional LSTMs with hidden dimension 256. For each context position, similarity scores are computed via dot-product between context and question representations, followed by masked softmax normalization to ignore padded tokens. The attended question representation is fused with the context representation through concatenation of $(h_c, h_q^{\text{att}}, h_c \odot h_q^{\text{att}})$, followed by linear projection and tanh activation. Two independent linear heads produce start and end logits.

Recurrent architectures exhibit sequential inductive biases and may be sensitive to gradient stability over long contexts. Consequently, gradient clipping may constitute a useful stabilization mechanism in this setting.

3.5.4 Transformer-Based QA Model

The second architecture employs separate Transformer encoders for context and question sequences. The Transformer architecture replaces recurrence with multi-head self-attention mechanisms, enabling parallelized sequence modeling and improved long-range dependency handling.

Tokens are embedded into a $d_{\text{model}} = 256$ dimensional space and augmented with learned positional embeddings. Each encoder consists of stacked self-attention layers with $n_{\text{head}} = 4$. After encoding, the question representation is aggregated through masked mean pooling and fused with context representations via concatenation and linear projection. Two linear heads produce start and end logits, with padded positions masked to prevent invalid predictions.

Compared to recurrent architectures, Transformers typically exhibit different gradient scaling properties and may be more sensitive to learning rate scheduling and weight decay. Their reliance on attention-based global interactions can produce sharper token-level distributions, influencing both convergence speed and stability under different optimization algorithms.

3.5.5 Exact Match and Task-Specific Metrics

Evaluation is conducted according to the span-based criteria introduced in the SQuAD benchmark [**45**]. Let $(\hat{s}, \hat{e})$ denote the predicted start and end indices of the answer span, and let (s, e) represent the corresponding ground-truth indices.

Exact Match (EM) is defined as a binary indicator function that evaluates whether the predicted span coincides exactly with the ground-truth span:

$$\text{EM} = \begin{cases} 1 & \text{if } \hat{s} = s \text{ and } \hat{e} = e, \\ 0 & \text{otherwise.} \end{cases}$$

In addition to exact match, token-level Precision, Recall, and F1 score are computed based on the overlap between the predicted and reference token index sets. All reported metrics correspond to arithmetic means aggregated over the full evaluation set.

The F1 score is adopted as the primary metric for model selection, as it accounts for partial overlap between predicted and ground-truth spans and therefore provides a more graded and informative evaluation signal than exact match. While

EM constitutes a stricter correctness criterion, Precision and Recall offer complementary perspectives on over-prediction and under-prediction tendencies. The adopted metrics are fully aligned with the implemented evaluation procedure and remain consistent with the span-based question answering evaluation paradigm established in the relevant literature.

3.6 Named Entity Recognition

Named Entity Recognition (NER) is included in the benchmark as a structured sequence labeling task in which each token is assigned an entity label. Unlike independent token classification, NER exhibits strong inter-token dependencies: valid label sequences must satisfy structural constraints and contextual consistency across the sentence is essential.

Formally, let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ denote a corpus of annotated sentences, where $x_i = (x_{i,1}, \ldots, x_{i,T_i})$ is a token sequence and $y_i = (y_{i,1}, \ldots, y_{i,T_i})$ with $y_{i,t} \in \{0, \ldots, C-1\}$ represents the corresponding tag sequence. After numerical encoding through a word vocabulary $\mathcal{V}$, each sentence is mapped to integer identifiers and padded to a maximum length T. A parametric model f_θ produces emission scores

$$f_\theta(x_i) \in \mathbb{R}^{T \times C},$$

which are subsequently decoded into a structured tag sequence, either greedily or via structured inference.

From the perspective of optimizer benchmarking, NER constitutes a structured prediction problem trained via Conditional Random Field (CRF) negative log-likelihood. Learned transition parameters, masked sequence operations, rare entities, and class imbalance together produce heterogeneous gradient behavior, making this task particularly informative for evaluating optimizer stability and convergence.

3.6.1 CoNLL2003 Dataset

The first dataset employed is the CoNLL2003 English newswire corpus, accessed through the HuggingFace `conll2003` interface. The dataset contains manually annotated sentences with entity categories PER, LOC, ORG, MISC, and the non-entity label O, as originally described in [**47**].

In the implemented pipeline, a word-level vocabulary is constructed from the training split via frequency-based truncation, reserving dedicated indices for `<pad>` and `<unk>`. Tokens are lowercased prior to vocabulary lookup. Sequences are

truncated to a maximum length of 128 tokens and padded to fixed length. The corresponding tag identifiers are either taken directly as numeric indices or remapped to a consistent label index space shared across the dataset. An optional character-level vocabulary is constructed from the training tokens, enabling subword feature extraction. Character sequences are truncated to a maximum length and padded accordingly, producing a tensor of shape $B \times T \times C$ when enabled.

The choice of CoNLL2003 is motivated by its canonical status and the stability of F_1 comparisons across literature. Particular care must be taken to ensure consistency between token padding and mask construction, since CRF computations depend critically on correct sequence-length handling.

3.6.2 WikiANN Dataset

The second dataset corresponds to the English subset of WikiANN, accessed through the HuggingFace `wikiann` interface with configuration `en`. This corpus consists of automatically extracted and annotated Wikipedia sentences.

Preprocessing mirrors that of CoNLL2003: lowercasing, frequency-based vocabulary construction, truncation at 128 tokens, and fixed-length padding. Tag identifiers are provided as integers and are padded consistently with token sequences. A character-level vocabulary is also constructed from the training tokens, enabling experiments with character-based representations. Compared to CoNLL2003, WikiANN exhibits greater lexical diversity and noisier annotation patterns, thereby stressing model generalization and robustness.

3.6.3 BiLSTM-CRF Model

The first architecture considered is a bidirectional LSTM with a CRF decoding layer, following the classical formulation introduced in [**48**]. Given embedded input tokens $e_t \in \mathbb{R}^d$, a bidirectional LSTM computes contextual representations

$$h_t = \mathrm{BiLSTM}(e_{1:T})_t,$$

which are linearly projected to emission scores

$$z_t = W h_t + b \in \mathbb{R}^C.$$

A CRF layer parameterized by learned transition matrices models dependencies between consecutive tags. Training minimizes the negative log-likelihood

$$\mathcal{L}_{\mathrm{CRF}} = -\log p_\theta(y \mid x),$$

computed via the difference between the log-partition function and the joint sequence score. Decoding during evaluation is performed via Viterbi inference. This architecture represents a classical structured sequence model whose optimization dynamics involve both recurrent parameters and CRF transitions.

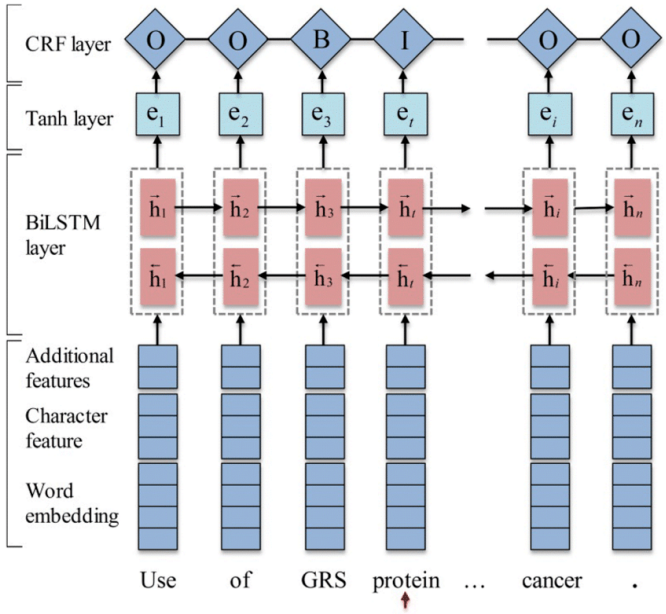

Figure 3.10: BiLSTM-CRF architecture for NER

3.6.4 BiLSTM-CRF with Character CNN Model

The second architecture augments the previous model with character-level representations, following approaches such as the one proposed by Ma et al. [**49**]. Each token is decomposed into characters, embedded, and processed by a one-dimensional convolution with ReLU activation, followed by max-pooling

$$c_t = \max_k \mathrm{ReLU}(\mathrm{Conv1D}(\mathrm{Emb}_{\mathrm{char}}(x_t)))_k.$$

The resulting character feature vector c_t is concatenated with the word embedding e_t, forming an enriched representation passed to the bidirectional LSTM. Emission and CRF layers remain structurally identical to the previous model.

Character-level features are particularly beneficial for rare and out-of-vocabulary tokens, since they encode subword morphology and orthographic patterns. Their inclusion increases parameter count moderately and introduces additional convolutional parameters whose gradients interact with recurrent and CRF components during optimization.

3.6.5 F1 Score and Task-Specific Metrics

The evaluation protocol strictly adheres to the implemented `NERMetrics` class, thereby ensuring full consistency between the mathematical formulation reported in this section and the actual computational procedure adopted during experimentation. In particular, all padding positions introduced during sequence batching are excluded from evaluation through a binary mask, so that metric computation is restricted exclusively to valid token positions.

Let $\hat{y}_t$ and y_t denote respectively the predicted and ground-truth tag at position t for a given sequence, after masking padded elements. The metric accumulator maintains three global counters: true positives (TP), false positives (FP), and false negatives (FN). These quantities are computed at token level according to the following definitions:

$$\mathrm{TP} = \sum_t \mathbf{1}\Big[(\hat{y}_t = y_t) \wedge (y_t \neq 0)\Big],$$

$$\mathrm{FP} = \sum_t \mathbf{1}\Big[(\hat{y}_t \neq y_t) \wedge (\hat{y}_t \neq 0)\Big],$$

$$\mathrm{FN} = \sum_t \mathbf{1}\Big[(\hat{y}_t \neq y_t) \wedge (y_t \neq 0)\Big].$$

In this formulation, the label 0 corresponds to the non-entity class `O`. Consequently, only tokens associated with entity categories contribute positively to the true positive count, while misclassifications involving entity labels are reflected in the false positive and false negative terms. This design choice yields a micro-averaged token-level evaluation over entity predictions, rather than a span-level assessment.

Based on these accumulated statistics, precision and recall are defined in the standard manner as

$$\mathrm{Precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}, \qquad \mathrm{Recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}},$$

with appropriate safeguards against division by zero in degenerate cases. The harmonic mean of precision and recall yields the F_1 score,

$$\mathrm{F}_1 = \frac{2\,\mathrm{Precision}\,\mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}},$$

which constitutes the primary metric for model selection and comparative evaluation within the benchmark.

It is important to emphasize that the reported F_1 score corresponds to a micro-average computed over all valid token positions, excluding padding. While this differs from the traditional CoNLL span-based evaluation protocol, it remains strictly aligned with the implemented metric and therefore guarantees methodological coherence across all experimental configurations considered in this thesis. Such internal consistency is essential in the context of optimizer benchmarking, where the objective is comparative analysis under a fixed and reproducible evaluation framework.

3.7 Text Generation

Text generation is included in the benchmark through autoregressive language modeling, a setting in which models are trained to predict the next token in a sequence. Given a tokenized sequence $x = (x_1, \ldots, x_T)$ drawn from a vocabulary $\mathcal{V}$, a language model parameterized by θ defines

$$p_\theta(x) = \prod_{t=1}^{T} p_\theta(x_t \mid x_{<t}),$$

where $x_{<t}$ denotes the prefix $(x_1, \ldots, x_{t-1})$. Training proceeds by maximizing the log-likelihood of the observed corpus, equivalently minimizing the average cross-entropy loss

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T_i} \log p_\theta(x_t^{(i)} \mid x_{<t}^{(i)}).$$

This formulation underlies neural probabilistic language modeling and many modern generative architectures. In the implemented task, next-token prediction is performed over fixed-length blocks extracted from concatenated token streams. Given an input tensor $x \in \mathbb{N}^{B \times T}$ and its one-step shifted target y, the model outputs logits $z \in \mathbb{R}^{B \times T \times |\mathcal{V}|}$, and the loss is computed as cross-entropy with padding tokens ignored via a dedicated index. This formulation directly aligns with perplexity-based evaluation and provides a sensitive indicator of optimization quality in generative settings.

From the perspective of optimizer benchmarking, language modeling is particularly informative because the output space is high-dimensional, gradients propagate through long temporal dependencies, and autoregressive training amplifies conditioning differences between recurrent and attention-based architectures. This makes the task suitable for comparative analysis of optimizer robustness across heterogeneous model families.

3.7.1 WikiText-2 Dataset

The first dataset employed is WikiText-2, introduced by Merity et al. in the context of improved language modeling benchmarks [**50**]. It consists of curated Wikipedia articles designed to preserve long-range dependencies and realistic lexical statistics.

In the implemented pipeline, raw text lines are extracted and tokenized using simple whitespace splitting. A vocabulary is constructed from the training corpus up to a maximum size of 50,000 tokens, reserving indices for `<pad>` and `<unk>` symbols. The entire token stream is concatenated and partitioned into fixed-length blocks of size $T = 64$, yielding input–target pairs (x, y) such that y corresponds to a one-position right shift of x.

WikiText-2 is adopted in this benchmark because it represents a widely recognized word-level language modeling corpus whose perplexity values are highly sensitive to architectural design and optimization strategy, thereby providing a controlled yet informative setting for analyzing optimizer behavior. Its moderate scale enables efficient experimentation while preserving realistic lexical diversity, making it particularly suitable for systematic benchmarking studies.

3.7.2 Penn Treebank (PTB) Text-Only Dataset

The second dataset is a text-only variant of the Penn Treebank, originally introduced by Marcus et al. [**51**]. PTB has historically served as a compact yet influential benchmark for language modeling research. As in WikiText-2, preprocessing consists of whitespace tokenization, vocabulary construction from the training split, and fixed-length block packing. PTB is comparatively smaller and exhibits a more constrained lexical distribution, resulting in faster iteration times and lower computational overhead.

