ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

School of Information Sciences and Technology
Department of Informatics
Athens, Greece

Master of Science Thesis
in
Computer Science

Performance of Adaptive Stochastic Gradient Descent Optimization Algorithms in Natural Language Processing Tasks

Pavlos Poulos

Supervisor: Stavros Toumpis
Department of Informatics
Athens University of Economics and Business

Co-Supervisors: Ion Androutsopoulos
Department of Informatics
Athens University of Economics and Business

Makis Malakasiotis
Department of Informatics
Athens University of Economics and Business

September 2024

Pavlos Poulos

Performance of Adaptive Stochastic Gradient Descent Optimization Algorithms in Natural Language Processing Tasks

September 2024

Supervisor: Stavros Toumpis

Athens University of Economics and Business

School of Information Sciences and Technology

Department of Informatics

Athens, Greece

Abstract

This thesis explores the impact of hyperparameter tuning across various optimization algorithms in deep learning, focusing specifically on Natural Language Processing (NLP) tasks. Building on the work of Gkouti et al. [Gko+24], we investigate the effectiveness of different optimizers—both adaptive (e.g., Adam, AdamW) and non-adaptive (SGD, SGDM)—on an encoder-decoder model architecture applied to text summarization and machine translation tasks. Each task utilizes distinct datasets, providing a diverse evaluation setting.

Our goal is to assess whether tuning only the learning rate can yield near-optimal performance, as suggested by Gkouti et al., or if broader hyperparameter tuning is necessary for certain optimizers. Through extensive experimentation, we find that adaptive optimizers consistently perform well with minimal hyperparameter tuning, when the learning rate is the primary focus. This finding has significant implications for training efficiency, potentially reducing the computational cost of tuning multiple hyperparameters. However, non-adaptive optimizers, particularly SGDM, require more comprehensive hyperparameter tuning to achieve performance comparable to that of adaptive optimizers.

Our research extends the conclusions drawn from classification tasks in the study by Gkouti et al. to generative tasks, specifically text summarization and machine translation, across multiple datasets. More specifically, we confirm that the findings of Gkouti et al. hold for adaptive optimizers in these tasks. However, we challenge their conclusion that tuning additional hyperparameters in non-adaptive optimizers does not improve performance. In particular, we find that for SGDM, adjusting momentum alongside the learning rate provides a performance boost.

This thesis thus offers practical insights into optimization strategies for NLP, highlighting efficient tuning practices and examining the trade-offs between adaptive and non-adaptive optimization methods.

Περίληψη

Αυτή η διατριβή εξετάζει την επίδραση της ρύθμισης των υπερπαραμέτρων σε διάφορους αλγορίθμους βελτιστοποίησης στη βαθιά μάθηση, εστιάζοντας συγκεκριμένα σε προβλήματα Επεξεργασίας Φυσικής Γλώσσας (Natural Language Processing). Βασιζόμενοι στην εργασία των Gkouti et al. [Gko+24], διερευνούμε την αποτελεσματικότητα διαφόρων αλγορίθμων βελτιστοποίησης τόσο προσαρμοστικών (π.χ. Adam, AdamW) όσο και μη προσαρμοστικών (SGD, SGDM)—σε μια αρχιτεκτονική μοντέλου κωδικοποιητή-αποκωδικοποιητή που εφαρμόζεται σε προβλήματα περίληψης κειμένου και μηχανικής μετάφρασης.

Στόχος μας είναι να αξιολογήσουμε εάν η ρύθμιση μόνο του ρυθμού μάθησης μπορεί να αποδώσει σχεδόν βέλτιστη απόδοση, όπως προτείνουν οι Gkouti et al., ή αν απαιτείται εκτενέστερη ρύθμιση υπερπαραμέτρων για ορισμένους αλγορίθμους βελτιστοποίησης. Μέσα από εκτεταμένα πειράματα, διαπιστώνουμε ότι οι προσαρμοστικοί αλγόριθμοι βελτιστοποίησης αποδίδουν σταθερά καλά ακόμα και με μοναδική ρύθμιση αυτή του ρυθμού μάθησης. Αυτό το εύρημα έχει σημαντικές επιπτώσεις στην αποδοτικότητα της εκπαίδευσης, καθώς μπορεί να μειώσει το υπολογιστικό κόστος της ρύθμισης πολλαπλών υπερπαραμέτρων. Ωστόσο, οι μη προσαρμοστικοί αλγόριθμοι βελτιστοποίησης, ιδιαίτερα ο SGDM, απαιτούν πιο εκτεταμένη ρύθμιση υπερπαραμέτρων για να επιτύχουν απόδοση συγκρίσιμη με αυτή των προσαρμοστικών αλγορίθμων βελτιστοποίησης.

Η έρευνά μας επεκτείνει τα συμπεράσματα που εξήχθησαν από τα προβλήματα ταξινόμησης της μελέτης των Gkouti et al. σε προβλήματα γέννησης κειμένου, συγκεκριμένα στη σύνοψη κειμένου και στη μηχανική μετάφραση. Πιο συγκεκριμένα, επιβεβαιώνουμε ότι τα ευρήματα των Gkouti et al. ισχύουν για τους προσαρμοστικούς αλγορίθμους σε αυτά τα προβλήματα. Ωστόσο, αμφισβητούμε το συμπέρασμά τους ότι η ρύθμιση επιπλέον υπερπαραμέτρων στους μη προσαρμοστικούς αλγορίθμους βελτιστοποίησης δεν βελτιώνει την απόδοση. Συγκεκριμένα, διαπιστώνουμε ότι για τον SGDM, η προσαρμογή της ορμής (momentum) παράλληλα με τον ρυθμό μάθησης προσφέρει βελτίωση της απόδοσης.

Αυτή η διατριβή παρέχει έτσι πρακτικές γνώσεις σχετικά με τις στρατηγικές βελτιστοποίησης για την Επεξεργασία Φυσικής Γλώσσας, αναδεικνύοντας αποδοτικές πρακτικές ρύθμισης και εξετάζοντας τους συμβιβασμούς μεταξύ προσαρμοστικών και μη προσαρμοστικών αλγορίθμων βελτιστοποίησης.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Stavros Toumpis, without whom this thesis would not have been possible. His unwavering support, enthusiasm, and guidance throughout my MSc in Computer Science have left an indelible mark on my academic journey. Dr. Toumpis's passion for the field not only inspired my work on this thesis but also sparked my commitment to joining the scientific community and pursuing an academic career. His patience and dedication throughout this process were invaluable.

I am also deeply grateful to the co-supervisor, Professor Ion Androutsopoulos, for his insightful guidance and thought-provoking questions, which helped shape the baseline hypothesis of this thesis. His lectures during my time in the MSc program were instrumental in my intellectual growth, and his support motivated me to explore new ideas and push the boundaries of my research.

Finally, I would like to extend my heartfelt thanks to Dr. Makis Malakasiotis, an expert in Natural Language Processing, whose technical expertise and extensive experience in research were crucial to the success of my experiments. His guidance not only ensured the robustness of the code I developed, but also provided me with the confidence to complete my work. Without his help, I would still be working to set up my experiments.

The experiments on this thesis were supported by the TPU Research Cloud (TRC) program of Google.[1]

[1] https://sites.research.google/trc/about/

Contents

1 Introduction

Machine Learning is a subfield of Artificial Intelligence that focuses on the development of algorithms that can improve automatically through experience. In 1959, Arthur Samuel said, "Programming computers to learn from experience should eventually eliminate the need for much of this detailed programming effort" [Sam59], meaning that it is the field of study that gives computers the ability to learn without being explicitly programmed (see [AK15]).

In the early 1940s[1], Warren McCulloch and Walter Pitts invented the Perceptron, introducing a pioneering mathematical model aimed at solving binary classification problems. The Perceptron was inspired by the human brain, which automatically adjusts biological neurons to transmit signals in order to achieve specific outputs. This led to the creation of Artificial Neural Networks, which, like the Perceptron, can adjust the weights of mathematical model neurons to perform tasks such as classification, regression, prediction and many more, but on a larger scale. By the 2010s, with advances in computational power, data availability, and algorithms, Deep Neural Networks (DNN) began to rise significantly in popularity. These models dramatically increased the number of neurons to millions or even billions, allowing them to surpass traditional machine learning techniques in fields like computer vision, natural language processing, and more [LBH15].