Its compact size makes it particularly suitable for controlled optimizer ablations. Nevertheless, classical preprocessing conventions may normalize or simplify punctuation and token boundaries, which affects vocabulary richness and typical perplexity ranges.

3.7.3 GPT Small Model

The first architecture considered is a compact transformer-based autoregressive model inspired by generative pretraining framework of Radford et al. [**52**]. It comprises token embeddings of dimension $d = 256$, learned positional embeddings, a stack of transformer encoder layers with multi-head self-attention, and a linear language modeling head projecting to $|\mathcal{V}|$ logits.

Although implemented via a `TransformerEncoder`, autoregressive behavior is enforced through a causal mask that prevents attention to future tokens. Given input embeddings $h_0 = E_{\text{tok}}(x) + E_{\text{pos}}(x)$, the stacked self-attention blocks produce contextual representations h_L, which are linearly mapped to vocabulary logits.

Transformer-based models rely on global self-attention mechanisms rather than sequential recurrence, which fundamentally alters gradient flow characteristics and conditioning properties. These structural differences are particularly relevant in optimizer benchmarking.

3.7.4 GRU-Based Language Model

The second model is a recurrent language model based on the Gated Recurrent Unit (GRU) formulation introduced by Cho et al. [**53**]. It consists of an embedding layer, a two-layer GRU with hidden size 256, dropout regularization, and a linear output projection. An optional weight-tying mechanism shares parameters between the embedding matrix and the output projection, reducing parameter count and potentially improving generalization by enforcing representational consistency.

Recurrent architectures process sequences sequentially and rely on hidden-state propagation rather than global self-attention mechanisms. From an optimization standpoint, GRUs typically exhibit different gradient conditioning properties, especially on smaller corpora, and may display distinct sensitivity to learning rate schedules and gradient clipping strategies.

3.7.5 Perplexity and Task-Specific Metrics

The principal evaluation criterion adopted for this task is *perplexity*, which constitutes the standard metric for autoregressive language modeling. Let $\mathcal{T}$ denote the set of evaluated tokens and let $\ell_i = -\log p_\theta(x_i \mid x_{<i})$ represent the negative log-likelihood associated with token x_i. The average per-token cross-entropy is therefore given by

$$\bar{\ell} = \frac{1}{M} \sum_{i=1}^{M} \ell_i,$$

where $M = |\mathcal{T}|$ denotes the total number of evaluated tokens. Perplexity is defined as the exponential of this average:

$$\text{PPL} = \exp(\bar{\ell}).$$

In the implemented evaluation pipeline, perplexity is computed from accumulated token-level negative log-likelihood values derived from model predictions over the evaluation corpus and aggregated across fixed-length blocks. Although the computation follows a simplified proxy formulation consistent with the implemented accumulator logic, it preserves the monotonic relationship between likelihood improvement and perplexity reduction. Consequently, perplexity is designated as the primary model selection criterion, and lower values are considered indicative of better performance.

In addition to perplexity, auxiliary overlap-based metrics are computed in order to provide complementary structural information. A simplified BLEU score estimates n-gram overlap between candidate and reference token sequences, while ROUGE-L evaluates similarity based on the longest common subsequence. A METEOR-like proxy is derived from ROUGE statistics, serving as an additional indicator of token-level alignment. These metrics operate on deterministic next-token predictions rather than unconstrained free-form generation; as such, they should be interpreted as approximate structural similarity measures rather than canonical generation benchmarks.

3.8 Text Summarization

Text summarization is included in the benchmark as a sequence-to-sequence generation task requiring models to produce concise target sequences from longer source documents. Abstractive text summarization can be formalized as a conditional sequence-to-sequence learning problem in which a source sequence $x = (x_1, \ldots, x_S)$ is mapped to a target sequence $y = (y_1, \ldots, y_T)$ representing a condensed semantic rendering of the input. Given source and target vocabularies $\mathcal{V}_s$ and $\mathcal{V}_t$, respectively, a model parameterized by θ defines the conditional distribution

$$p_\theta(y \mid x) = \prod_{t=1}^{T} p_\theta(y_t \mid y_{<t}, x),$$

where $y_{<t}$ denotes the previously generated prefix. This probabilistic formulation follows the encoder–decoder paradigm introduced by Sutskever et al. [**41**], in which a neural network learns to model structured conditional distributions over variable-length sequences.

Training proceeds by maximizing the conditional log-likelihood over a dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$, equivalently minimizing the cross-entropy objective

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T_i} \log p_\theta\Big(y_t^{(i)} \mid y_{<t}^{(i)}, x^{(i)}\Big).$$

In the implemented task, teacher forcing is adopted, meaning that the decoder receives ground-truth prefixes during training. This training strategy is consistent with classical neural sequence transduction frameworks and stabilizes gradient propagation in deep encoder–decoder architectures. Padding tokens are excluded from loss computation via a dedicated ignore index, and label smoothing with coefficient 0.1 is enabled when supported by the backend implementation.

From the perspective of optimizer benchmarking, abstractive summarization provides a particularly informative setting. The task combines deep stacked transformations, autoregressive decoding, and cross-attention interactions. The loss landscape therefore reflects both compression fidelity and linguistic fluency, producing gradient dynamics that differ from those observed in classification or unconditional language modeling tasks.

3.8.1 CNN/DailyMail Dataset

The first dataset employed is the CNN/DailyMail corpus, introduced by Hermann et al. [**54**]. It consists of news articles paired with human-written highlights summarizing their core content. The dataset is accessed through the HuggingFace `cnn_dailymail` interface, which provides standardized training, validation, and test splits.

Within the implemented pipeline, raw textual fields `article` and `highlights` are processed via lowercase whitespace tokenization. Separate vocabularies are constructed from a subset of the training split up to a maximum size of 50,000 tokens. Sequences are encoded with explicit boundary tokens and truncated to predefined maximum lengths. During batching, dynamic padding yields tensors

$$\{\texttt{src}, \texttt{tgt\_in}, \texttt{tgt\_out}\},$$

where `tgt_in` and `tgt_out` correspond to left- and right-shifted target sequences for teacher-forced decoding. CNN/DailyMail is widely regarded as a canonical summarization benchmark due to its realistic article length, lexical diversity, and nontrivial compression requirements. The relatively long source sequences stress encoder depth and attention capacity, thereby providing a meaningful testbed for analyzing optimizer stability and convergence behavior.

3.8.2 AESLC Dataset

The second dataset is AESLC, which pairs email bodies with corresponding subject lines. In contrast to CNN/DailyMail, AESLC features substantially shorter targets, as subject lines typically consist of only a few tokens. The dataset is accessed via

the HuggingFace `aeslc` interface and provides predefined splits.

Preprocessing mirrors that of CNN/DailyMail, including whitespace tokenization, vocabulary construction from a subset of the training split, and explicit insertion of boundary tokens. The shorter target sequences enable rapid training iterations and amplify differences in convergence speed across optimizers. However, the brevity of outputs may also cause overlap-based metrics to saturate quickly, making regularization mechanisms such as label smoothing particularly relevant.

3.8.3 BART Small Model

The first architecture, `bart_small`, is a compact transformer-based encoder–decoder model implemented using `nn.Transformer`. It comprises distinct embedding layers for source and target vocabularies with dimension $d_{\text{model}} = 256$, sinusoidal positional encodings, and a symmetric encoder–decoder stack with $n_{\text{head}} = 4$ attention heads and $L = 2$ layers in both the encoder and decoder. The decoder outputs are mapped to target vocabulary logits through a linear projection.

Given discrete source tokens $x \in \mathbb{N}^{B\times S}$ and decoder input tokens $y_{\text{in}} \in \mathbb{N}^{B\times T}$, the model computes

$$H_s = \mathrm{PE}(E_s(x)), \qquad H_t = \mathrm{PE}(E_t(y_{\text{in}})),$$

followed by an encoder–decoder transformer mapping

$$U = \mathrm{Transformer}(H_s, H_t), \qquad Z = W_o U,$$

where $Z \in \mathbb{R}^{B\times T\times|\mathcal{V}_t|}$ denotes the logits and W_o is the output projection matrix. This configuration provides a moderate-capacity baseline whose optimization behavior reflects attention-driven encoder–decoder coupling. Possible architectural variants include learned positional embeddings, deeper stacks, alternative normalization placement, or different feedforward dimensionalities; however, the implemented model adopts a compact setting to enable controlled benchmarking under limited computational budgets.

3.8.4 Tiny Transformer Seq2Seq Model

The second architecture, `tiny_transformer_seq2seq`, instantiates an explicit transformer encoder and decoder through *TransformerEncoderLayer* and *TransformerDecoderLayer* modules. It uses sinusoidal positional encoding, $d_{\text{model}} = 256$, $n_{\text{head}} = 4$, $L = 3$ layers for both encoder and decoder, a feedforward dimension of 1024, and dropout 0.1. A pre-layer normalization configuration is adopted

(`norm_first=True`), and the output projection is implemented as a bias-free linear layer. When dimensional compatibility holds, the model optionally ties the target embedding matrix and the output projection weights, reducing parameter count and enforcing representational coupling between input and output spaces.

In addition to encoder self-attention and decoder cross-attention, the implementation explicitly constructs a causal mask for decoder self-attention. Denoting by T the target length, the causal mask $M \in \{0,1\}^{T\times T}$ satisfies $M_{ij} = 1$ for $j > i$, preventing access to future tokens. This ensures autoregressive conditioning within the decoder while maintaining teacher forcing through the provided prefixes. Compared to the previously described model, this architecture exposes finer-grained design decisions relevant to optimization, including pre-layer normalization and embedding tying. The implemented choice provides a lightweight yet nontrivial encoder–decoder transformer whose convergence can be sensitive to optimizer hyperparameters, making it suitable for rapid optimizer sweeps.

3.8.5 ROUGE and Task-Specific Metrics

Evaluation is performed using overlap-based metrics computed from deterministic argmax predictions. After mapping token identifiers back to lexical units (excluding special symbols), each candidate summary is compared against its reference counterpart. The primary metric is ROUGE, originally proposed by Lin **[55]** as a recall-oriented measure of n-gram and subsequence overlap for summarization evaluation. In particular, ROUGE-L relies on the length of the longest common subsequence (LCS) between candidate c and reference r. Denoting by $\mathrm{LCS}(c, r)$ the subsequence length, the score is computed through precision and recall derived from this quantity and averaged over the dataset:

$$\mathrm{ROUGE} = \frac{1}{N}\sum_{i=1}^{N} R\Big(c^{(i)}, r^{(i)}\Big).$$

Higher values indicate stronger lexical overlap between generated and reference summaries. For this reason, ROUGE is designated as the primary model selection criterion. In addition, a simplified BLEU score is computed to capture n-gram precision overlap. A METEOR-like proxy is defined by reusing the ROUGE-L statistic as a recall-oriented indicator, providing complementary structural information without implementing the full METEOR alignment procedure. Finally, a compression ratio is reported as

$$\mathrm{CR} = \frac{1}{N}\sum_{i=1}^{N} \frac{|c^{(i)}|}{\max(1, |r^{(i)}|)},$$

quantifying the average length of generated summaries relative to references.

Values close to unity indicate comparable compression ratios, whereas systematic deviations may reveal under- or over-generation tendencies. Within the benchmarking framework, ROUGE serves as the principal performance indicator, while BLEU, the METEOR proxy, and compression ratio provide auxiliary diagnostic perspectives on lexical overlap and structural compression behavior.

3.9 Experimental Protocol and Global Benchmarking Framework

This section provides a unified description of the global experimental protocol adopted across all tasks in the benchmark. While each task differs in data modality, model architecture, and task-specific evaluation criteria, the overall execution pipeline, optimization policies, logging strategy, and reporting infrastructure remain structurally consistent. Such uniformity is essential for ensuring methodological fairness, numerical comparability, and interpretability of results across heterogeneous problem domains.

3.9.1 Run Configuration and Reproducibility

All experiments are executed through a centralized command-line interface that explicitly specifies the tuple

$$(\texttt{task}, \texttt{dataset}, \texttt{model}, \texttt{optimizer}),$$

together with global training hyperparameters including the number of epochs, batch size, learning rate policy, scheduler policy, random seed, and data-loading configuration.

To guarantee reproducibility, a fixed random seed is enforced for each run. The computational device (CPU or GPU) is automatically detected, and all datasets are stored under a user-defined root directory to ensure deterministic caching behavior. This design minimizes uncontrolled variability and allows fair cross-optimizer comparisons.

Each run stores a complete `run_metadata.json` file, recording all relevant configuration parameters, including per-optimizer learning rates. This explicit metadata serialization ensures traceability and supports subsequent auditability of numerical outcomes.

3.9.2 Learning Rate Policies and Optimizer Configuration

The benchmark supports two learning rate selection strategies:

- **Fixed policy**: a single learning rate is imposed across all optimizers.
- **Per-optimizer policy**: each optimizer is assigned a default learning rate calibrated according to commonly accepted practical guidelines from the original optimizer publications.

The adoption of standardized, publication-recommended hyperparameter configurations for all evaluated optimizers is a deliberate methodological choice that warrants explicit justification. Rather than performing per-optimizer hyperparameter searches, the benchmark evaluates each method under its intended out-of-the-box behavior. This design reflects a principled commitment to ecological validity: benchmarking optimizers under default configurations mirrors the most common real-world scenario, in which practitioners apply optimization algorithms without extensive tuning. As Choi et al. demonstrate, comparisons that rely on heavily tuned hyperparameters risk conflating the quality of the optimization algorithm with the quality of the tuning procedure, thereby obscuring the intrinsic properties of each method [**28**].

Equally important, standardizing hyperparameters across all optimizers eliminates a significant source of experimental confounding. If each optimizer were tuned independently, observed performance differences could reflect the effort or effectiveness of the tuning process rather than the update rule itself. By holding hyperparameters constant, the benchmark isolates the contribution of the optimization algorithm to convergence behavior and final performance. This approach is consistent with the methodology adopted in related benchmarking work, including the systematic study by Schmidt et al., which similarly evaluates optimizers under unified hyperparameter conditions [**26**].

In addition, weight decay coefficients are assigned according to optimizer families (e.g., decoupled weight decay for AdamW-like methods), ensuring coherent regularization across experiments.

3.9.3 Scheduler Policies and Stability Controls

To improve numerical stability and convergence behavior, learning rate schedulers are integrated into the training loop. The scheduler policy can be manually selected or automatically assigned according to task category. In particular:

- Cosine annealing is preferred for vision-oriented tasks.

- Plateau-based reduction is adopted for NLP and sequence modeling tasks.

This differentiation reflects empirical observations that distinct modalities often benefit from different learning rate decay dynamics.

Furthermore, gradient clipping is optionally applied for sequence-based tasks, where exploding gradients may compromise stability. The clipping threshold is chosen conservatively to preserve gradient signal while preventing numerical divergence. These mechanisms collectively reduce variance in training trajectories, thus improving the reliability of optimizer benchmarking.

3.9.4 Universal Metrics and Convergence Measurement

Although each task defines its own primary metric, two quantities are systematically tracked across all experiments:

1. **Training and validation loss**
2. **Convergence behavior**

The loss function provides a task-independent signal directly tied to the optimization objective. Tracking both training and validation loss allows monitoring of generalization gaps and overfitting dynamics.

Convergence is quantified through:

- the epoch at which a plateau condition is detected,
- the cumulative runtime required to reach convergence.

Runtime is computed as the cumulative sum of per-epoch training and evaluation durations measured during execution. This dual perspective (epoch-based and time-based) is essential for a meaningful comparison between optimizers whose per-step computational costs differ.

3.9.5 History Tracking and Data Serialization

For each optimizer, the training process produces structured histories containing:

- per-epoch validation loss,
- primary metric values,
- runtime statistics,
- auxiliary scalar metrics (where applicable).

These histories are serialized in JSON format for full fidelity storage and in CSV format for tabular analysis. Two complementary CSV files are generated:

- **Summary table**: aggregates best primary metric, best validation loss, convergence epoch, and associated runtimes.
- **Extensive table**: stores per-epoch metrics for detailed temporal analysis.

This dual-level reporting structure enables both high-level comparative evaluation and fine-grained diagnostic inspection, thereby supporting rigorous numerical interpretation.

3.9.6 Visualization Strategy

To facilitate interpretability and comparative analysis, multiple plots are automatically generated:

- Primary metric and validation loss (dual-axis representation),
- Training vs. validation loss curves,
- Cross-optimizer comparisons of primary metrics,
- Cross-optimizer comparisons of loss trajectories.

The dual-axis representation is particularly useful for visualizing the interaction between objective minimization and task-level performance. By maintaining a consistent plotting structure across tasks, visual inspection becomes standardized and cognitively efficient.

All visual outputs are saved in structured directories organized by task, model, dataset, and optimizer. This hierarchical organization simplifies large-scale experimental campaigns and ensures that results remain systematically interpretable.

3.9.7 Design Rationale for Numerical Benchmarking

The overall experimental design is guided by three core principles:

1. **Fairness**: identical data splits, seeds, and training budgets across optimizers.
2. **Stability**: scheduler policies and gradient controls tailored to task modality.
3. **Interpretability**: structured logging, standardized plots, and aggregated tables.

By enforcing a unified yet adaptable training protocol, the framework ensures that observed performance differences primarily reflect optimizer behavior rather than confounding implementation artifacts. The training horizon of 20 epochs is itself a deliberate feature of the benchmark design rather than an incidental constraint. Restricting training to a fixed and relatively short schedule focuses the evaluation on early-to-mid-stage optimization dynamics — a regime that is critical in practice, since training budgets are frequently constrained and practitioners must often make deployment decisions well before asymptotic convergence is achieved. This focus on the transient regime provides insights that are complementary to, and in some respects more practically relevant than, the asymptotic behavior studied under unconstrained training conditions. The protocol is deliberately designed to remain reproducible in low-budget and time-constrained computational environments, consistent with the realistic resource conditions described by Schmidt et al. [**26**]. As a result, the framework supports not only final performance comparisons but also structured analyses of convergence speed, stability patterns, generalization dynamics, and computational efficiency. The protocol defined in this chapter serves as the common experimental basis for all results reported in the next chapter, where the focus shifts from benchmark design to empirical outcomes.

Chapter 4

Experiments

This chapter presents the empirical evaluation conducted using the benchmarking framework introduced in the previous chapter. The objective of this chapter is to provide a systematic comparison of widely used optimization algorithms across a heterogeneous set of deep learning tasks spanning both computer vision and natural language processing. By evaluating all optimizers within a unified experimental protocol, the study aims to isolate differences that arise from the optimization strategies themselves rather than from variations in implementation, training configuration, or hardware conditions.

All experiments are executed under standardized and reproducible settings. Training procedures follow identical initialization policies, consistent logging strategies, and controlled computational resources across all runs. The experiments are conducted in a cloud-based environment using Google Colab with a Python 3 backend deployed on a Google Compute Engine virtual machine equipped with an NVIDIA A100 GPU accelerator, approximately 167.1 GB of system RAM, and 80.0 GB of GPU memory. The use of a homogeneous hardware environment ensures that runtime measurements remain comparable across experiments.

A common training protocol is adopted across all tasks in order to preserve methodological consistency. As described in Chapter 3, each training run is limited to a maximum of 20 epochs. While longer training schedules could yield higher absolute performance, the fixed training horizon reflects a realistic resource-constrained scenario and makes it possible to compare optimizers in terms of early training efficiency, convergence behavior, and stability under a common computational budget. This design choice also ensures that all optimizers are evaluated under comparable computational budgets.

The experimental analysis is organized by task. For each problem domain, the

chapter first introduces the experimental configuration and the commands required to reproduce the runs, followed by a detailed discussion of the empirical results. The analysis examines task-specific evaluation metrics together with validation loss, convergence behavior, runtime, and training stability. By comparing optimizer behavior across multiple datasets and model architectures, the chapter highlights recurring patterns as well as task-dependent differences in optimization dynamics.

Depending on the task, different primary evaluation metrics are used. Classification tasks are assessed using accuracy, machine translation using BLEU, summarization using ROUGE-L, language modeling using perplexity, and structured prediction tasks such as question answering and named entity recognition using F1 score. Unless otherwise noted, higher values indicate better performance; perplexity is the only primary metric for which lower values are preferred. For consistency, certain evaluation metrics are reported on a normalized [0,1] scale when required by the evaluation framework.

To avoid redundancy, dataset and model descriptions are kept concise in the following sections, as their full methodological motivation has already been presented in Chapter 3. The emphasis here is placed on the empirical behavior of the optimizers under the shared protocol defined previously.

4.1 Image Classification

Image classification provides a controlled environment for analyzing optimization behavior under non-convex objectives. The task is evaluated using two datasets (MNIST and CIFAR-10) and two model architectures: a convolutional neural network trained from scratch and a deeper ResNet18 architecture. This combination allows the benchmark to capture both relatively simple visual recognition problems and moderately complex image classification scenarios.

Each run is limited to 20 epochs and uses a batch size of 128. A cosine learning rate scheduler is employed in all configurations, consistent with the protocol defined in Chapter 3. To maintain tractable training times, the number of training samples is restricted using the `-max-samples` parameter. Data loading is parallelized with eight workers to minimize input pipeline overhead. The following commands report the configurations used to reproduce the evaluated experiments.

MNIST – CNN trained from scratch

```
python -m benchmark.main
  --task image_classification --dataset mnist
  --model cnn_scratch --optimizers all
  --epochs 20 --batch-size 128 --max-samples 20000 --num-workers 8
```

MNIST – ResNet18

```
python -m benchmark.main
  --task image_classification --dataset mnist
  --model resnet18 --optimizers all
  --epochs 20 --batch-size 128 --max-samples 20000 --num-workers 8
```

CIFAR-10 – CNN trained from scratch

```
python -m benchmark.main
  --task image_classification --dataset CIFAR-10
  --model cnn_scratch --optimizers all
  --epochs 20 --batch-size 128 --max-samples 15000 --num-workers 8
```

CIFAR-10 – ResNet18

```
python -m benchmark.main
  --task image_classification --dataset CIFAR-10
  --model resnet18 --optimizers all
  --epochs 20 --batch-size 128 --max-samples 15000 --num-workers 8
```

The image classification results indicate that optimizer differences become meaningful primarily as dataset and architectural complexity increase. On the relatively simple MNIST dataset, performance differences between optimizers remain limited. In particular, when training the deeper ResNet18 architecture, several optimizers reach nearly identical peak accuracy values around 0.996.

In this regime, the choice of optimizer has only a minor influence on final predictive performance, while differences in convergence speed and runtime become more relevant. More informative differences emerge when considering the CIFAR-10 dataset, which presents a more challenging visual recognition problem. For the CNN trained from scratch, AdaDelta achieves the highest accuracy (0.733) together with the lowest validation loss.

When the architecture is replaced with ResNet18, Adam produces the strongest overall performance (0.768), closely followed by other adaptive optimizers including

Table 4.1: Summary of image classification results across datasets and architectures. Bold values indicate the best accuracy achieved within each configuration.

Dataset	Model	Top Optimizers	Accuracy	Loss	Runtime
MNIST	CNN Scratch	**AdaFactor**, GSAM, AdaDelta, RMSProp, Adam	**0.994**	0.527	28–54 s
CIFAR-10	CNN Scratch	**AdaDelta**, AdaFactor, AdaBelief, AdamW	**0.733**	1.15	∼44 s
MNIST	ResNet18	SAM, AdaBelief, Lion, AdaFactor, GSAM	**0.996**	0.511–0.512	90–166 s
CIFAR-10	ResNet18	**Adam**, AdamW, AdaBelief, RAdam, AdaFactor	**0.768**	1.04	∼84 s

AdamW, AdaBelief, and RAdam. These results suggest that adaptive learning-rate mechanisms provide advantages when the optimization landscape becomes more complex and training horizons are limited.

SGD-based optimizers generally achieve slightly lower final accuracy values across all configurations. Nevertheless, these methods often exhibit stable learning trajectories and lower computational cost. Their relative disadvantage in this benchmark is therefore primarily related to slower convergence within the constrained 20-epoch budget rather than to fundamentally weaker optimization behavior.

Sharpness-aware optimizers such as SAM and GSAM present a different trade-off. In some settings, particularly on MNIST with deeper architectures, they match the best accuracy values obtained by adaptive optimizers. However, their runtime is consistently higher due to the additional gradient evaluations required at each optimization step. Consequently, while these methods can achieve competitive performance, their computational overhead limits their efficiency under the fixed training budget adopted in this benchmark.

A comparison between architectures further highlights the influence of model capacity on optimization behavior. The deeper ResNet18 architecture enables several optimizers to reach near-saturated accuracy levels on MNIST, whereas performance on CIFAR-10 remains substantially lower across all configurations. This difference reflects the higher visual complexity of CIFAR-10 and the increased difficulty of optimizing deeper models under a limited training horizon.

Overall, the image classification experiments indicate that adaptive optimizers are particularly effective in the evaluated settings, especially under the adopted training budget. SGD-based methods remain attractive from an efficiency and stability perspective, while sharpness-aware methods provide competitive accuracy but incur significantly higher computational cost.

4.2 Semantic Segmentation

Semantic segmentation represents a more demanding computer vision task compared to image classification, as the model must produce dense pixel-level predictions rather than a single class label per image. From an optimization perspective, this substantially increases the dimensionality of the learning problem and introduces more complex gradient dynamics. The benchmark therefore evaluates optimizer behavior under two segmentation datasets, Oxford-IIIT Pet and Pascal VOC2012, together with two architectures: a vanilla U-Net and DeepLabV3 with a ResNet-50 backbone. This configuration allows the analysis to capture optimization behavior under both moderately complex and more challenging dense prediction scenarios.

Each run is limited to 20 epochs. Due to the higher memory requirements of dense prediction models, the batch size is reduced to 8. Inputs are resized to 256×256 pixels and normalized according to the preprocessing strategy described in Chapter 3.

The training objective combines cross-entropy loss with a soft Dice component. This formulation encourages spatial overlap between predicted and ground-truth segmentation masks while maintaining stable gradient signals during optimization. Class imbalance is mitigated through dynamically computed batch-level class weights within the cross-entropy component, and gradient clipping with maximum norm 1.0 is applied in order to stabilize training.

The following commands summarize the configurations adopted for reproducibility.

Oxford-IIIT Pet – U-Net

```
python -m benchmark.main
  --task semantic_segmentation --dataset oxford_iiit_pet
  --model unet_vanilla --optimizers all
  --epochs 20 --batch-size 8 --max-samples 5000 --num-workers 8
```

Oxford-IIIT Pet – DeepLabV3 ResNet-50

```
python -m benchmark.main
  --task semantic_segmentation --dataset oxford_iiit_pet
  --model deeplabv3_ResNet-50 --optimizers all
  --epochs 20 --batch-size 8 --max-samples 5000 --num-workers 8
```

Pascal VOC2012 – U-Net

```
python -m benchmark.main
  --task semantic_segmentation --dataset voc2012
  --model unet_vanilla --optimizers all
  --epochs 20 --batch-size 8 --max-samples 3000 --num-workers 8
```

Pascal VOC2012 – DeepLabV3 ResNet-50

```
python -m benchmark.main
  --task semantic_segmentation --dataset voc2012
  --model deeplabv3_ResNet-50 --optimizers all
  --epochs 20 --batch-size 8 --max-samples 3000 --num-workers 8
```

The semantic segmentation results reveal a strong interaction between optimizer behavior, dataset difficulty, and architectural choice. In contrast to the classification experiments, where adaptive methods frequently dominate, segmentation reveals a more complex interaction between model structure and optimization strategy.