Natural Language Processing (NLP), a field that studies the creation of models (Artificial Neural Networks) capable of understanding human text, has seen significant improvements in recent years. With the advent of Large Language Models (LLMs), such as GPT [Rad+19] and BERT [Dev+18], we have now reached a point where we can rely on these models for diverse language-based tasks in our daily lives. LLMs are distinguished by their extensive parameter counts—often billions—and their remarkable ability to process and generate human-like text across various applications. Their scale and complexity allow them to capture nuances of language, enabling sophisticated reasoning and creative outputs, which marks a significant advancement in NLP.

Understanding the basics of machine learning and deep neural networks provides a solid foundation for grasping the significance of optimizers in training models, particularly for LLMs. These models behave like super-powered language processors, capable of analyzing and generating vast amounts of text. But teaching a machine to comprehend human language is a difficult task—as it requires careful adjustments to millions, even billions, of parameters. That is where optimizers step in. Picture them as the guiding hands that help these models navigate the complexities of the data landscape. Optimizers like stochastic gradient descent (SGD) [Cau47], Adam [KB14], and RMSprop [TH12] act as the driving force behind the learning process, fine-tuning the model's parameters to improve its performance over time. They play a crucial role in guiding the model towards

[1] https://en.wikipedia.org/wiki/Perceptron

convergence, ensuring that it learns efficiently and effectively from the extensive data (textual or not) that it is exposed to.

Having a closer look into the world of LLMs, understanding the nuances of different optimizers becomes increasingly important. Each optimizer has its own unique characteristics and mechanisms, and knowing how to leverage them can significantly impact the model's ability to tackle various tasks. Whether we are dealing with language translation, text summarization, or sentiment analysis, a deeper understanding of optimizers empowers us to harness the full potential of LLMs and push the boundaries of what they can achieve.

1.1 Motivation and Problem Statement

In recent years, deep learning has seen a substantial rise of optimization algorithms, with Adam (Adaptive Moment Optimization algorithm), introduced by Kingma and Ba [KB14] in 2014, becoming one of the most widely used methods. The growing body of literature now includes hundreds of optimizers, each claiming specific advantages for certain tasks or model architectures. This variety has made selecting the right optimizer a critical decision in deep learning projects.

Gkouti et al. [Gko+24] provided important insights into optimizer performance for encoder-only models in classification tasks. Their findings indicated that tuning only the learning rate can often achieve results comparable to tuning all hyperparameters, and that Stochastic Gradient Descent with Momentum (SGDM) generally outperforms standard SGD. However, their experiments were limited to BERT-based [Dev+18] models in NLP classification tasks, leaving open questions about the applicability of their conclusions to other architectures and tasks.

This thesis builds on the work of Gkouti et al. by exploring additional dimensions:

1. **Model Architectures**: We extend the analysis by using an encoder-decoder model. this architecture is key to sequence-to-sequence tasks and may exhibit different optimization behavior.
2. **Task Diversity**: Beyond classification, we examine generative tasks such as summarization and machine translation to assess whether the optimization patterns observed in classification tasks hold for these contexts too.
3. **Dataset Variability**: To ensure robust conclusions, we include multiple datasets per task, reducing biases from domain-specific characteristics.
4. **Optimizer Comparison**: We reevaluate the performance of adaptive optimizers (e.g., Adam and its variants) versus traditional methods (e.g., SGD and SGDM) under these expanded conditions.

The main goal of this thesis is to evaluate whether the findings of Gkouti et al.—specifically, the effectiveness of learning rate tuning and the advantage of SGDM over SGD—apply to a broader range of deep learning applications. Key observations from their work include:

1. Tuning only the learning rate often yields results similar to tuning all hyperparameters for adaptive optimizers.
2. Among the adaptive optimizers there is not a substantial difference, when tuning only the learning rate or all their hyperparameters, and picking any of them will often yield similar results.
3. Hyperparameter tuning for the adaptive optimizers, even limited to only the learning rate, provides notable improvements compared to using default values.
4. When hyperparameter tuning is not feasible, SGDM with default settings is the most reliable choice.

Due to budget constraints, this thesis focuses on comparing two scenarios: tuning only the learning rate versus tuning all hyperparameters, leaving out the use of default settings. Specifically, we aim to address the following questions:

1. Does tuning only the learning rate often yields results similar to tuning all the hyperparameters for adaptive optimizers, when it comes to generative tasks (machine translation and summarization) or not?
2. Are there any substantial differences among the adaptive optimizers, and picking any of them will be a good practice when it comes in generative task as well ?
3. Do adaptive optimizers like Adam and its variants offer significant performance advantages over simpler methods such as SGDM in generative tasks like summarization and machine translation?
4. Can SGDM deliver comparable or better performance than adaptive optimizers, does it consistently outperform standard SGD or in specific scenarios only?

By answering these questions, we aim to provide the NLP community with clearer guidance on optimizer selection and hyperparameter tuning strategies. Our findings could streamline the model development process by identifying broadly applicable optimization principles. Additionally, this thesis contributes to the ongoing debate about the trade-offs between adaptive and non-adaptive optimization methods, particularly in the context of large language models and generative NLP tasks.

1.2 Thesis Structure

Chapter 2 introduces the fundamental concepts underlying this thesis. The chapter mainly focuses on optimization algorithms, discussing both adaptive and non-adaptive ones commonly used for training deep learning models. It also covers critical mathematical concepts such as loss minimization, momentum, and Nesterov's Accelerated Gradient (NAG), and how these algorithms and techniques enhance model training performance. This foundational overview sets the stage for the experimental analysis and results presented in later chapters.

Chapter 3 explores text summarization, a key task in natural language processing (NLP) where large texts are condensed while preserving key ideas. It distinguishes between

extractive and abstractive summarization, with a focus on abstractive methods, which require understanding and paraphrasing content while maintaining factual accuracy. The chapter introduces key evaluation metrics, including intrinsic metrics like ROUGE, BLEU, METEOR, and BERTScore, which assess summary quality, and extrinsic metrics that measure real-world utility. Special attention is given to BERTScore for its ability to evaluate summaries at a contextual level. This chapter sets the foundation for understanding summarization models and their evaluation in NLP research.

Chapter 4 examines the task of machine translation, a core area of natural language processing that involves automatically converting text from one language to another. At first we mention some old machine translation approaches, including Rule-Based Machine Translation (RBMT) and Statistical Machine Translation (SMT). We then introduce some of the most commonly used evaluation metrics in machine translation, such as the Bilingual Evaluation Understudy (BLEU), Metric for Evaluation of Translation with Explicit Ordering (METEOR), and BERTScore, highlighting their respective strengths and limitations. Finally, we emphasize the critical role of Large Language Models in machine translation, particularly their ability to excel in understanding and preserving context across long text sequences.

Chapter 5 outlines the experimental setup for the models, which share the same architecture but differ in hyperparameter values based on the chosen optimization algorithms. It details the datasets, training procedures, evaluation protocols, and preprocessing steps. The chapter also presents plots of the learning curves during training and the test results of the best-performing models in each specific training scenario. Additionally, it includes a comparison of the performance of different optimizers, providing insights into their effectiveness.

In the conclusion chapter (Chapter 6), we summarize the key findings of this thesis, which investigated hyperparameter tuning for various optimizers in NLP tasks. The experimental results for the summarization and machine translation tasks align with some findings of Gkouti et al. but also diverge in significant ways. Firstly, we confirmed that adaptive optimizers exhibit similar performance regardless of whether only the learning rate is tuned or all hyperparameters are tuned. Additionally, we confirmed that there are no substantial differences among adaptive optimizers when their hyperparameters are tuned, suggesting that selecting any of them (e.g., Adam) is a reasonable choice. Secondly, we found that SGDM outperforms plain SGD, but only when its momentum parameter is also tuned. This contrasts with the findings of Gkouti et al., who suggested that SGDM is superior to SGD in all scenarios. Thirdly, contrary to Gkouti et al., we observed that SGDM does not perform comparably to adaptive optimizers. Even when SGDM was trained with all hyperparameters tuned, it failed to match the performance of any adaptive optimizer. Gkouti et al. proposed that SGDM could be a viable choice when hyperparameter tuning is not possible; however, our results do not support this recommendation, as SGDM consistently lagged behind adaptive optimizers across all tested configurations, even when most of the hyperparameters of SGDM were tuned. These findings challenge previous

assumptions about the generalizability and effectiveness of SGDM, particularly in more complex NLP tasks such as summarization and machine translation.

Background and Related Work

2

To understand the optimization techniques used in artificial neural networks, it is essential to have a foundational grasp of their layered structure and how they transform inputs into outputs. Key concepts include loss functions, gradient-based weight adjustments, and basic calculus principles, like the chain rule, which underpin backpropagation. Additionally, familiarity with linear algebra, matrix operations, and optimization elements such as learning rates and convergence criteria is critical for interpreting various optimizers, including SGD and Adam, along with techniques like momentum.

In the following section, we introduce only the fundamental aspects of deep neural networks (DNNs), limiting the discussion to the basics. The experiments conducted, however, employ far more advanced techniques than those covered here, such as recurrent neural networks (RNNs) [HS97], convolutional neural networks (CNNs) [LeC+98], Transformers, attention mechanisms [Vas+17], residual connections [He+16] and many more. For readers interested in a more comprehensive foundation, we recommend consulting additional references such as [Pri23].

2.1 Gentle Introduction of a DNN

As we already mentioned, DNNs are sophisticated machine learning models inspired by the structure and function of the human brain. At their core, DNNs are composed of interconnected units called neurons, organized in layers, that process and transmit information.

The fundamental building block of a DNN is the artificial neuron, conceptually similar to a biological neuron. In a DNN, each neuron receives input signals, x_i, processes them, and produces an output y. This processing involves a weighted sum of inputs, followed by the application of an activation function. Mathematically, we can express the output y of a neuron as $y = f(\sum(w_i x_i) + b)$, where x_i are the inputs, w_i are the corresponding weights, b is a bias term, f is the activation function and the range of index i is the number of input neurons.

DNNs are typically organized in layers: an input layer, one or more hidden layers, and an output layer (see Figure 2.1). The depth of a network refers to the number of hidden layers, hence the term "deep" in Deep Neural Networks. Each layer processes the information received from the previous layer and passes it to the next, allowing the network to learn increasingly complex representations of the input data.

As shown in Figure 2.1, the input layer consists of neurons $x_1, x_2, \ldots, x_n$, which receive the initial data. The hidden layers, represented by the weights $w_{ij}^{(l)}$ where l denotes the layer number, process this information. The superscript (0) in $w_{ij}^{(0)}$ indicates weights connecting the input layer to the first hidden layer, while $w_{ij}^{(1)}$ represents weights between

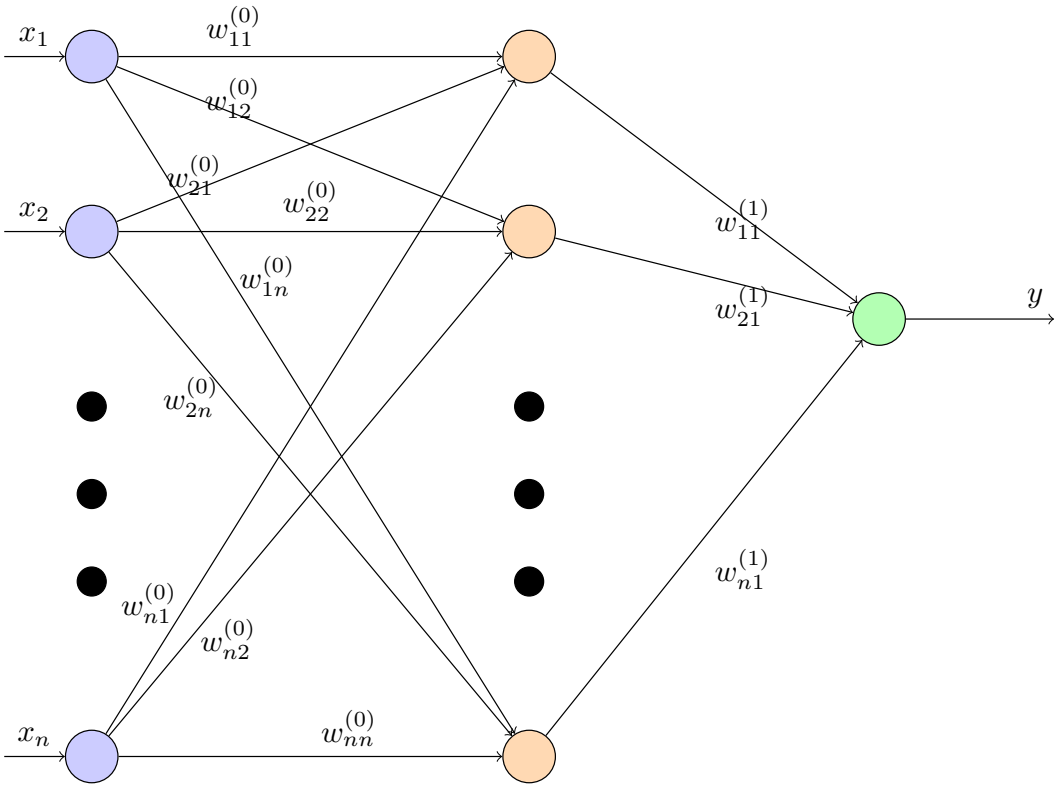

Fig. 2.1: Schematic representation of a Deep Neural Network (DNN) with input layer $(x_1, x_2, \ldots, x_n)$, hidden layers, weights $w_{ij}^{(l)}$, and output y.

the first and next hidden layer, in Figure 2.1 that will be the output layer. The final output of the network is denoted by y.

Various activation functions are used in DNNs to introduce non-linearity and enable the network to learn complex patterns. Common activation functions include:

1. ReLU (Rectified Linear Unit): $f(x) = \max(0, x)$
2. Sigmoid: $S(x) = \frac{1}{1+e^{-x}}$
3. Tanh (Hyperbolic Tangent): $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

Each of these functions has its characteristics and is suitable for different types of problems or network architectures.

Understanding the basic structure and function of DNNs, as illustrated in Figure 2.1, is crucial for grasping the concepts of loss minimization and optimization, which form the foundation of effective neural network training and performance. The figure provides a visual representation of how information flows from the input layer through the hidden layers to produce the final output, emphasizing the interconnected nature of neurons in a DNN.

2.2 Minimization of Objective Function (Loss) in Machine Learning

The power of DNNs lies in their ability to learn and adapt through training, where the network adjusts its weights and biases to minimize the difference between its predictions and correct outcomes. This process revolves around the use of an objective (or loss) function, which quantifies the error between predictions and true outcomes. The goal of training is to minimize this objective function, improving the model's predictive accuracy in tasks like classification and regression.

Objective functions vary depending on the specific task at hand. For instance, Mean Squared Error (MSE) is commonly used in regression tasks, while Cross Entropy loss is prevalent in classification tasks. The MSE for a dataset with n samples is defined as: $\mathcal{L}_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$, where y_i is the true value and $\hat{y}_i$ is the predicted value for the i-th sample. In contrast, the Cross Entropy loss for a classification task with m classes is given by $\mathcal{L}_{\text{CrossEntropy}} = -\sum_{i=1}^{n}\sum_{j=1}^{m} y_{i,j}\log\hat{y}_{i,j}$, where $y_{i,j}$ is a binary indicator (0 or 1) if class label j is the correct classification for sample i, and $\hat{y}_{i,j}$ is the predicted probability that sample i belongs to class j. For the special case that the $\hat{y}_{i,j} = 0$, we assume that $0\log 0 = 0$ [JJ00].

The process of minimizing the objective function entails iteratively adjusting model parameters (weights and biases in neural networks) using gradient descent algorithms like Stochastic Gradient Descent (SGD) [Cau47] and its variants such as Adam [KB14]. These algorithms compute gradients of the objective function with respect to model parameters and update them in directions that reduce the objective function value.

Mathematically, the minimization of the objective function involves finding optimal parameters θ that minimize $\mathcal{L}(\theta)$, where $\mathcal{L}$ denotes the objective function (loss) and θ the model's parameters. This optimization process forms the cornerstone of effective machine learning model training, ensuring generalization to unseen data and robust performance.