When training DeepLabV3 with a ResNet-50 backbone on the Oxford-IIIT Pet dataset, SGD-based optimizers achieve the strongest segmentation performance. In particular, SGD with Momentum reaches the highest primary metric value (0.910), closely followed by standard SGD and LAMB. These methods also produce competitive validation loss values while maintaining relatively stable training dynamics. This behavior suggests that the DeepLab architecture interacts favorably with the stable gradient updates provided by SGD-based optimizers.

A different pattern emerges on the Pascal VOC2012 dataset. Although LAMB achieves the highest primary metric (0.547), the overall segmentation quality remains substantially lower than on Oxford-IIIT Pet. Many optimizers struggle to achieve stable convergence within the 20-epoch training horizon, and performance differences between methods become less pronounced. These results highlight the increased difficulty of the dataset and suggest that the limited training budget may prevent certain optimization strategies from fully converging.

When the architecture is replaced with the simpler U-Net model, the ranking of optimizers changes considerably. On the Oxford-IIIT Pet dataset, adaptive optimizers dominate the best-performing region. RAdam achieves the highest primary metric (0.837), followed closely by AdaBelief and AdaFactor. These methods combine strong segmentation quality with relatively stable optimization trajectories, indicating that adaptive learning-rate mechanisms are particularly beneficial for this architecture.

The Pascal VOC2012 experiments with U-Net represent the most challenging scenario in the segmentation benchmark. None of the evaluated optimizers exceed a primary metric of 0.22, indicating that the architecture struggles to capture the complexity of the dataset within the limited training horizon. Although LAMB achieves the highest performance in this configuration, the differences between optimizers remain relatively small, and training curves frequently display unstable behavior. This suggests that the architectural capacity of the model becomes the dominant limiting factor in this setting.

Across all segmentation experiments, sharpness-aware methods such as SAM and GSAM achieve intermediate segmentation quality but incur substantially higher computational cost due to the additional gradient computations required during each optimization step. While these methods occasionally approach the performance of the strongest optimizers, they do not emerge as the best-performing strategy in any configuration.

Table 4.2: Summary of semantic segmentation results

Dataset	Model	Top Optimizers	DICE	Loss	Runtime
Oxford-IIIT Pet	ResNet-50	**SGD with Momentum**, SGD, LAMB, LARS, Yogi	**0.910**	0.419–0.435	338–751 s
Pascal VOC2012	ResNet-50	**LAMB**, SGD with Momentum, LARS, SGD, AdaDelta	**0.547**	0.679–0.750	465–586 s
Oxford-IIIT Pet	U-Net	**RAdam**, AdaBelief, AdaFactor, Lion, RMSProp	**0.837**	0.579–0.588	347–660 s
Pascal VOC2012	U-Net	**LAMB**, SGD with Momentum, LARS, SGD, Yogi	**0.219**	1.42–1.90	356–1070 s

Taken together, the segmentation experiments suggest that optimizer behavior in dense prediction tasks is strongly influenced by architectural design. DeepLabV3 tends to favor SGD-based methods, whereas U-Net benefits more consistently from adaptive optimizers. At the same time, the difficulty of the dataset plays a crucial role in shaping optimization dynamics. Oxford-IIIT Pet allows optimizers to reach relatively high segmentation quality within the limited training budget, whereas Pascal VOC2012 remains challenging regardless of the optimization strategy. These observations emphasize that evaluating optimizers in segmentation requires considering not only final performance but also the interaction between architecture, dataset complexity, and training horizon.

4.3 Sentiment Analysis

This section presents the experimental evaluation of optimization algorithms on the sentiment analysis task, which represents a standard benchmark in natural language processing. Sentiment classification requires models to map textual sequences to discrete sentiment labels and therefore involves high-dimensional parameter spaces, non-convex objectives, and heterogeneous dataset characteristics. In contrast to the computer vision tasks examined earlier, the optimization dynamics in this setting are influenced not only by model architecture but also by the representation learning mechanisms used for textual data.

The experiments consider two widely used sentiment classification datasets, SST-2 and IMDB, together with two model architectures representing different modeling paradigms. The first architecture is an LSTM-based classifier that processes token sequences using recurrent neural networks, while the second architecture is based on the transformer model DistilBERT. This configuration enables a comparison between classical sequence modeling approaches and modern transformer-based language models.

Each training run is limited to 20 epochs. The batch size is fixed to 64 in most configurations, while the IMDB experiments with the LSTM model use batch size 16 because of the larger number of training samples.

Several regularization strategies are applied to stabilize training and mitigate overfitting. Label smoothing with coefficient 0.05 is used during training, and dropout with rate 0.5 is applied between the encoder and classification layer in the LSTM architecture. Weight decay values are chosen according to model type, with values between 10^{-5} and 10^{-4} used for recurrent models and 0.01 applied for DistilBERT following common transformer fine-tuning practices. Gradient clipping with threshold 1.0 is applied in all experiments to prevent gradient explosion. Learning rate scheduling is implemented using the `ReduceLROnPlateau` strategy, allowing the learning rate to adapt dynamically based on validation performance.

For sentiment analysis, the main evaluation criterion is classification accuracy, which quantifies the proportion of correctly predicted sentiment labels. This primary metric is complemented by validation loss, runtime, and convergence information so as to provide a broader view of optimization behavior beyond final predictive performance alone. The following commands summarize the experimental configurations used in the benchmark.

SST-2 – LSTM Sentiment Classifier

```
python -m benchmark.main
  --task sentiment_analysis --dataset sst2
  --model lstm_sentiment --optimizers all
  --epochs 20 --batch-size 64 --max-samples 30000 --num-workers 8
```

SST-2 – DistilBERT Sentiment Classifier

```
python -m benchmark.main
  --task sentiment_analysis --dataset sst2
  --model distilbert_sentiment --optimizers all --epochs 20
  --batch-size 64 --max-samples 10000 --num-workers 8
  --lr-overrides
```

IMDB – LSTM Sentiment Classifier

```
python -m benchmark.main
  --task sentiment_analysis --dataset imdb
  --model lstm_sentiment --optimizers all
  --epochs 20 --batch-size 16 --max-samples 115000 --num-workers 8
```

IMDB – DistilBERT Sentiment Classifier

```
python -m benchmark.main
  --task sentiment_analysis --dataset imdb
  --model distilbert_sentiment --optimizers all
  --epochs 20 --batch-size 64 --max-samples 90000
  --num-workers 8 --lr-overrides
```

Across sentiment classification experiments, adaptive and sharpness-aware optimizers show a consistent advantage. Across both datasets and architectures, sharpness-aware and adaptive optimizers occupy the top-performing region in terms of accuracy. In particular, SAM frequently achieves the highest accuracy values across multiple configurations, with GSAM typically producing similar results while occasionally improving convergence behavior or validation loss.

Adaptive optimizers also demonstrate strong and stable performance across all experiments. Methods such as Adam, AdamW, AdaFactor, and AdaBelief repeatedly appear among the best-performing configurations, achieving accuracy values close to the best sharpness-aware methods while maintaining more moderate computational overhead. These optimizers therefore provide a particularly favorable balance between predictive performance, training efficiency, and stability.

In contrast, SGD-based optimizers consistently produce lower final accuracy values in both datasets. On SST-2 and IMDB, these methods generally remain several percentage points below the strongest adaptive or sharpness-aware approaches. However, SGD variants often exhibit smoother validation trajectories and appear less sensitive to overfitting. This behavior suggests that their main disadvantage within the benchmark arises from slower convergence rather than fundamentally weaker optimization properties.

The comparison between architectures further highlights how model design influences optimization behavior. DistilBERT consistently achieves higher accuracy than the LSTM classifier across both datasets, reflecting the stronger representational capacity of transformer-based models. At the same time, the choice of optimizer has a more pronounced impact in the transformer setting, where advanced optimization strategies produce clearer improvements in predictive performance.

Another notable phenomenon observed across several experiments is the presence of overfitting. Many adaptive and sharpness-aware optimizers rapidly achieve very high training accuracy, particularly in the SST-2 experiments, but subsequently show increasing validation loss after a few epochs. By contrast, several SGD-based methods display more stable validation curves, suggesting greater resistance to overfitting despite lower final accuracy.

Table 4.3: Summary of sentiment analysis results

Dataset	Model	Top Optimizers	Accuracy	Loss	Runtime
IMDB	DistilBERT	**SAM**, GSAM, AdaFactor, Adam, AdaBelief	**0.883**	0.276	∼325 s
SST-2	DistilBERT	**SAM**, GSAM, AdamW, Adam, AdaFactor	**0.911**	0.253	∼783 s
IMDB	LSTM	**SAM**, GSAM, AdaFactor, Adam, AdaBelief	**0.874**	0.332	∼104 s
SST-2	LSTM	**SAM**, GSAM, AdaFactor, AdamW, Adam	**0.888**	0.242	∼176–287 s

Overall, the sentiment analysis experiments indicate that sharpness-aware optimizers provide the strongest peak performance, while adaptive methods offer the most balanced trade-off between accuracy, convergence speed, and runtime. SGD-based approaches remain comparatively robust but are disadvantaged by slower convergence within the constrained 20-epoch training horizon. Taken together, these results suggest that modern NLP architectures benefit most from adaptive and advanced optimization strategies under limited training budgets.

4.4 Machine Translation

This section presents the experimental evaluation of optimization algorithms on the machine translation task. Neural machine translation represents a particularly demanding learning problem, as models must generate coherent sequences in a large output space while capturing long-range dependencies between tokens. Compared to classification tasks, the optimization landscape is therefore significantly more complex, and training stability becomes an important factor in determining the effectiveness of optimization strategies.

The benchmark considers two bilingual translation datasets, a lightweight IWSLT14 setting and EuroParl, together with two sequence-to-sequence architectures. The first model is a recurrent encoder–decoder architecture based on Long Short-Term Memory (LSTM) networks, while the second is a Transformer-based model relying on self-attention mechanisms. This configuration allows the experiments to capture optimization behavior across both classical recurrent translation systems and modern attention-based architectures.

Each training run is limited to 20 epochs with a batch size of 64. The number of training samples is restricted through the `-max-samples` parameter in order to control runtime. Data loading is parallelized using eight workers, following the experimental protocol described in Chapter 3.

Learning rate scheduling is implemented using the `ReduceLROnPlateau` strategy, which adjusts the learning rate dynamically according to validation performance. Machine translation performance is evaluated primarily through the BLEU score, which reflects the degree of overlap between generated translations and their reference counterparts. To ensure that optimizer comparisons are not based solely on translation quality, validation loss, runtime, and convergence behavior are also taken into account. The following commands report the configurations used to reproduce the translation experiments.

IWSLT14 En-De – Transformer Seq2Seq

```
python -m benchmark.main
  --task machine_translation --dataset iwslt14_en_de
  --model transformer_seq2seq --optimizers all
  --epochs 20 --batch-size 64 --max-samples 10000 --num-workers 8
```

IWSLT14 En-De – LSTM Seq2Seq

```
python -m benchmark.main
  --task machine_translation --dataset iwslt14_en_de
  --model lstm_seq2seq --optimizers all
  --epochs 20 --batch-size 64 --max-samples 10000 --num-workers 8
```

EuroParl – Transformer Seq2Seq

```
python -m benchmark.main
  --task machine_translation --dataset europarl_bilingual
  --model transformer_seq2seq --optimizers all
  --epochs 20 --batch-size 64 --max-samples 30000 --num-workers 8
```

EuroParl – LSTM Seq2Seq

```
python -m benchmark.main
  --task machine_translation --dataset europarl_bilingual
  --model lstm_seq2seq --optimizers all
  --epochs 20 --batch-size 64 --max-samples 30000 --num-workers 8
```

The machine translation results reveal a strong architectural divide between recurrent and transformer-based sequence-to-sequence models. Across both datasets, the Transformer-based architecture consistently achieves substantially higher BLEU scores than the LSTM sequence-to-sequence model. On the EuroParl dataset, normalized BLEU scores approach the upper bound of the evaluation scale for the Transformer model, whereas the LSTM architecture reaches maximum scores around 0.18. A similar gap appears on the IWSLT14 dataset, where the Transformer reaches BLEU values above 0.59, while the LSTM model remains below 0.15. These results are consistent with the widely observed advantage of attention-based architectures in neural machine translation.

Although architectural capacity plays a dominant role in determining translation quality, the optimizer choice still significantly influences training behavior within each model class. In the LSTM-based experiments, adaptive optimizers consistently dominate the best-performing region of the BLEU spectrum. Methods such as Adam, RMSProp, RAdam, AdaBelief, and AdaFactor achieve the strongest translation quality on both datasets while maintaining relatively stable optimization trajectories. By contrast, SGD-based methods produce considerably lower BLEU scores together with higher validation loss values, indicating that recurrent translation models benefit strongly from adaptive learning-rate mechanisms.

Sharpness-aware optimization methods exhibit competitive BLEU scores in the LSTM configurations but incur substantially higher runtime due to the additional gradient evaluations required during each optimization step. As a result, these methods rarely provide a favorable trade-off between translation quality and computational cost within the constrained training horizon adopted in the benchmark.

The Transformer experiments display somewhat different optimization dynamics. Several adaptive optimizers achieve very strong peak BLEU scores, with AdaDelta and LAMB reaching the highest values on the EuroParl dataset. At the same time, optimizers such as Adam and RAdam often achieve more efficient training trajectories, converging more rapidly while maintaining competitive translation quality. These results suggest that in attention-based architectures the most relevant distinction between optimizers is not only the final BLEU score but also the balance between convergence speed and stability.