Understanding the nuances of minimizing the objective function is crucial for practitioners in machine learning and neural networks. This process forms the foundation for developing and evaluating training algorithms, directly impacting the reliability and performance of machine learning applications. However, we only touch upon the surface here. A deeper understanding requires familiarity with foundational concepts such as embeddings, transformer architectures, encoder-only models, and encoder-decoder models. We encourage readers to consult additional resources for further exploration of these topics [Pri23].

2.3 Non-Adaptive Methods

2.3.1 Gradient Descent Algorithm

The Gradient Descent (GD) algorithm, suggested by Cauchy in 1847 [Cau47], is one of the most fundamental algorithms in the field of optimization, particularly in the training

of neural networks. It operates by iteratively adjusting the parameters of the model to minimize the loss function $\mathcal{L}$, which quantifies the difference between the model's predictions and the actual outcomes. The ultimate goal is to find the parameters that minimize this loss function, ideally reaching a global minimum where the loss is the lowest.

Gradient Descent works by calculating g over the training set, defined as $g = \frac{1}{k}\sum_{i=1}^{k}\nabla\mathcal{L}(\theta_t)$, where k is the number of examples in the training set. Here, g represents the gradient of $\mathcal{L}(\theta)$, the loss function, with respect to all parameters together (viewed as a vector). The individual gradients $\nabla\mathcal{L}(\theta_t)$ are vectors of partial derivatives that point in the direction of the steepest ascent of the loss function. By updating the parameters in the opposite direction of the gradient, we move towards points with smaller loss. The update rule can be expressed as: $\theta_{t+1} = \theta_t - \epsilon g$, where θ_t represents the parameters at iteration t and ϵ is the learning rate, a hyper-parameter that determines the step size in each iteration.

The learning rate is crucial in the Gradient Descent process. If it is too small, the algorithm will converge slowly, requiring many iterations to reach a local minimum. Conversely, if it is too large, the algorithm may overshoot the minimum or even diverge, failing to find a solution. Therefore, selecting an appropriate learning rate is vital for efficient and effective training.

Below is the Algorithm 1 of Gradient Descent. In the next section we discuss its alternative, Stochastic Gradient Descent, and why it is needed.

Algorithm 1 Gradient Descent

Require: Learning rate ϵ, Initial parameter θ_0, Tolerance δ

1: Initialize time step $t = 0$
2: **while** $|\mathcal{L}(\theta_t)| \geq \delta$ **do**
3: Compute the $\mathcal{L}$ loss function for every example of the training set
4: Compute gradient: $g = \frac{1}{k}\sum_{i=1}^{k}\nabla\mathcal{L}(\theta_t)$ ▷ $\mathcal{L}$ loss function for i-th example
5: Apply update: $\theta_{t+1} = \theta_t - \epsilon g$
6: Increment time step: $t = t + 1$
7: **return** θ_t

2.3.2 Stochastic Gradient Descent Algorithm

Stochastic Gradient Descent (SGD) (Algorithm 2) is an extension of the Gradient Descent algorithm with one key difference. Unlike Gradient Descent, which computes the gradients using the entire dataset, SGD updates the model parameters using only a single example or a small batch of examples, termed the mini-batch at each iteration. This approach introduces randomness into the gradient computation.

SGD was first introduced by Robbins and Monro in 1951 [RM51] and later extended by Kiefer and Wolfowitz in 1952 [KW52].

The primary advantage of SGD lies in its computational efficiency, particularly with large datasets. Instead of calculating the gradient over the entire training set, SGD approximates it using a subset, known as a mini-batch, which significantly reduces the computational burden. This makes it suitable for scenarios where the dataset is too large to fit into memory or when the cost of computing the full gradient is prohibitive.

Although SGD, may not guarantee convergence to the global minimum of the loss function, it often finds a good approximation in a reasonable amount of time. This makes it particularly useful for large-scale machine learning problems.

SGD is essentially a stochastic approximation of the Gradient Descent method. By using an estimated gradient based on a randomly selected mini-batch, it achieves faster iterations but may have a lower convergence rate. Despite this trade-off, SGD's ability to efficiently handle large datasets makes it a popular choice in machine learning.

Algorithm 2 Stochastic Gradient Descent

Require: learning rate ϵ, Initial Parameter θ_0, Tolerance δ

Initialize time step $t = 0$
while $|\mathcal{L}(\theta_t)| \geq \delta$ **do**
    Sample a mini-batch of m examples from the training set
    Compute gradient: $g = \frac{1}{m} \sum_{i=1}^{m} \nabla \mathcal{L}(\theta_t)$ ▷ $\mathcal{L}$ loss function for i-th example
    Apply update: $\theta_{t+1} = \theta_t - \epsilon g$
    Increment time step: $t = t + 1$
return θ_t

2.3.3 Stochastic Gradient Descent Better with Momentum

Basic Momentum Method

In 1999, Ning Qian came up with a smart way to include momentum in the SGD algorithm (Algorithm 3) [Qia99]. This was achievable by using a weighted average of past steps to avoid large changes in the direction traveled when updating values. It adds a speed vector v, which controls how fast and in what direction the updates happen.

The speed update is calculated using a weighted average of the negative gradients. This idea comes from physics, where the negative gradient is like a force pushing an object that is subject to inertia through space, following Newton's laws. A hyperparameter $\alpha \in (0, 1)$ controls the importance of past gradients with larger α values indicating higher importance. This is different from normal gradient descent because it changes the object's speed based on these forces, instead of just taking steps based on each calculation.

Algorithm 3 SGD with Momentum

Require: Step size ϵ, momentum number α, Starting values θ_0, starting speed v, Tolerance δ

Initialize time step $t = 0$
while $|\mathcal{L}(\theta_t)| \geq \delta$ **do**
    Sample a mini-batch of m examples from the training set
    Compute gradient: $g = \frac{1}{m} \sum_{i=1}^{m} \nabla \mathcal{L}(\theta_t)$ ▷ $\mathcal{L}$ is loss for example i
    Update speed: $v = \alpha v - \epsilon g$
    Apply update: $\theta_{t+1} = \theta_t + v$
    Increment time step: $t = t + 1$
return θ_t

Nesterov's Improved Momentum

Inspiration from Nesterov's accelerated gradient method [Nes83] led to the development of an algorithm known as Nesterov momentum (Algorithm 4), in which the gradient is calculated at a different stage. Specifically, in Nesterov momentum, the gradient is computed after applying the current velocity update. This subtle change leads to faster convergence compared to standard momentum. Theoretical analysis shows that Nesterov momentum can be more effective for certain types of functions, and in practice, it often results in better performance, including improved generalization. This means that the trained model performs better on unseen data [JL22] [LTP22].

Algorithm 4 SGD with Nesterov Momentum

Require: Step size ϵ, Momentum number α, Starting values θ_0, Starting speed v, Tolerance δ

Initialize time step $t = 0$
while $|\mathcal{L}(\theta_t)| \geq \delta$ **do**
    Sample a mini-batch of m examples from the training set
    Find middle update: $\hat{\theta} = \theta_t + \alpha v$
    Compute gradient: $g = \frac{1}{m} \sum_{i=1}^{m} \nabla \mathcal{L}(\hat{\theta})$ ▷ $\mathcal{L}$ is loss for example i
    Update speed: $v = \alpha v - \epsilon g$
    Apply update: $\theta_{t+1} = \theta_t + v$
    Increment time step: $t = t + 1$
return θ_t

2.4 Adaptive Methods

In machine learning, selecting the appropriate hyperparameters is crucial for achieving good model performance. Among these hyperparameters, the learning rate is particularly important. If it is set too high, the model may never converge to a good solution. If it is set too low, the training process can become excessively slow. Adaptive methods help address

this issue by adjusting the learning rate during training. Additionally, these methods adjust the learning rate elementwise.

The concept of using second derivatives is fundamental in understanding adaptive methods. The second derivative of a function $f(x)$ provides insight into how the first derivative $f'(x)$ changes as we modify the input x. This information is crucial because it helps us determine whether a step based on the gradient will be effective. The second derivative essentially measures the curvature of the function. For functions with multiple inputs, these second derivatives form the Hessian matrix.