Another notable observation concerns training stability. In several Transformer experiments, certain optimizers exhibit rapid improvements in BLEU score during early epochs followed by later performance degradation. This behavior is particularly visible on the EuroParl dataset and indicates that evaluating optimizers solely based on maximum BLEU may not fully capture the underlying training dynamics. Some optimizers that achieve slightly lower peak scores nevertheless produce smoother and more stable learning trajectories.

The comparison between datasets further highlights the influence of data characteristics on optimization behavior. The EuroParl experiments generally produce higher BLEU values but also display more unstable learning curves, whereas the IWSLT14 experiments exhibit more moderate performance levels together with more stable convergence patterns. These differences suggest that dataset scale and linguistic diversity can significantly affect optimization dynamics in sequence-to-sequence models.

Table 4.4: Summary of machine translation results

Dataset	Model	Top Optimizers	BLEU	Loss	Runtime
EuroParl	Transformer	**AdaDelta**, LAMB, AdamW, SAM, Lion	**0.995**	∼1.96	∼50–762 s
IWSLT14	Transformer	**AdaDelta**, RAdam, Adam, Lion, LAMB	**0.591**	∼4.98	∼80–254 s
EuroParl	LSTM	**Adam**, RAdam, RMSProp, AdaBelief, AdaFactor	**0.179**	4.97	∼560–1500 s
IWSLT14	LSTM	**RMSProp**, AdaFactor, SAM, AdaBelief, AdamW	**0.149**	7.48	∼202–471 s

Overall, the machine translation experiments indicate that adaptive optimization methods perform particularly well across the evaluated architectures and datasets.

However, the results also emphasize that optimizer evaluation should consider not only final BLEU scores but also convergence behavior and training stability. Under the fixed 20-epoch training budget adopted in this benchmark, optimizers that combine strong performance with efficient convergence tend to provide the most practical advantage.

4.5 Question Answering

This section presents the experimental evaluation of optimization algorithms on the extractive question answering task. Compared to classification problems, extractive question answering requires models to predict the start and end positions of an answer span within a context passage. From an optimization perspective, this formulation introduces additional complexity, as models must simultaneously learn contextual representations and accurately localize answer boundaries within high-dimensional token sequences.

The benchmark considers two datasets, SQuAD v1 and TweetQA, together with two model architectures. The first model is a transformer-based question answering architecture, while the second is a recurrent model based on a BiLSTM encoder combined with an attention mechanism. This setup allows the experiments to capture optimization behavior across both modern transformer-based systems and more traditional sequence modeling approaches.

Each run is limited to 20 epochs in accordance with the global training budget defined in Chapter 3. Gradient clipping with threshold 1.0 is applied to stabilize training, and learning rate scheduling follows the `ReduceLROnPlateau` strategy. The number of training samples is constrained through the `-max-samples` parameter in order to maintain computational feasibility. The primary metric adopted for extractive question answering is the F1 score, as it captures the token-level overlap between predicted and ground-truth answer spans and therefore provides a more informative signal than exact matching alone. Exact Match, validation loss, runtime, and convergence behavior are also monitored to obtain a more complete picture of optimizer performance. The following commands summarize the reproducibility settings used for the evaluated configurations.

SQuAD v1 – Transformer QA Model

```
python -m benchmark.main
  --task question_answering --dataset squad_v1
  --model transformer_qa --optimizers all
  --epochs 20 --batch-size 32 --max-samples 20000 --num-workers 8
```

SQuAD v1 – BiLSTM with Attention

```
python -m benchmark.main
  --task question_answering --dataset squad_v1
  --model bilstm_attention_qa --optimizers all
  --epochs 20 --batch-size 64 --max-samples 20000 --num-workers 8
```

TweetQA – Transformer QA Model

```
python -m benchmark.main
  --task question_answering --dataset tweet_qa
  --model transformer_qa --optimizers all
  --epochs 20 --batch-size 32 --max-samples 8000 --num-workers 8
```

TweetQA – BiLSTM with Attention

```
python -m benchmark.main
  --task question_answering --dataset tweet_qa
  --model bilstm_attention_qa --optimizers all
  --epochs 20 --batch-size 64 --max-samples 8000 --num-workers 8
```

Question answering is one of the few tasks in the benchmark for which optimizer rankings remain remarkably stable across datasets and architectures. Adaptive optimizers dominate the top-performing region, with RMSProp repeatedly achieving the highest primary metric values across both datasets and architectures. Adam and AdaBelief follow closely in most experiments, providing similar performance while often achieving lower validation loss or faster convergence.

The stability of this ranking distinguishes the question answering task from several other benchmarks considered in this study. In contrast to segmentation or summarization tasks, where optimizer performance varies significantly with architecture or dataset complexity, the QA experiments consistently favor a small group of adaptive methods. This suggests that extractive question answering induces an optimization regime in which adaptive learning-rate mechanisms provide reliable and robust convergence.

The comparison between architectures further highlights the importance of representation capacity in this task. Transformer-based models consistently outperform the BiLSTM architecture across both datasets. On SQuAD and TweetQA, the transformer model achieves substantially higher F1 scores, reflecting the stronger contextual modeling capabilities of attention-based architectures. However, the transformer configurations also display a stronger tendency toward overfitting,

particularly on the TweetQA dataset.

Overfitting behavior is visible in several experiments where validation loss increases after only a few epochs despite stable or improving F1 scores. This phenomenon is especially pronounced for adaptive optimizers in the BiLSTM experiments on the SQuAD dataset. In contrast, SGD-based methods often produce smoother validation loss trajectories, suggesting greater resistance to overfitting. Nevertheless, this stability does not translate into superior final performance, as these methods consistently achieve lower F1 scores than the strongest adaptive baselines.

Table 4.5: Summary of question answering results

Dataset	Model	Top Optimizers	F1 Score	Loss	Runtime
SQuAD	BiLSTM+Attention	**RMSProp**, Adam, AdaBelief, AdaGrad, GSAM	**0.159**	7.43	∼45 s
TweetQA	BiLSTM+Attention	**RMSProp**, AdaBelief, Adam, GSAM, SAM	**0.462**	4.34	∼48 s
SQuAD	Transformer QA	**RMSProp**, AdaBelief, Adam, GSAM, SAM	**0.689**	2.05	∼241 s
TweetQA	Transformer QA	**RMSProp**, AdaBelief, Adam, GSAM, SAM	**0.781**	1.66	∼224 s

Sharpness-aware optimization methods such as SAM and GSAM occupy an intermediate position in the optimizer ranking. While they occasionally achieve competitive performance, they rarely surpass the best adaptive optimizers and incur higher runtime due to the additional gradient evaluations required at each training step. Their main advantage lies in producing relatively stable loss trajectories, which may be beneficial in settings where training stability is prioritized over peak performance.

Overall, the question answering experiments highlight a clear pattern in which adaptive optimizers provide the most reliable performance across both architectures and datasets. At the same time, the results illustrate an important trade-off between peak performance and stability. Methods that maximize the F1 score often exhibit stronger overfitting tendencies, while more stable optimizers may converge more slowly or achieve slightly lower final performance. Within the constrained 20-epoch training budget adopted in this benchmark, adaptive optimizers such as RMSProp, Adam, and AdaBelief provide a particularly effective balance between performance and stability in the evaluated configurations.

4.6 Named Entity Recognition

This section presents the experimental evaluation of optimization algorithms on the Named Entity Recognition (NER) task. NER is a structured prediction problem in which each token in an input sequence must be assigned a semantic label

corresponding to entity categories such as persons, locations, or organizations. Compared to standard classification tasks, this formulation introduces additional complexity because predictions are not independent: the model must learn dependencies between labels within the sequence.

The benchmark considers two widely used datasets, CoNLL-2003 and WikiANN (English), together with two neural architectures for sequence labeling. The first model is a conventional LSTM-CRF architecture, while the second extends the BiLSTM-CRF framework by incorporating a character-level convolutional network (CharCNN) to capture subword information. This configuration allows the experiments to examine how optimizer behavior interacts with both model capacity and linguistic representation mechanisms.

Each run is limited to 20 epochs under the shared training protocol defined in Chapter 3. Input sequences are padded and batched according to sequence length to improve efficiency. In the character-enhanced architecture, character-level embeddings are processed through a convolutional network before being combined with word-level representations.

Gradient clipping with threshold 1.0 is applied in all configurations to prevent instability during optimization. Learning rate scheduling follows the `ReduceLROnPlateau` strategy, allowing the learning rate to adapt dynamically according to validation performance. As in previous tasks, the number of training samples is limited using the `-max-samples` parameter in order to maintain tractable training times within the available computational resources.

For the NER task, optimizer performance is assessed mainly through the F1 score computed over predicted token labels, which summarizes the balance between precision and recall under the implemented evaluation procedure. This metric is considered together with validation loss, runtime, and convergence behavior in order to evaluate not only the final labeling quality but also the efficiency and stability of training. The following commands summarize the configurations adopted for reproducibility.

CoNLL-2003 – LSTM-CRF

```
python -m benchmark.main
  --task ner --dataset conll2003 --model lstm_crf_ner
  --optimizers all
  --epochs 20 --batch-size 64 --max-samples 8000 --num-workers 8
```

CoNLL-2003 – BiLSTM-CRF + CharCNN

```
python -m benchmark.main
  --task ner --dataset conll2003
  --model bilstm_crf_charcnn_ner --optimizers all
  --epochs 20 --batch-size 64 --max-samples 8000 --num-workers 8
```

WikiANN (English) – LSTM-CRF

```
python -m benchmark.main
  --task ner --dataset wikiann_en --model lstm_crf_ner
  --optimizers all
  --epochs 20 --batch-size 64 --max-samples 10000 --num-workers 8
```

WikiANN (English) – BiLSTM-CRF + CharCNN

```
python -m benchmark.main
  --task ner --dataset wikiann_en
  --model bilstm_crf_charcnn_ner --optimizers all
  --epochs 20 --batch-size 64 --max-samples 10000 --num-workers 8
```

The NER results suggest that optimizer behavior is shaped not only by dataset characteristics but also by the representational strength of the selected architecture. In many configurations, adaptive optimizers achieve the highest final F1 scores, although their advantage over other methods varies depending on the architectural setting.

The BiLSTM-CRF architecture augmented with character-level features consistently achieves the highest overall performance across both datasets. In these configurations, adaptive optimizers such as LAMB, Lion, AdaFactor, and AdaBelief frequently occupy the top-performing region of the F1 spectrum. These methods combine strong final performance with relatively efficient convergence behavior, indicating that adaptive learning-rate mechanisms are well suited for the structured optimization landscape of sequence labeling tasks.

The simpler LSTM-CRF architecture exhibits somewhat different optimization dynamics. In this configuration, SGD-based methods occasionally achieve competitive or even superior performance, as illustrated by the CoNLL-2003 experiments where SGD with Momentum obtains the highest F1 score. This behavior suggests that recurrent sequence labeling models may benefit from the more stable gradient updates provided by SGD-like optimizers, particularly when training dynamics evolve more gradually.

However, the analysis of training curves indicates that several SGD-based methods improve steadily throughout training without fully converging within the 20-epoch horizon. As a result, their relative performance disadvantage in some experiments may partly reflect the limited training budget rather than inherent optimization limitations. Adaptive methods, by contrast, tend to converge more rapidly and therefore appear stronger under short training schedules.

Sharpness-aware optimizers such as SAM and GSAM typically achieve intermediate performance levels across the NER experiments. While these methods sometimes approach the best-performing adaptive optimizers, they do not consistently surpass them and incur additional computational cost due to repeated gradient evaluations. Their advantage therefore remains limited within the constrained training setting adopted in the benchmark.

Table 4.6: Summary of named entity recognition results

Dataset	Model	Top Optimizers	F1 Score	Loss	Runtime
CoNLL-2003	BiLSTM-CRF CNN	**LAMB**, Lion, SAM, AdaFactor, AdaBelief	**0.673**	1.35	∼247 s
WikiANN	BiLSTM-CRF CNN	**LAMB**, Lion, AdaFactor, AdaGrad, AdamW	**0.613**	≈0.00	∼288 s
CoNLL-2003	LSTM-CRF	**SGD with Momentum**, AdaBelief, AdaDelta, SGD, LARS	**0.402**	1.97	∼318 s
WikiANN	LSTM-CRF	**LAMB**, RAdam, Lion, Adam, AdaFactor	**0.568**	0.00113	∼74 s

Another important observation concerns training stability. Several adaptive optimizers display strong initial improvements followed by gradual performance degradation during later epochs. In contrast, many SGD-based methods exhibit smoother and more monotonic learning trajectories. This suggests that, although adaptive methods frequently achieve the highest peak F1 scores, their optimization trajectories may be less stable over longer training horizons.

Overall, the NER experiments highlight the importance of considering both convergence dynamics and architectural capacity when evaluating optimization strategies. The stronger BiLSTM-CRF architecture consistently achieves higher performance across datasets, while optimizer choice primarily influences convergence speed and training stability. Within the 20-epoch training horizon adopted in this benchmark, adaptive optimizers achieve the strongest final results in most evaluated configurations, although more stable methods such as SGD variants might close part of the performance gap under longer training schedules.

4.7 Text Generation

This section presents the experimental evaluation of optimization algorithms on the text generation task in the form of autoregressive language modeling. In this setting, models are trained to predict the next token in a sequence given the preceding context. Compared to standard classification problems, language modeling introduces a more complex optimization landscape because the model must learn long-range dependencies while operating in a very large output space.

The benchmark considers two commonly used language modeling datasets, Penn Treebank (PTB) and WikiText-2, together with two model architectures representing different modeling paradigms. The first architecture is a transformer-based autoregressive model (GPT-small), while the second is a recurrent language model based on gated recurrent units (GRU). This configuration enables the analysis of optimization behavior across both modern attention-based architectures and classical recurrent sequence models.