The Hessian matrix, H_f, is a $n \times n$-dimensional square matrix comprising the second-order partial derivatives of a function f with n variables $x = (x_1, x_2, ..., x_n)$. The formula for the (i, j)-th position of the H_f is given by $H_{ij} = \frac{\partial^2 f}{\partial x_i x_j}$ with the full matrix formula to be:

$$H_f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

It is symmetric and for this reason can be decomposed into a set of real eigenvalues λ_i and their corresponding eigenvectors d_i: $H = D\Lambda D^{-1}$, where D is an orthogonal matrix of the eigenvectors d_i and Λ is a diagonal matrix of the eigenvalues λ_i. This decomposition allows us to understand the curvature in different directions. The eigenvalues indicate the rate at which the function's derivative changes in the direction of their corresponding eigenvectors.

The maximum and minimum eigenvalues of the Hessian matrix are particularly significant. The eigenvector(s) maximum eigenvalue shows the direction(s) in which the function's curvature is steepest, while the eigenvector(s) minimum eigenvalue indicates the direction(s) with the least curvature. When the difference between these eigenvalues is large, it is said that the Hessian has a poor condition number, leading to difficulties in optimization.

Observe that, Gradient Descent, does not inherently account for these variations in curvature. As a result, it may not efficiently explore directions with different curvatures, necessitating a smaller step size to avoid overshooting minima. This often results in a slower convergence.

Loss functions encountered in Neural Networks, characterized by their high-dimensional and non-convex nature, present additional challenges. Each parameter dimension can have a distinct sensitivity, making the optimization landscape complex. Representing the entire Hessian matrix in such cases is impractical due to its size, especially when dealing with millions or billions of parameters.

Adaptive methods have been introduced to tackle these challenges by dynamically adjusting the learning rate for each dimension. This approach helps in efficiently navigating the optimization landscape, leading to better performance and faster convergence.

Choosing an optimal learning rate beforehand is challenging, which is why adaptive learning rate methods are both useful and popular. These methods simplify the optimization process by automatically tuning the learning rate per dimension, allowing for more effective and efficient training of complex models.

2.4.1 AdaGrad

AdaGrad (Algorithm 5), introduced by Duchi et al. [DHS11], stands for Adaptive Gradient Algorithm. It is designed to adapt the learning rates of all parameters by scaling them elementwise and inversely proportional to the square root of the sum of all their past squared gradients.

Algorithm 5 AdaGrad

Require: Global learning rate ϵ, Initial parameter θ_0, Small constant δ, Tolerance δ
  Initialize gradient accumulation variable $r = 0$
  Initialize time step $t = 0$
  while $|\mathcal{L}(\theta_t)| \geq \delta$ **do**
    Sample a minibatch of m examples from the training set
    Compute gradient: $g = \frac{1}{m} \sum_{i=1}^{m} \nabla \mathcal{L}(\theta_t)$
    Accumulate squared gradient: $r = r + g \odot g$ ▷ $\odot$ denotes element-wise multiplication
    Update rule: $\Delta\theta = -\frac{\epsilon}{\sqrt{r}+\delta} \odot g$ ▷ division and square root applied element-wise
    Apply update: $\theta_{t+1} = \theta_t + \Delta\theta$
    Increment time step: $t = t + 1$

AdaGrad adapts the learning rates based on the history of partial derivatives for each parameter. Parameters with high gradients, as well as those with high variability, will see a rapid decrease in their learning rate, while those with small gradients, and those with low variability, will have a slower decrease. However, one issue with AdaGrad is that it can decrease the learning rate too quickly, causing the algorithm to stop learning early. This issue has led to the development of extensions to AdaGrad.

2.4.2 RMSProp

RMSProp (Algorithm 6) was introduced by Hinton [TH12] to address the problem of AdaGrad's aggressive learning rate decay. RMSProp, short for Root Mean Square Propagation, modifies AdaGrad by applying an exponentially weighted moving average of the squared gradients.

This exponentially weighted moving average approach keeps track of a smoothed average of the squared gradients from previous iterations, giving more weight to recent gradients than older ones. Mathematically, this is achieved by updating a running average of the squared gradients $r = \rho r + (1 - \rho) g \odot g$ where ρ, typically within the range $[0.9, 0.999]$, is a decay factor that controls how quickly the influence of older gradients

diminishes. Here, $g \odot g$ represents the element-wise square of the gradients from the previous step. This approach enables the algorithm to adjust to recent changes in gradient magnitude while still leveraging historical information.

Algorithm 6 RMSProp

Require: Global learning rate ϵ, Initial parameter θ_0, Decay rate ρ, Small constant $\delta > 0$, Tolerance δ

Initialize gradient accumulation variable $r = 0$

Initialize time step $t = 0$

while $|\mathcal{L}(\theta_t)| \geq \delta$ **do**

Sample a minibatch of m examples from the training set

Compute gradient: $g = \frac{1}{m} \sum_{i=1}^{m} \nabla \mathcal{L}(\theta_t)$

Update moving average of squared gradient: $r = \rho r + (1 - \rho) g \odot g$

Update rule: $\Delta\theta = -\frac{\epsilon}{\sqrt{r}+\delta} \odot g$ ▷ division and square root applied element-wise

Apply update: $\theta_{t+1} = \theta_t + \Delta\theta$

Increment time step: $t = t + 1$

By using this moving average, RMSProp effectively controls the learning rate: gradients that have been consistently large in recent steps will lead to smaller adjustments, while directions with smaller recent gradients will allow for relatively larger updates. This stabilizes the learning rate throughout training, preventing it from diminishing too rapidly.

2.4.3 AdaDelta

As previously mentioned, AdaGrad can cause the learning rate to decrease too quickly, potentially stopping the training process prematurely. AdaDelta (Algorithm 7), introduced by Zeiler [Zei12], addresses this issue by eliminating the need for a manually set global learning rate ϵ (as used in previous algorithms). Instead, AdaDelta adapts the learning rate dynamically based on a moving window of gradient updates, allowing it to adjust more effectively throughout training.

Algorithm 7 AdaDelta

Require: Decay rates ρ and β, Initial parameter θ_0, Small constant δ, Tolerance δ

Initialize $D_{-1} = 0$

Initialize gradient accumulation variable $r = 0$

Initialize time step $t = 0$

while $|\mathcal{L}(\theta_t)| \geq \delta$ **do**

Sample a minibatch of m examples from the training set

Compute gradient: $g = \frac{1}{m} \sum_{i=1}^{m} \nabla \mathcal{L}(\theta_t)$

Accumulate squared gradient: $r = \rho r + (1 - \rho) g \odot g$

Compute update: $\Delta\theta = -\frac{\sqrt{D_{t-1}+\delta}}{\sqrt{r}+\delta} \odot g$ ▷ division and square root applied element-wise

Apply update: $\theta_{t+1} = \theta_t + \Delta\theta$

Accumulate squared updates: $D_t = \beta D_{t-1} + (1 - \beta)(\Delta\theta)^2$

Increment time step: $t = t + 1$

The key difference between AdaDelta and RMSProp is that AdaDelta eliminates the need for a learning rate parameter by using the exponential moving average of squared updates. This makes AdaDelta more appealing for scenarios where no hyperparameter search is possible or when only minimal tuning is available, as it doesn't require setting a learning rate. In contrast, other optimizers often need careful adjustment of the learning rate during hyperparameter search to perform effectively.

2.4.4 Adam

Adam (Algorithm 8), which stands for Adaptive Moment Estimation, was introduced by Kingma and Ba [KB14]. It combines the benefits of two other extensions of stochastic gradient descent: the momentum algorithm and RMSProp. Adam computes adaptive learning rates for each parameter.

Algorithm 8 Adam

Require: Global learning rate ϵ, Initial parameter θ_0, Exponential decay rates for moment estimates, $\rho_1, \rho_2 \in [0, 1]$, Small constant δ, Tolerance δ

Initialize first and second moment variables $s = 0, r = 0$

Initialize time step $t = 0$

while $|\mathcal{L}(\theta_t)| \geq \delta$ **do**

Sample a minibatch of m examples from the training set

Compute gradient: $g = \frac{1}{m} \sum_{i=1}^{m} \nabla \mathcal{L}(\theta_t)$

Update biased first moment estimate: $s = \rho_1 s + (1 - \rho_1) g$

Update biased second moment estimate: $r = \rho_2 r + (1 - \rho_2) g \odot g$

Correct bias in first moment: $\hat{s} = \frac{s}{1-\rho_1^t}$

Correct bias in second moment: $\hat{r} = \frac{r}{1-\rho_2^t}$

Compute update: $\Delta\theta = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}}+\delta}$ ▷ operations applied element-wise

Apply update: $\theta_{t+1} = \theta_t + \Delta\theta$

Increment time step: $t = t + 1$

Adam uses both the first moment (mean) and the second moment (uncentered variance) of the gradients to adapt the learning rate for each respective parameter. This helps in efficiently navigating the optimization landscape.

2.4.5 NAdam

NAdam, short for Nesterov-accelerated Adaptive Moment Estimation (Algorithm 9), integrates the strengths of both Adam optimization and Nesterov momentum. By combining Adam's adaptive learning rate with Nesterov's forward-looking gradient approach, NAdam enhances convergence speed and accuracy, making it particularly effective in scenarios requiring fast, stable optimization.

The key advantage of NAdam is its ability to adjust the learning rate for each parameter while also benefiting from Nesterov momentum-based acceleration. This can lead to faster convergence in many optimization scenarios.

Algorithm 9 NAdam

Require: Global learning rate ϵ, Initial parameter θ_0, Exponential decay rates for moment estimates, $\rho_1, \rho_2 \in [0, 1)$, Small constant $\delta > 0$, Tolerance δ

Initialize time step $t = 0$

Initialize 1st and 2nd moment variables $s = 0, r = 0$

while $|\mathcal{L}(\theta_t)| \geq \delta$ **do**

Sample a minibatch of m examples from the training set

Compute gradient: $g = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \mathcal{L}(\theta_t)$

Update biased 1st moment estimate: $s = \rho_1 s + (1 - \rho_1) g$

Update biased 2nd moment estimate: $r = \rho_2 r + (1 - \rho_2) g \odot g$

Correct bias in 1st moment with Nesterov momentum: $\hat{s} = \frac{\rho_1}{1-\rho_1^t} s + \frac{(1-\rho_1)}{1-\rho_1^t} g$

Correct bias in 2nd moment: $\hat{r} = \frac{r}{1-\rho_2^t}$

Update rule: $\Delta\theta = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}}+\delta}$ (operations applied element-wise)

Apply update: $\theta_{t+1} = \theta_t + \Delta\theta$

Increment time step: $t = t + 1$

2.4.6 AdaMax

AdaMax (Algorithm 10), is a variant of Adam that uses the L_∞ norm instead of the L_2 norm for the second moment estimate [KB14]. This change aims to improve stability, especially when dealing with sparse gradients or during long training periods.

The main difference in AdaMax is how it updates and uses the second moment estimate. Instead of using the root mean square of past gradients, it uses the maximum. This can help prevent the learning rates from becoming too small over time.

Algorithm 10 AdaMax

Require: Global learning rate ϵ, Initial parameter θ_0, Exponential decay rates for moment estimates, $\rho_1, \rho_2 \in [0, 1)$, Small constant $\delta > 0$, Tolerance δ

Initialize time step $t = 0$

Initialize 1^{st} and 2^{nd} moment variables $s = 0, r = 0$

while $|\mathcal{L}(\theta_t)| \geq \delta$ **do**

Sample a minibatch of m examples from the training set

Compute gradient: $g = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \mathcal{L}(\theta_t)$

Update 1^{st} moment estimate: $s = \rho_1 s + (1 - \rho_1) g$

Update infinite-order moment estimate: $r = \max(\rho_2 r, |g|)$

Update rule: $\Delta\theta = -\epsilon \frac{s}{\sqrt{r}+\delta}$ (operations applied element-wise)

Apply update: $\theta_{t+1} = \theta_t + \Delta\theta$

Increment time step: $t = t + 1$

2.4.7 AdaBound

AdaBound (Algorithm 11), proposed by Luo et al. [Luo+19], is a variant of Adam that introduces dynamic bounds on learning rates. This optimizer is designed to enhance robustness when dealing with extreme learning rates. AdaBound applies dynamic bounds to learning rates, where the lower bound $\eta_l(t)$ and upper bound $\eta_u(t)$ are functions of the time step t. Initially, the lower bound is set to zero and the upper bound to infinity. Both bounds smoothly converge to a constant final step size ϵ. Specifically, $\eta_l(t)$ is nondecreasing, starting from 0 at $t = 0$ and converging to ϵ asymptotically, while $\eta_u(t)$ is nonincreasing, starting from ∞ at $t = 0$ and also converging to ϵ asymptotically.

At the beginning of training, AdaBound behaves similarly to Adam, as the bounds have minimal impact on the learning rates. This allows for fast initial progress, leveraging Adam's adaptive learning rate benefits. As training progresses, the bounds become more restrictive, and the algorithm gradually transforms into an SGD-like optimizer with momentum.

The algorithm for AdaBound is as follows:

Algorithm 11 AdaBound

Require: Global learning rate ϵ, Initial Parameter θ_0, Exponential decay rates for moment estimates, $\rho_1, \rho_2 \in [0, 1)$, Tolerance δ

Initialize step size at first iteration $\eta_0 = \epsilon$

Initialize time step $t = 0$

Initialize 1st and 2nd moment variables $s = 0, r = 0, \hat{r} = 0$

Lower bound function η_l, Upper bound function η_u

while $|\mathcal{L}(\theta_t)| \geq \delta$ **do**

Sample a minibatch of m examples from the training set

Compute gradient: $g = \frac{1}{m} \sum_{i=1}^{m} \nabla \mathcal{L}(\theta_t)$

Update first moment estimate: $s = \rho_1 s + (1 - \rho_1) g$

Update second moment estimate: $r = \rho_2 r + (1 - \rho_2) g \odot g$

$\eta_t = Clip(\frac{\epsilon}{\sqrt{r}}, \eta_l(t), \eta_u(t))$ (clips the learning rate element-wisely)

Update rule: $\Delta\theta = -\eta_t \odot s$

Apply update: $\theta_{t+1} = \theta_t + \Delta\theta$

Increment time step: $t = t + 1$

2.4.8 AdamW

AdamW (Algorithm 12), presented by Loshchilov and Hutter [LH17], is a variant of Adam that decouple the weight decay from the gradient-based update. This modification addresses a key issue with L2 regularization in Adam, i.e., the fact that the regularization term is added to the loss function and is thus affected by the adaptive learning rates. AdamW, in contrast, applies the weight decay directly to the weights, separate from the gradient update.

The update rule in AdamW is given by: $\theta_{t+1} = \theta_t - \left(\alpha \cdot \frac{1}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t + \lambda \theta_t\right)$, where λ is the weight decay rate, α is the learning rate, $\hat{m}_t$ is the bias-corrected first moment estimate, and $\hat{v}_t$ is the bias-corrected second moment estimate. This approach to regularization is often referred to as "weight decay" in the context of deep learning libraries. It's worth noting that in Adam with L2 regularization, the weight decay is normalized by the second moment estimate, while in AdamW, it is applied directly to the weights.

Algorithm 12 AdamW

Require: Global learning rate α, Weight decay rate λ, Initial parameter θ_0, Exponential decay rates for moment estimates, $\beta_1, \beta_2 \in [0, 1)$, Tolerance δ

Initialize time step $t = 0$

Initialize 1st and 2nd moment variables $m = 0, v = 0$

while $|\mathcal{L}(\theta_t)| \geq \delta$ **do**

Sample a minibatch of m examples from the training set

Compute gradient: $g_t = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \mathcal{L}(\theta_t)$

Update biased 1st moment estimate: $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$

Update biased 2nd moment estimate: $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$

Correct bias in 1st moment: $\hat{m}_t = m_t / (1 - \beta_1^t)$

Correct bias in 2nd moment: $\hat{v}_t = v_t / (1 - \beta_2^t)$

Apply update: $\theta_t = \theta_{t-1} - \left(\alpha \cdot \frac{1}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t + \lambda \theta_{t-1} \right)$

Increment time step: $t = t + 1$

2.4.9 AMSGrad

AMSGrad (Algorithm 13), proposed by Reddi et al. [RKK19], addresses a potential issue with Adam where the adaptive learning rate can sometimes increase, leading to poor convergence. The key difference in AMSGrad is that it maintains the maximum of past squared gradients, rather than the exponentially decaying average used in Adam. In AMSGrad, the maximum value of the second moment estimate is used for normalizing the gradient update. This ensures that the effective step size never increases, which helps to avoid the convergence issues that can sometimes occur with Adam.