Each training run is limited to 20 epochs with a batch size of 64. The number of training samples is restricted using the `-max-samples` parameter to control runtime. Data loading is parallelized with eight workers, consistent with the experimental protocol defined in Chapter 3.

In autoregressive language modeling, the main evaluation criterion is perplexity, which measures the uncertainty of the model over the predicted token distribution and therefore directly reflects language modeling quality. Since perplexity alone does not fully describe optimization behavior, validation loss, runtime, and convergence are also examined throughout the experiments. The following commands report the configurations used to reproduce the evaluated runs.

WikiText-2 – GPT-small

```
python -m benchmark.main
  --task text_generation --dataset wikitext2 --model gpt_small
  --optimizers all
  --epochs 20 --batch-size 64 --max-samples 15000 --num-workers 8
```

WikiText-2 – GRU Language Model

```
python -m benchmark.main
  --task text_generation --dataset wikitext2 --model gru_lm
  --optimizers all
  --epochs 20 --batch-size 64 --max-samples 15000 --num-workers 8
```

PTB – GPT-small

```
python -m benchmark.main
  --task text_generation --dataset ptb_text_only --model gpt_small
  --optimizers all
  --epochs 20 --batch-size 64 --max-samples 50000 --num-workers 8
```

PTB – GRU Language Model

```
python -m benchmark.main
  --task text_generation --dataset ptb_text_only --model gru_lm
  --optimizers all
  --epochs 20 --batch-size 64 --max-samples 50000 --num-workers 8
```

Across all text generation experiments, adaptive optimizers dominate the strongest region of the perplexity spectrum. Optimizers such as RMSProp, Adam, RAdam, LAMB, and AdaBelief repeatedly appear among the best-performing methods on both datasets and architectures. In contrast, SGD-based methods generally produce higher perplexity values and slower improvement in validation loss, indicating that language modeling strongly benefits from adaptive learning-rate mechanisms.

The comparison between architectures reveals notable differences in optimization behavior. The transformer-based GPT model exhibits more complex and less stable learning dynamics than the GRU language model. Several optimizers demonstrate rapid improvements during the first few epochs of GPT training followed by signs of overfitting or performance degradation. This behavior is particularly visible for certain adaptive optimizers and sharpness-aware methods. In contrast, the GRU architecture typically produces smoother and more stable optimization trajectories.

Sharpness-aware optimization methods show an interesting trade-off in this task. In the GPT experiments, SAM and GSAM frequently converge substantially faster than most adaptive optimizers and achieve lower runtime. However, their final perplexity values tend to remain slightly worse than those of the strongest adaptive baselines. As a result, these methods provide improved optimization efficiency but do not consistently achieve the best predictive performance.

Dataset characteristics also influence optimization behavior. The PTB experiments generally produce lower perplexity values compared to WikiText-2, reflecting the smaller vocabulary and simpler linguistic structure of the dataset. However, the relative ranking of optimizers remains largely consistent across datasets, suggesting

that the fundamental optimization dynamics of language modeling are determined primarily by model architecture and training strategy rather than dataset-specific properties.

Table 4.7: Summary of text generation results

Dataset	Model	Top Optimizers	Perplexity	Loss	Runtime
PTB	GPT-small	**RMSProp**, Adam, RAdam, LAMB, GSAM	**2.19**	5.13	∼275 s
WikiText-2	GPT-small	**RMSProp**, Adam, RAdam, AdaGrad, LAMB	**2.21**	5.66	∼431 s
PTB	GRU LM	**LAMB**, GSAM, RMSProp, AdaBelief, AdamW	**2.18**	5.12	∼305 s
WikiText-2	GRU LM	**LAMB**, GSAM, RMSProp, AdaBelief, AdamW	**2.20**	5.76	∼387 s

Overall, the text generation experiments highlight the strong performance of adaptive optimization methods in the evaluated sequence modeling settings. Across both datasets and architectures, adaptive optimizers consistently provide the strongest balance between convergence speed, stability, and final perplexity. Sharpness-aware methods offer faster convergence but slightly weaker final performance, while SGD-based optimizers remain disadvantaged within the limited 20-epoch training horizon adopted in this benchmark.

4.8 Text Summarization

This section presents the experimental evaluation of optimization algorithms on the text summarization task. Abstractive summarization is formulated as a sequence-to-sequence generation problem in which the model must produce a concise textual summary of an input document. Compared to classification or span prediction tasks, summarization introduces additional complexity because the output consists of variable-length token sequences that must remain semantically consistent with the source text.

From an optimization perspective, this setting produces high-dimensional non-convex objectives in which models must simultaneously learn contextual encoding, content selection, and fluent text generation. The benchmark considers two datasets, CNN/DailyMail and AESLC, together with two encoder–decoder architectures. The first model is a lightweight BART-based architecture, while the second is a compact transformer sequence-to-sequence model. This configuration allows the analysis to capture optimization behavior across models with substantially different representational capacity.

Each training run is limited to 20 epochs under the shared training protocol

described in Chapter 3. Input preprocessing follows the standard sequence-to-sequence formulation with sequence padding and teacher forcing during training. The training objective is cross-entropy loss with padding tokens ignored and label smoothing set to 0.1. Gradient clipping with threshold 1.0 is applied to stabilize training, and learning rate scheduling follows the `ReduceLROnPlateau` strategy.

Text summarization performance is evaluated primarily through ROUGE-L, which measures the structural overlap between generated summaries and reference summaries through longest-common-subsequence matching. This primary metric is complemented by validation loss, runtime, and convergence behavior so that optimizer comparisons reflect both summary quality and training dynamics. The following commands summarize the reproducibility settings for the evaluated configurations.

CNN/DailyMail – BART Small

```
python -m benchmark.main
  --task text_summarization --dataset cnn_dailymail
  --model bart_small --optimizers all
  --epochs 20 --batch-size 16 --max-samples 8000 --num-workers 8
```

CNN/DailyMail – Tiny Transformer Seq2Seq

```
python -m benchmark.main
  --task text_summarization --dataset cnn_dailymail
  --model tiny_transformer_seq2seq --optimizers all
  --epochs 20 --batch-size 32 --max-samples 10000 --num-workers 8
```

AESLC – BART Small

```
python -m benchmark.main
  --task text_summarization --dataset aeslc
  --model bart_small --optimizers all
  --epochs 20 --batch-size 16 --max-samples 10000 --num-workers 8
```

AESLC – Tiny Transformer Seq2Seq

```
python -m benchmark.main
  --task text_summarization --dataset aeslc
  --model tiny_transformer_seq2seq --optimizers all
  --epochs 20 --batch-size 32 --max-samples 10000 --num-workers 8
```

The summarization experiments are dominated by architectural effects, with model capacity playing a central role in determining optimizer outcomes. Across both datasets, the BART-based architecture consistently achieves substantially higher ROUGE scores than the Tiny Transformer model. On the AESLC dataset, BART reaches a primary metric close to 1.0, whereas the Tiny Transformer remains well below this level even under its best optimizer configurations. This difference indicates that model capacity plays a major role in abstractive summarization tasks under the training budget considered here.

Within the BART experiments, adaptive optimization methods dominate the most reliable region of the performance spectrum. Optimizers such as LAMB, AdaDelta, AdamW, and RMSProp frequently achieve the highest ROUGE scores while maintaining relatively low validation loss. These results indicate that adaptive learning-rate mechanisms are particularly effective when training encoder–decoder architectures for text generation.

The Tiny Transformer experiments exhibit a different optimization landscape. Several optimizers, including Yogi and SGD-based methods, occasionally achieve very high ROUGE values during the first epochs of training. However, analysis of the learning curves shows that these results often correspond to transient performance spikes followed by rapid degradation. Consequently, the highest reported metric values do not always correspond to stable or reliable optimization behavior.

More stable optimizers such as Adam, RAdam, or AdaGrad typically produce smoother learning trajectories even if their peak ROUGE scores are slightly lower. These methods therefore provide more consistent performance over the full training process. The results illustrate an important limitation of evaluating optimization algorithms solely on the basis of maximum metric values, particularly in sequence generation tasks where early fluctuations may not reflect long-term model quality.

Sharpness-aware optimization methods again occupy an intermediate position in the optimizer ranking. While SAM and GSAM occasionally achieve competitive ROUGE scores, they rarely outperform the strongest adaptive optimizers and incur additional computational cost due to repeated gradient evaluations. Their advantage therefore remains limited within the constrained training horizon adopted in this benchmark.

Dataset characteristics also influence optimization behavior. The AESLC dataset generally allows models to achieve higher ROUGE scores and faster convergence, suggesting that it represents a comparatively easier summarization setting. In contrast, the CNN/DailyMail dataset produces lower overall performance and

requires more stable optimization trajectories to achieve consistent improvements.

Table 4.8: Summary of text summarization results

Dataset	Model	Top Optimizers	ROUGE	Loss	Runtime
AESLC	BART Small	**LAMB**, SGD with Momentum, AdaDelta, Lion, GSAM	**0.992**	1.30	∼27 s
CNN/DailyMail	BART Small	**AdamW**, AdaDelta, RMSProp, AdaBelief, Adam	**0.843**	2.02	∼123 s
AESLC	Tiny Transformer	**Yogi**, SGD, LARS, SGD with Momentum, AdaGrad	**0.751**	5.20	<10 s
CNN/DailyMail	Tiny Transformer	**Yogi**, SGD, LARS, AdaDelta, RAdam	**0.575**	6.23	∼128 s

Overall, the summarization experiments highlight the strong interaction between model capacity and optimization strategy. Larger encoder–decoder architectures such as BART consistently outperform smaller models regardless of the chosen optimizer. Within this architectural context, adaptive optimizers provide the most reliable performance, while methods that produce early performance spikes may not maintain stable learning dynamics throughout training. These observations highlight the importance of evaluating optimization algorithms not only in terms of peak performance but also in terms of stability and consistency throughout the training process.

4.9 Summary of Empirical Observations

Across the diverse set of benchmark tasks considered in this chapter, several consistent patterns emerge. Adaptive optimization methods generally achieve the strongest final performance across most tasks, particularly in natural language processing and sequence modeling settings. Sharpness-aware methods occasionally match or approach the best results but typically incur higher computational cost due to additional gradient evaluations. In contrast, SGD-based optimizers often exhibit smoother training dynamics and improved stability but tend to converge more slowly within the constrained 20-epoch training budget adopted in this benchmark.

Another important observation concerns the strong interaction between optimization strategies and model architecture. Modern transformer-based architectures consistently benefit from adaptive learning-rate mechanisms, whereas certain convolutional and recurrent models occasionally remain competitive with SGD-based methods. These findings highlight that optimizer effectiveness depends not only on the algorithm itself but also on the learning task, the model architecture, and the available training horizon.

Table 4.9 provides a consolidated cross-task overview of the top-performing optimizer family in each evaluated configuration. Each cell indicates the optimizer family that achieved the strongest primary metric in that task–architecture combination, providing a structured basis for the cross-task interpretation developed in the following chapter. To improve readability, the entries in Table 4.9 follow a unified notation convention. The label **Adaptive** refers to any optimizer belonging to the adaptive learning-rate family. The label **GD** refers to the broader SGD-based family, while the label **SA** refers to sharpness-aware methods. Where a single optimizer is particularly dominant within its family, its name is indicated in parentheses for precision. Bold formatting is used to highlight configurations in which the winning family deviates from the overall dominant pattern of adaptive optimizers, drawing attention to the empirically meaningful exceptions that motivate the cross-task discussion in Chapter 5.

Table 4.9: Cross-task summary of best-performing optimizer families

Task	Dataset	Lightweight Model	Stronger Model
Image Classification	MNIST	Adaptive	Adaptive
	CIFAR-10	Adaptive	Adaptive
Semantic Segmentation	Oxford Pet	Adaptive (RAdam)	**GD**
	Pascal VOC	Adaptive (LAMB)	Adaptive (LAMB)
Sentiment Analysis	IMDB	**SA**	**SA**
	SST-2	**SA**	**SA**
Machine Translation	EuroParl	Adaptive	Adaptive
	IWSLT14	Adaptive	Adaptive
Question Answering	SQuAD	Adaptive (RMSProp)	Adaptive (RMSProp)
	TweetQA	Adaptive (RMSProp)	Adaptive (RMSProp)
Named Entity Recog.	CoNLL-2003	**GD**	Adaptive
	WikiANN	Adaptive	Adaptive
Text Generation	PTB	Adaptive (RMSProp)	Adaptive (LAMB)
	WikiText-2	Adaptive (RMSProp)	Adaptive (LAMB)
Text Summarization	AESLC	Adaptive (Yogi)	Adaptive (LAMB)
	CNN/DM	Adaptive (AdamW)	Adaptive (AdamW)

The table makes visible two patterns that are easy to miss when reading task sections individually. Adaptive methods dominate the majority of configurations, but the exceptions are meaningful: SGD-based optimizers prove superior for DeepLabV3 segmentation and LSTM-CRF named entity recognition, while sharpness-aware methods hold a consistent advantage exclusively in sentiment analysis. These exceptions, rather than the dominant trend, constitute the most practically informative findings of the benchmark and motivate the detailed cross-task analysis presented in Chapter 5.

Chapter 5

Discussion

The experimental evaluation presented in the previous chapter investigated the behavior of several optimization algorithms across a diverse set of deep learning tasks. By combining multiple datasets with different model architectures, the benchmark provides a heterogeneous yet controlled setting for examining how optimization strategies influence convergence dynamics, computational efficiency, and final task performance.

Rather than focusing on individual experimental outcomes, this chapter synthesizes the broader patterns that emerge across tasks and models. The objective is to identify common behaviors among optimizer families and to interpret the empirical results in light of the theoretical properties discussed in earlier chapters. Special attention is devoted to findings that challenge prevailing assumptions in the optimizer literature, as these departures from expected behavior are often more informative than results that merely confirm established intuitions.

In particular, the discussion addresses four key questions: what cross-task conclusions about optimizer families can be drawn from the experiments; under which conditions certain optimizers should be preferred; which behavioral patterns remain consistent across different tasks; and which observations appear to be strongly dependent on the specific learning setting.

5.1 The Cross-Domain Strength of RMSProp

A first and notable finding of the benchmark is the consistent cross-domain competitiveness of RMSProp. Across question answering, text generation, and machine translation, RMSProp repeatedly ranked among the top-performing optimizers, often matching or exceeding Adam and its variants. This cross-domain robustness

is noteworthy given that Adam has become the implicit default in modern deep learning practice, sometimes without empirical justification specific to the task at hand.

RMSProp adapts learning rates using a running average of squared gradients without accumulating a first-moment estimate or applying bias correction. This simpler adaptation mechanism may confer greater stability in settings where gradient distributions are irregular or where momentum accumulation in Adam leads to overshooting. The empirical results here are consistent with theoretical observations by Reddi et al., who identify conditions under which Adam's convergence behavior is weaker than simpler adaptive methods [**10**]. Across the question answering experiments in particular, RMSProp reached the highest F1 scores on both the SQuAD and TweetQA benchmarks under both BiLSTM and Transformer architectures, suggesting that its adaptation mechanism is well-suited to the structured and sparse gradient signals produced by span-prediction objectives.

These results suggest that the prevailing dominance of Adam in practice may partly reflect convention rather than consistent empirical superiority. RMSProp constitutes a robust and computationally efficient alternative that practitioners would benefit from including as a strong baseline in optimizer selection decisions.

5.2 Violations of a Standard Dichotomy

A widely held assumption in the optimizer literature is that SGD with Momentum constitutes the method of choice for computer vision tasks involving convolutional architectures, while adaptive methods are preferred for natural language processing and sequence modeling. The results of this benchmark provide only partial support for this narrative, and document several systematic exceptions that merit careful consideration.

On the vision side, the segmentation experiments reveal a more complex picture. While SGD with Momentum achieved the highest primary metric on the Oxford-IIIT Pet dataset with DeepLabV3 (0.910), the advantage did not hold consistently. On U-Net architectures, adaptive optimizers such as RAdam and AdaBelief were the strongest performers, suggesting that the preference for SGD in vision is architecture-dependent rather than universal. The classical association between SGD and vision performance appears to be most valid for deeper, well-conditioned convolutional networks with residual connections, rather than for encoder-decoder architectures more generally.

On the NLP side, a more striking deviation from expectations emerges. In the CoNLL-2003 named entity recognition experiments with the LSTM-CRF architecture, SGD with Momentum achieved the highest F1 score, outperforming all adaptive optimizers in that configuration. This is a counter-intuitive result, given that NER is precisely the kind of structured, high-dimensional sequence labeling task where adaptive learning-rate methods are expected to hold the clearest advantage. One plausible explanation is that the CRF decoding layer introduces a strongly structured optimization surface in which the more controlled, non-adaptive updates of SGD navigate more effectively, avoiding the instability that adaptive methods can exhibit when interacting with learned transition parameters.

Taken together, these observations suggest that the SGD-vision / adaptive-NLP dichotomy is a useful heuristic but an unreliable rule. The interaction between optimizer design and model architecture appears to be the more fundamental determinant of performance, and this interaction cannot be predicted from task domain alone.

5.3 Simple Optimizers Match Complex Ones

A recurring and practically significant observation across the benchmark is the competitive performance of algorithmically simpler optimization methods relative to more sophisticated alternatives. Across multiple tasks, RMSProp, SGD with Momentum, and standard Adam matched or exceeded the performance of more recently proposed variants with additional adaptive components, including AdaBelief, Yogi, and RAdam.

This finding is particularly visible in the text summarization experiments, where on the Tiny Transformer architecture, simple SGD-based methods including SGD and LARS achieved high ROUGE scores comparable to or exceeding more complex adaptive methods. A similar pattern appears in the text generation experiments, where RMSProp and Adam consistently ranked at the top, offering little empirical motivation for the added algorithmic complexity of newer variants.

Each additional component in an optimizer introduces tunable hyperparameters, increases per-iteration computational overhead, and adds implementation complexity. When simpler methods achieve equivalent or better performance, the marginal value of this complexity must be justified on other grounds. These results suggest that the proliferation of Adam variants in recent years may reflect overfitting to specific benchmarks or compute regimes rather than general performance improvements across diverse training settings.

This has an important methodological implication: empirical optimizer evaluations that omit strong simple baselines risk systematically overstating the advances represented by newer methods. The design of the present benchmark, which retains SGD and RMSProp as explicit comparison points throughout, allows this effect to be directly observed.

5.4 Behavior of Sharpness-Aware Methods

Sharpness-aware optimization methods, including SAM and its variant GSAM, aim to improve generalization by explicitly seeking flatter minima of the loss landscape. Across the benchmark, these methods achieved their clearest advantage in sentiment analysis, where both SAM and GSAM consistently ranked first across all four evaluated configurations (SST-2 and IMDB, under both LSTM and DistilBERT architectures). The stability of this result across architectures and datasets suggests that sharpness-aware updates are particularly well-suited to the noisy, high-variance gradient environment of sentiment classification, where the training signal can fluctuate substantially across batches of short textual inputs.

However, the primary limitation of these methods is their computational overhead. SAM and GSAM require two forward-backward passes per update step, effectively doubling per-iteration cost relative to standard first-order methods. In the semantic segmentation experiments, this resulted in training runtimes of up to 1070 s for SAM on Pascal VOC with U-Net, compared to 356 s for standard SGD, which represents a threefold increase in cost for marginal performance gains. Across computationally intensive tasks, sharpness-aware methods rarely provided a favorable cost-benefit ratio within the 20-epoch training horizon.

The practical implication is clear: SAM and GSAM are best suited to settings where generalization is the primary concern, the training budget is abundant, and the gradient landscape is sufficiently noisy to benefit from sharpness-aware perturbations. Under the realistic low-budget conditions of this benchmark, their applicability is limited to tasks such as sentiment analysis where the performance gain justifies the additional cost.

5.5 Convergence Speed and Efficiency

A consistent finding across tasks is the decoupling between convergence speed measured in training epochs and convergence speed measured in wall-clock time. Adaptive optimizers, particularly RMSProp and Adam, consistently exhibited

faster reduction of training loss per epoch during the early phases of optimization. However, this epoch-efficiency advantage was partially or fully offset in wall-clock terms by the higher per-iteration computational cost of more complex update rules.

This decoupling is practically important. Benchmarks that evaluate convergence solely in terms of epochs systematically favor adaptive methods and may overstate their practical advantages. Conversely, benchmarks that evaluate only final performance after a fixed wall-clock budget may disadvantage methods that require more epochs to reach their characteristic behavior. The dual perspective adopted in this benchmark, reporting both epoch-level convergence and cumulative runtime, reveals that the most epoch-efficient optimizer is not always the most time-efficient.

Lion deserves specific attention in this context. Across text generation and translation tasks, Lion demonstrated a favorable combination of competitive task performance and low per-iteration runtime, reflecting the computational efficiency of its sign-based update rule. In the segmentation experiments, Lion appeared among the top performers for U-Net on Oxford-IIIT Pet with a runtime substantially lower than SAM. Its efficiency profile makes it a practically relevant choice for settings where wall-clock training time is the binding constraint.

5.6 Generalization Behavior and Overfitting

Beyond convergence speed, the benchmark reveals systematic differences in generalization behavior across optimizer families. In several experiments, adaptive optimizers achieved lower training losses without corresponding improvements in validation metrics, a pattern most visible in the LSTM-based question answering experiments on SQuAD, where several adaptive methods exhibited increasing validation loss after early epochs despite stable or improving F1 scores.

SGD-based optimizers, by contrast, exhibited smoother validation trajectories in multiple tasks, consistent with the theoretical hypothesis that the stochasticity and global learning rate of SGD updates provide an implicit regularization effect. This is most directly observed in the NER experiments, where the LSTM-CRF model trained with SGD with Momentum not only achieved the highest F1 score but did so with a more monotonically improving learning curve than competing adaptive methods.

The text summarization experiments provide a complementary illustration. For the Tiny Transformer model, several optimizers produced high peak ROUGE scores

during early epochs followed by rapid degradation. More stable optimizers, including Adam and RAdam, produced lower peak scores but maintained consistent improvement. This pattern highlights an important limitation of evaluating optimization algorithms solely on maximum metric values: peak performance does not always reflect reliable optimization behavior, particularly in sequence generation tasks where early learning curve dynamics can be misleading.

Overall, the experiments reveal a complex and task-dependent relationship between optimization dynamics and generalization. The ability to minimize training loss efficiently must be balanced with an optimizer's tendency to find flat versus sharp minima and its interaction with other regularization mechanisms such as dropout and weight decay.

5.7 Key Empirical Findings

The experimental results across the benchmark yield five consolidated empirical findings that extend and, in some cases, challenge prevailing assumptions in the optimizer literature.

First, RMSProp emerges as the most consistently strong optimizer across the evaluated tasks. Its cross-domain robustness, spanning question answering, text generation, and translation, suggests that its simpler adaptation mechanism provides reliable convergence across a wider range of gradient distributions than commonly assumed.

Second, the conventional SGD-vision / adaptive-NLP dichotomy receives only partial empirical support. SGD with Momentum outperformed all adaptive methods on the CoNLL-2003 NER task with the LSTM-CRF architecture, while adaptive optimizers proved superior for U-Net segmentation. Architecture and loss structure, rather than task domain alone, appear to be the more fundamental determinants of optimizer suitability.

Third, simpler optimizers consistently matched or exceeded more algorithmically complex alternatives. Across text summarization and generation, RMSProp and standard Adam provided as strong or stronger results than recent Adam variants, raising questions about the practical value of added algorithmic complexity under realistic training budgets.

Fourth, sharpness-aware methods offer a genuine performance advantage in noisy,

high-variance training regimes such as sentiment analysis, but introduce computational overhead that is rarely justified in computationally intensive tasks under constrained budgets.

Fifth, optimizer effectiveness is inseparable from its computational cost profile. The decoupling between epoch-level convergence and wall-clock efficiency, which was most strikingly illustrated by SAM's runtime in segmentation, means that optimizer selection must account for per-iteration cost, not only final metric values.

Taken together, these findings emphasize that optimizer selection requires careful consideration of the specific training context. No single method consistently dominates across tasks, architectures, and computational budgets. This cross-task interpretation provides the foundation for the concluding remarks presented in the final chapter.

Chapter 6

Conclusion

This thesis investigated the behavior of modern optimization algorithms through a systematic empirical benchmark covering a broad range of deep learning tasks. The study evaluated multiple optimizer families across computer vision and natural language processing problems, including image classification, semantic segmentation, sentiment analysis, named entity recognition, question answering, text summarization, language modeling, and machine translation. By combining different datasets and model architectures within a unified and reproducible experimental framework, the benchmark provided a structured environment for analyzing how optimization strategies influence convergence dynamics, computational efficiency, and generalization performance.

The main contribution of this work lies in the construction of a multi-domain benchmark for evaluating contemporary optimization algorithms under realistic, resource-constrained conditions. The experimental design deliberately reflects the constraints faced by most practitioners: a fixed training horizon, standardized hyperparameter configurations drawn from optimizer publications, and a heterogeneous set of tasks spanning diverse modalities and architectures. This framing ensures that the findings are not only internally valid but also practically relevant to researchers and practitioners who must select optimization strategies under similar conditions.

6.1 Summary of Findings

The benchmark yields several empirically grounded conclusions that both confirm and challenge prevailing assumptions in the optimizer literature. The most prominent finding is the consistent cross-domain strength of RMSProp, which ranked among the top-performing optimizers across question answering, text generation,

and machine translation. This result is notable given the near-universal adoption of Adam as the default optimizer in modern deep learning, and suggests that RMSProp constitutes a robust and underutilized alternative whose simpler adaptation mechanism may provide more reliable convergence across a broader range of gradient distributions.

A second key finding concerns the limits of the SGD-vision / adaptive-NLP dichotomy. While SGD with Momentum remained highly competitive in several vision tasks, achieving the highest segmentation score on Oxford-IIIT Pet with DeepLabV3, it also outperformed all adaptive methods on the CoNLL-2003 NER task with the LSTM-CRF architecture. Conversely, adaptive optimizers dominated U-Net segmentation. These results indicate that architecture and loss structure are more reliable predictors of optimizer suitability than task domain alone, and that the classical dichotomy should be treated as a heuristic rather than a prescriptive rule.

Third, the benchmark provides direct empirical evidence that algorithmically simpler optimizers frequently match or exceed more complex alternatives. Across text summarization and generation, RMSProp and standard Adam performed at least as well as recently proposed adaptive variants such as AdaBelief, Yogi, and RAdam. Under realistic training budgets, the marginal benefit of added algorithmic complexity, does not consistently translate into measurable performance gains.

Fourth, sharpness-aware methods demonstrate a genuine advantage in high-variance training regimes. SAM and GSAM consistently ranked first in sentiment analysis across all four evaluated configurations. However, their computational overhead limited their practical utility in computationally intensive settings. Their applicability under constrained budgets is therefore task-dependent rather than general.

Finally, the benchmark highlights a systematic decoupling between epoch-level convergence and wall-clock efficiency. Adaptive optimizers frequently required fewer epochs to reach competitive performance, but this advantage was often offset by higher per-iteration cost. Optimizer selection that ignores computational efficiency risks overestimating the practical value of methods that are fast in epochs but slow in real time.