AMSGrad maintains a running maximum of past squared gradients and uses this maximum for normalization. This approach results in a nonincreasing step size over time. Consequently, AMSGrad is claimed to have a smaller learning rate in each step compared with Adam.

Both AdamW and AMSGrad represent attempts to address specific limitations of the original Adam optimizer. While they can offer improvements in certain scenarios, their effectiveness may vary depending on the specific problem and model architecture. As with any optimization algorithm, it is often beneficial to experiment with different options to find the best fit for a particular task.

Algorithm 13 AMSGrad

Require: Global learning rate ϵ
Require: Initial parameter θ_0
Require: Exponential decay rates for moment estimates, $\beta_1, \beta_2 \in [0, 1)$, Tolerance δ
Initialize time step $t = 0$
Require: Small constant δ
Initialize 1st, 2nd, and maximum 2nd moment variables $m = 0, v = 0, \hat{v} = 0$
while $|\mathcal{L}(\theta_t)| \geq \delta$ **do**
Sample a minibatch of m examples from the training set
Compute gradient: $g = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \mathcal{L}(\theta_t)$
Update 1st moment estimate: $m = \beta_1 m + (1 - \beta_1) g$
Update 2nd moment estimate: $v = \beta_2 v + (1 - \beta_2) g \odot g$
Maintain maximum value: $\hat{v} = \max(\hat{v}, v)$
Update rule: $\Delta\theta = -\epsilon \frac{m}{\sqrt{\hat{v}} + \delta}$ (operations applied element-wise)
Apply update: $\theta_{t+1} = \theta_t + \Delta\theta$
Increment time step: $t = t + 1$

3 The Summarization Task

Building upon the work of Nefeli et al. [Gko+24], this chapter focuses on a different problem in the NLP community, specifically the *summarization task*. Summarization involves condensing a large body of text into a shorter version while retaining the main ideas and key information. The evaluation of summarization models is crucial to ensure that the generated summaries are both informative and coherent. This chapter will cover the problem statement and the metrics used to evaluate summarization.

3.1 Problem Statement

The task of text summarization aims to create a brief and accurate shorter version of a longer text. There are two main types of summarization: extractive and abstractive. Extractive summarization involves selecting key sentences or phrases directly from the source text, while abstractive summarization generates new sentences that convey the same meaning as the source text. Extractive summarization is often simpler and more straightforward, as it involves ranking and selecting existing sentences, but it may result in summaries that are disjointed or lack coherence. Abstractive summarization, on the other hand, requires a deeper understanding of the text and the ability to paraphrase and generate coherent, grammatically correct sentences.

In this work, we focus on the task of abstractive summarization, which presents several challenges. These include not only the deep understanding of the original content and the creation of grammatically correct, coherent sentences but also ensuring that the generated summaries are factually accurate and free from hallucinations.

When we refer to the term "hallucinations" in the context of large language models (LLMs), we are primarily discussing instances where the model generates information that is either missing from the source text or factually incorrect. For example, an LLM might incorrectly state that the capital of France is Athens [Ban+23] [Hua+23].

3.2 Summarization Metrics

To assess the performance of summarization models, various evaluation metrics are employed. These metrics can be broadly classified into two categories: intrinsic and extrinsic. Intrinsic metrics measure the quality of the summary itself, while extrinsic metrics evaluate the usefulness of the summary in a specific task. Both types of metrics are essential for a comprehensive evaluation of summarization models, as they provide insights into different aspects of summary quality.

3.2.1 Intrinsic Metrics

Intrinsic metrics focus on the properties of the generated summary, such as its relevance, informativeness, coherence, and fluency. Below, we define four commonly used intrinsic metrics in summarization: ROUGE, BLEU, METEOR, and BERTScore.

ROUGE (Recall-Oriented Understudy for Gisting Evaluation)

ROUGE is a set of metrics used to evaluate the overlap between n-grams in a generated summary and a reference summary [Lin04]. It includes several variants, such as ROUGE-N, ROUGE-L, and ROUGE-W. For more details, see [Gra15]. Mathematically, ROUGE-N is defined as:

$$\text{ROUGE-N} = \frac{\sum_{S \in \text{RefSummaries}} \sum_{gram_n \in S} \text{Count}_{match}(gram_n)}{\sum_{S \in \text{RefSummaries}} \sum_{gram_n \in S} \text{Count}(gram_n)}$$

where $\text{Count}_{match}(gram_n)$ denotes the number of overlapping n-grams between the generated and reference summaries.

In our experiments, we focus on two ROUGE-N metrics: ROUGE-1 and ROUGE-2. ROUGE-1 evaluates unigram (single word) overlap, capturing the overall lexical similarity between the generated and reference summaries. ROUGE-2 measures bigram (two-word sequence) overlap, which considers the preservation of some word pairs and captures a higher level of contextual accuracy compared to ROUGE-1.

BLEU (Bilingual Evaluation Understudy)

Although BLEU [Pap+02] is primarily used in machine translation, it can also be applied to summarization tasks. It measures the precision of n-grams between generated and reference summaries, emphasizing exact matches. BLEU incorporates a brevity penalty (BP) to handle length discrepancies between the generated and reference summaries, and it is computed as follows:

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases}$$

where r and c are the reference and candidate lengths, respectively,

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

where p_n represents the modified n-gram precision, calculated as:

$$p_n = \frac{\sum_{C \in \{\text{Candidates}\}} \sum_{\text{n-gram} \in C} \text{Count}_{\text{clip}}(\text{n-gram})}{\sum_{C' \in \{\text{Candidates}\}} \sum_{\text{n-gram}' \in C'} \text{Count}(\text{n-gram}')}$$

In this formula, $\text{Count}_{\text{clip}}(\text{n-gram})$ is the clipped count of each n-gram in the candidate that matches any reference, limiting the count to the maximum that appears in any single reference sentence, the weights w_n are assigned to each n-gram precision and the

denominator sums the counts of all candidate n-grams. While BLEU is a widely recognized metric, especially in machine translation, we opted not to use it in our experiments.

METEOR (Metric for Evaluation of Translation with Explicit Ordering)

METEOR improves upon BLEU by considering synonyms, stemming, and word order [BL05]. It calculates a harmonic mean of precision and recall, with a penalty for word order mismatches:

$$\text{METEOR} = F_{mean} \cdot (1 - \text{Penalty})$$

with the harmonic mean to be $F_{mean} = \frac{10P \cdot R}{(R+9P)}$, where R represents unigram recall and P represents unigram precision (see [BL05], [Van79]). The harmonic mean places more weight on lower values, making it more sensitive to both precision and recall compared to the arithmetic mean.

The Penalty term is determined by fragmentation, or the number of non-contiguous segments in the match. This penalty factor accounts for word order mismatches between the system translation and reference translation. The flexibility of considering semantics, synonyms, and word order through the harmonic mean and penalty make METEOR suitable for capturing more nuanced similarities beyond just n-gram overlap, an improvement over metrics like BLEU. Like the BLUE score, METEOR did not take place in the evaluation of our experiments.

BERTScore

BERTScore [Zha+19] evaluates machine-generated text in a manner that closely mimics human assessment. While originally developed for machine translation tasks, its applicability extends to text summarization as well.

Introduced as an innovative evaluation metric, BERTScore harnesses the power of contextual embeddings derived from pre-trained language models such as BERT [Dev+18] to assess the quality of generated text by comparing it to reference examples. This method stands in contrast to conventional $n-$gram based metrics like ROUGE [Lin04] and BLEU [Pap+02], which rely exclusively on exact word matches. Instead, BERTScore calculates semantic similarity at a contextual level, offering a more refined and comprehensive evaluation for various text generation tasks, including summarization and translation.