6.2 Limitations

Despite the breadth of the experimental evaluation, several limitations of this study should be acknowledged. First, the benchmark was conducted under a

constrained training horizon of 20 epochs. While this choice reflects realistic resource conditions and enables controlled comparisons, longer training schedules may reveal different convergence dynamics, particularly for SGD-based methods whose generalization advantage often emerges in later training stages. Second, the benchmark relied on moderately sized datasets and architectures. Optimizer behavior may differ substantially in large-scale settings involving deeper models, larger vocabularies, or more data-intensive training regimes. Third, the study did not perform extensive per-optimizer hyperparameter searches. While this was a deliberate choice to preserve fairness and ecological validity, certain methods may benefit disproportionately from targeted tuning.

6.3 Future Directions

Future work could extend the present framework in several directions. Evaluating optimizers under longer training schedules would allow a more complete characterization of late-stage convergence and generalization dynamics, particularly the degree to which SGD-based methods close the performance gap with adaptive alternatives over extended optimization trajectories. Scaling the benchmark to larger architectures and datasets would test whether the observed trends generalize to more demanding training regimes. A systematic hyperparameter sensitivity analysis, conducted under controlled conditions, would clarify which optimizers are most robust to configuration choices and which are most sensitive. Finally, incorporating recently proposed optimizers and additional task domains would further strengthen the benchmark's scope and practical relevance.

In conclusion, this thesis contributes to the empirical understanding of optimization algorithms in deep learning by providing a comparative benchmark across multiple tasks, datasets, and model architectures under realistic training conditions. The results demonstrate that optimizer performance is inherently context-dependent: task structure, model architecture, convergence behavior, and computational constraints jointly shape practical effectiveness. The consistent strength of RMSProp across diverse settings, the architecture-dependence of the SGD-adaptive divide, and the limited practical gain of complex optimizer variants under constrained budgets are findings with direct implications for how optimization algorithms are selected and evaluated in modern deep learning practice. For this reason, selecting an optimizer requires not only theoretical understanding but also careful empirical assessment of the specific setting in which the model is trained.

Bibliography

[1] Matthew Hoy. «Alexa, Siri, Cortana, and More: An Introduction to Voice Assistants». In: *Medical Reference Services Quarterly* 37 (Jan. 2018), pp. 81–88. DOI: 10.1080/02763869.2018.1404391 (cit. on p. 1).

[2] Leon Bottou, Frank E. Curtis, and Jorge Nocedal. *Optimization Methods for Large-Scale Machine Learning*. 2018. arXiv: 1606.04838 [stat.ML]. URL: https://arxiv.org/abs/1606.04838 (cit. on p. 10).

[3] Augustin-Louis Cauchy. *Methode Generale pour la Resolution des Systemes d'Equations Simultanees*. 1847, pp. 536–538 (cit. on p. 11).

[4] Herbert E. Robbins. «*A Stochastic Approximation Method*». In: *Annals of Mathematical Statistics* 22 (1951), pp. 400–407. URL: https://api.semanticscholar.org/CorpusID:16945044 (cit. on p. 11).

[5] Aurelien Geron. *Hands-on Machine Learning with Scikit-Learn, Keras and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly, 2017, pp. 400–407. ISBN: 9781492032649 (cit. on p. 13).

[6] B. T. Polyak. *Some Methods of Speeding up the Convergence of Iteration Methods*. 1964, pp. 1–17 (cit. on p. 14).

[7] Yang You, Igor Gitman, and Boris Ginsburg. *Large Batch Training of Convolutional Networks*. 2017. arXiv: 1708.03888 [cs.CV]. URL: https://arxiv.org/abs/1708.03888 (cit. on p. 17).

[8] Geoffrey Hinton. *Neural Networks for Machine Learning - Lecture 6*. 2012. URL: https://www.coursera.org/learn/neural-networks (cit. on p. 19).

[9] Diederik P. Kingma and Jimmy Ba. «*Adam: A Method for Stochastic Optimization*». In: *arXiv* (2017). ISSN: 1412.6980 (cit. on p. 21).

[10] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. «*On the Convergence of Adam and Beyond*». In: *arXiv* (2018). ISSN: 1904.09237 (cit. on pp. 24, 108).

[11] Yijun Wang, Pengyu Zhou, and Wenya Zhong. «*An Optimization Strategy Based on Hybrid Algorithm of Adam and SGD*». In: *Proceedings of the 2018 International Conference on Algorithms, Computing and Artificial Intelligence* (2018) (cit. on p. 24).

[12] 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 (cit. on p. 24).

[13] Antoine Godichon-Baggioni, Wei Lu, and Bruno Portier. *A Full Adagrad Algorithm with O(Nd) Operations*. 2025. arXiv: 2405.01908 [math.ST]. URL: https://arxiv.org/abs/2405.01908 (cit. on p. 24).

[14] Matthew D. Zeiler. *ADADELTA: An Adaptive Learning Rate Method*. 2012. arXiv: 1212.5701 [cs.LG]. URL: https://arxiv.org/abs/1212.5701 (cit. on p. 25).

[15] Noam Shazeer and Mitchell Stern. *Adafactor: Adaptive Learning Rates with Sublinear Memory Cost*. 2018. arXiv: 1804.04235 [cs.LG]. URL: https://arxiv.org/abs/1804.04235 (cit. on p. 26).

[16] Juntang Zhuang, Tommy Tang, Yifan Ding, Sekhar Tatikonda, Nicha Dvornek, Xenophon Papademetris, and James S. Duncan. *AdaBelief Optimizer: Adapting Stepsizes by the Belief in Observed Gradients*. 2020. arXiv: 2010.07468 [cs.LG]. URL: https://arxiv.org/abs/2010.07468 (cit. on p. 27).

[17] Ilya Loshchilov and Frank Hutter. *Decoupled Weight Decay Regularization*. 2019. arXiv: 1711.05101 [cs.LG]. URL: https://arxiv.org/abs/1711.05101 (cit. on p. 28).

[18] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. *On the Variance of the Adaptive Learning Rate and Beyond*. 2021. arXiv: 1908.03265 [cs.LG]. URL: https://arxiv.org/abs/1908.03265 (cit. on p. 29).

[19] Xiangning Chen et al. *Symbolic Discovery of Optimization Algorithms*. 2023. arXiv: 2302.06675 [cs.LG]. URL: https://arxiv.org/abs/2302.06675 (cit. on p. 31).

[20] Yang You et al. *Large Batch Optimization for Deep Learning: Training BERT in 76 minutes*. 2020. arXiv: 1904.00962 [cs.LG]. URL: https://arxiv.org/abs/1904.00962 (cit. on pp. 32, 33).

[21] Zhenxun Zhuang. *Adaptive Strategies in Non-convex Optimization*. 2023. arXiv: 2306.10278 [cs.LG]. URL: https://arxiv.org/abs/2306.10278 (cit. on p. 33).

[22] Pierre Foret, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. *Sharpness-Aware Minimization for Efficiently Improving Generalization*. 2021. arXiv: 2010.01412 [cs.LG]. URL: https://arxiv.org/abs/2010.01412 (cit. on p. 36).

[23] Juntang Zhuang, Boqing Gong, Liangzhe Yuan, Yin Cui, Hartwig Adam, Nicha Dvornek, Sekhar Tatikonda, James Duncan, and Ting Liu. *Surrogate Gap Minimization Improves Sharpness-Aware Training*. 2022. arXiv: 2203.08065 [cs.LG]. URL: https://arxiv.org/abs/2203.08065 (cit. on p. 38).

[24] Sebastian Ruder. *An Overview of Gradient Descent Optimization Algorithms*. 2017. arXiv: 1609.04747v2 [cs.LG]. URL: https://arxiv.org/pdf/1609.04747 (cit. on p. 40).

[25] Thomas Frerix. «From Numerical Optimization to Deep Learning and Back». In: (2022). URL: https://mediatum.ub.tum.de/doc/1638886/gggp2k05j6o3chta9oyfb295o.Dissertation_ThomasFrerix.pdf (cit. on p. 40).

[26] Robin M. Schmidt, Frank Schneider, and Philipp Hennig. *Descending through a Crowded Valley - Benchmarking Deep Learning Optimizers*. 2021. arXiv: 2007.01547 [cs.LG]. URL: https://arxiv.org/abs/2007.01547 (cit. on pp. 40, 77, 80).

[27] Thomas P. Minka. *A Comparison of Numerical Optimizers for Logistic Regression*. 2003. URL: https://tminka.github.io/papers/logreg/minka-logreg.pdf (cit. on p. 40).

[28] Dami Choi, Christopher J. Shallue, Zachary Nado, Jaehoon Lee, Chris J. Maddison, and George E. Dahl. *On Empirical Comparisons of Optimizers for Deep Learning*. 2020. arXiv: 1910.05446 [cs.LG]. URL: https://arxiv.org/abs/1910.05446 (cit. on pp. 40, 41, 77).

[29] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791 (cit. on p. 45).

[30] Alex Krizhevsky and Geoffrey Hinton. *Learning multiple layers of features from tiny images*. Tech. rep. 0. Toronto, Ontario: University of Toronto, 2009. URL: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf (cit. on p. 46).

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep Residual Learning for Image Recognition». In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90 (cit. on p. 48).

[32] O.M. Parkhi, Andrea Vedaldi, A. Zisserman, and C.V. Jawahar. «Cats and dogs». In: June 2012, pp. 3498–3505. ISBN: 978-1-4673-1226-4. DOI: 10.1109/CVPR.2012.6248092 (cit. on p. 50).

[33] Mark Everingham, S. Eslami, Luc Van Gool, Christopher Williams, John Winn, and Andrew Zisserman. «The Pascal Visual Object Classes Challenge: A Retrospective». In: *International Journal of Computer Vision* 111 (Jan. 2014). DOI: 10.1007/s11263-014-0733-5 (cit. on p. 52).

[34] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. *Rethinking Atrous Convolution for Semantic Image Segmentation*. 2017. arXiv: 1706.05587 [cs.CV]. URL: https://arxiv.org/abs/1706.05587 (cit. on p. 52).

[35] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV]. URL: https://arxiv.org/abs/1505.04597 (cit. on p. 53).

[36] Andrew Maas, Raymond Daly, Peter Pham, Dan Huang, Andrew Ng, and Christopher Potts. «Learning Word Vectors for Sentiment Analysis». In: Jan. 2011, pp. 142–150 (cit. on p. 55).

[37] Richard Socher, A. Perelygin, J.Y. Wu, J. Chuang, C.D. Manning, A.Y. Ng, and C. Potts. «Recursive deep models for semantic compositionality over a sentiment treebank». In: *EMNLP* 1631 (Jan. 2013), pp. 1631–1642 (cit. on p. 56).

[38] Sepp Hochreiter and Jürgen Schmidhuber. «Long Short-Term Memory». In: *Neural Computation* 9 (Nov. 1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735 (cit. on pp. 56, 63).

[39] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. 2020. arXiv: 1910.01108 [cs.CL]. URL: https://arxiv.org/abs/1910.01108 (cit. on p. 57).

[40] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805 (cit. on p. 57).

[41] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. *Sequence to Sequence Learning with Neural Networks*. 2014. arXiv: 1409.3215 [cs.CL]. URL: https://arxiv.org/abs/1409.3215 (cit. on pp. 60, 72).

[42] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL]. URL: https://arxiv.org/abs/1409.0473 (cit. on p. 60).

[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762 (cit. on p. 60).

[44] Kishore Papineni, Salim Roukos, Todd Ward, and Wei Jing Zhu. «BLEU: a Method for Automatic Evaluation of Machine Translation». In: (Oct. 2002). DOI: 10.3115/1073083.1073135 (cit. on p. 61).

[45] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. *SQuAD: 100,000+ Questions for Machine Comprehension of Text*. 2016. arXiv: 1606.05250 [cs.CL]. URL: https://arxiv.org/abs/1606.05250 (cit. on pp. 62, 64).

[46] Wenhan Xiong, Jiawei Wu, Hong Wang, Vivek Kulkarni, Mo Yu, Shiyu Chang, Xiaoxiao Guo, and William Yang Wang. *TWEETQA: A Social Media Focused Question Answering Dataset*. 2019. arXiv: 1907.06292 [cs.CL]. URL: https://arxiv.org/abs/1907.06292 (cit. on p. 63).

[47] Erik F. Tjong Kim Sang and Fien De Meulder. *Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition*. 2003. arXiv: cs/0306050 [cs.CL]. URL: https://arxiv.org/abs/cs/0306050 (cit. on p. 65).

[48] Zhiheng Huang, Wei Xu, and Kai Yu. *Bidirectional LSTM-CRF Models for Sequence Tagging*. 2015. arXiv: 1508.01991 [cs.CL]. URL: https://arxiv.org/abs/1508.01991 (cit. on p. 66).

[49] Xuezhe Ma and Eduard Hovy. *End-to-end Sequence Labeling via Bi-directional LSTM-CNNs-CRF*. 2016. arXiv: 1603.01354 [cs.LG]. URL: https://arxiv.org/abs/1603.01354 (cit. on p. 67).

[50] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. *Pointer Sentinel Mixture Models*. 2016. arXiv: 1609.07843 [cs.CL]. URL: https://arxiv.org/abs/1609.07843 (cit. on p. 70).

[51] Mitchell Marcus, Mary Marcinkiewicz, and Beatrice Santorini. «Building a Large Annotated Corpus of English: The Penn Treebank». In: *Computational Linguistics* 19 (July 2002), pp. 313–330 (cit. on p. 70).

[52] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. «Improving language understanding by generative pre-training». In: (2018) (cit. on p. 70).

[53] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. 2014. arXiv: 1406.1078 [cs.CL]. URL: https://arxiv.org/abs/1406.1078 (cit. on p. 71).

[54] Karl Moritz Hermann, Tomáš Kočiský, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. *Teaching Machines to Read and Comprehend*. 2015. arXiv: 1506.03340 [cs.CL]. URL: https://arxiv.org/abs/1506.03340 (cit. on p. 73).

[55] Chin-Yew Lin. «ROUGE: A Package for Automatic Evaluation of Summaries». In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81. URL: https://aclanthology.org/W04-1013/ (cit. on p. 75).