At its core, BERTScore operates by transforming both the candidate text $\hat{x}$ and reference text x, using a pre-trained transformer model, into contextual embeddings to represent the tokens. The metric then computes the cosine similarity between each token in the candidate text and its most similar token in the reference text, effectively capturing the optimal semantic alignment, as shown in Figure 3.1. The maximum cosine similarity across the rows of the Maximum Similarity matrix is used to calculate the recall $R_{BERT} = \frac{1}{|x|} \sum_{x_i \in x} \max_{\hat{x}_j \in \hat{x}} x_i^\top \hat{x}_j$. Similarly, the precision $P_{BERT} = \frac{1}{|\hat{x}|} \sum_{\hat{x}_j \in \hat{x}} \max_{x_i \in x} x_i^\top \hat{x}_j$ is calculated by taking the maximum cosine similarity across the columns of the Maximum Similarity matrix (see Figure 3.1). These similarity scores are subsequently aggregated across all tokens, meaning that both sets of tokens (candidate and reference) are considered

in the final precision, recall, and F1 scores $F_{BERT} = 2 \cdot \frac{P_{BERT} \cdot R_{BERT}}{P_{BERT} + R_{BERT}}$, capturing the overall alignment between the two sequences. This methodology allows BERTScore to effectively account for synonyms and paraphrases, as words with similar meanings are positioned closely in the embedding space, regardless of their surface forms.

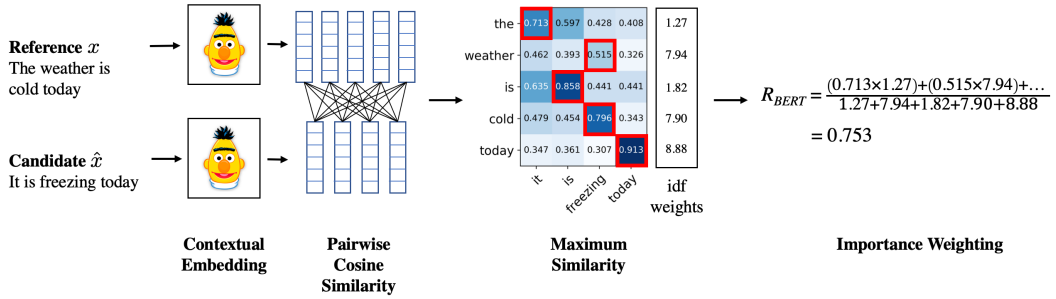

Fig. 3.1: Illustration of the computation of the recall metric R_{BERT} given the reference x and candidate $\hat{x}$, in addition with the computation of the BERT embeddings and pairwise cosine similarity. More on the official paper [Zha+19]

A key strength of BERTScore is its strong alignment with human evaluations across diverse text generation tasks. By leveraging deep contextual representations, it can more accurately assess the meaning and relevance of generated text compared to surface-level metrics. This characteristic makes BERTScore particularly valuable for abstractive summarization, where the aim is to produce summaries that are not only factually correct but also express content using potentially novel and varied phrasing relative to the source material. Furthermore, BERTScore's language-agnostic nature allows it to be applied across multiple languages by utilizing appropriate multilingual pre-trained models, enhancing its versatility in various NLP applications.

It is worth noting, however, that BERTScore comes with computational considerations. The process of computing contextual embeddings and pairwise similarities can be resource-intensive, particularly for lengthy texts. Additionally, the effectiveness of BERTScore evaluations is influenced by the choice of pre-trained model; models tailored to specific domains or updated with recent data may provide better correlation with human assessments. Despite these factors, BERTScore represents a significant step forward in text evaluation metrics, offering a more semantically informed and context-aware assessment framework that addresses many shortcomings of traditional methods.

3.2.2 Extrinsic Metrics

Extrinsic metrics evaluate the usefulness of the generated summary in accomplishing a specific task. These metrics are often more practical and aligned with real-world applications. Examples include:

- **Task-based Evaluation:** Involves using the generated summary in a downstream task, such as question answering or information retrieval, and measuring the performance of that task. For example, one could assess how well a summarization model aids in answering questions based on the summary. Task-based evaluation

provides a concrete measure of the summary's effectiveness in practical scenarios and highlights the real-world utility of the summarization model [NP04].

- **User Studies:** Involves human evaluation where users rate the summaries based on their usefulness, relevance, and readability. This type of evaluation can provide insights into the subjective quality of the summaries. User studies are invaluable for capturing human preferences and perceptions, which are not always reflected in automated metrics. This method involves collecting feedback from diverse users, including domain experts and laypersons, to get a comprehensive understanding of summary quality [Sch+23].
- **Readability Metrics:** Measures the readability of the generated summaries using established readability indices such as Flesch-Kincaid or Gunning Fog index [Kin+75] [See13]. These metrics help ensure that the summaries are not only informative but also easy to read and understand.

The Translation Task

4

Building upon the exploration of the summarization task, this chapter focuses into another fundamental challenge within the NLP community: the *translation task* (e.g. [Koe09]). Machine translation (MT) refers to the automatic conversion of text from one language to another, aiming to preserve the meaning, tone, and nuances of the original text (see [Lop08]). As a cornerstone of multilingual communication and cross-linguistic information access, machine translation has garnered significant attention in both research and application domains. This chapter will discuss the problem statement, the metrics used to evaluate translation quality, and the role of deep learning models, particularly large language models (LLMs), in advancing the field of machine translation.

4.1 Problem Statement

The machine translation task seeks to accurately translate text from a source language into a target language. Unlike traditional methods that relied heavily on rule-based or statistical approaches, modern MT systems leverage deep learning techniques to achieve more fluent and contextually appropriate translations. The key challenge in machine translation lies in capturing the nuances of the source language, including syntax, semantics, idiomatic expressions, and cultural references, and rendering them correctly in the target language.

There are several types of machine translation systems, including:

- **Rule-based Machine Translation (RBMT):** This approach relies on linguistic rules and dictionaries to translate text. While it can handle specific grammatical structures, RBMT often struggles with the complexities and irregularities of natural language, leading to translations that may be literal but lack fluency and naturalness [Tor+19]. Some notable examples of Rule Based Machine Translation systems are Systran [Tom77], Lucy LT [AT03] and Apertium [FTR09].
- **Statistical Machine Translation (SMT):** SMT models use large parallel corpora to learn probabilities of word and phrase translations [Bro+93]. By leveraging statistical patterns, SMT can produce more flexible translations than RBMT, but it may still fail to capture long-range dependencies and contextual nuances.
- **Neural Machine Translation (NMT):** The advent of NMT marked a significant shift in the field, with models like seq2seq (sequence-to-sequence) and transformer architectures becoming the standard [SVL14]. NMT models, particularly those based on transformers, use deep neural networks to map sequences in the source language to sequences in the target language, allowing for more accurate and fluent translations that account for context across entire sentences or paragraphs.

The recent advancements in Large Language Models (LLMs) have further enhanced the capabilities of NMT systems. LLMs, such as GPT and BERT (e.g. [Dev+18]; [Rad+19]), have shown remarkable proficiency in understanding and generating human language across various contexts. When applied to translation, these models can capture subtle linguistic nuances and produce translations that are not only accurate but also stylistically consistent with the source material. However, challenges remain, particularly in handling low-resource languages, translating idiomatic expressions, and ensuring the preservation of cultural context.

4.2 Translation Metrics

Evaluating the quality of machine translations is a critical aspect of MT research and deployment. Several metrics have been developed to assess various dimensions of translation quality [MB14], ranging from word-level accuracy to semantic adequacy and fluency. These metrics can be broadly classified into automatic metrics and human evaluation methods.

4.2.1 Automatic Metrics

Automatic metrics provide a quick and often reliable assessment of translation quality by comparing the machine-generated translation with one or more reference translations. Some of the most widely used automatic metrics include:

- **BLEU (Bilingual Evaluation Understudy):** BLEU [Pap+02] is one of the oldest and most widely used metrics in machine translation [Pos18]. It calculates the precision of $n-$grams (up to a specified length) between the generated translation and the reference translation(s). For more details see Chapter 3.2.1.
- **METEOR (Metric for Evaluation of Translation with Explicit Ordering):** METEOR improves upon BLEU by incorporating synonymy, stemming, and word order penalties, making it more sensitive to the nuances of translation. METEOR provides a more comprehensive evaluation by considering both precision and recall, and it often correlates better with human judgment compared to BLEU [BL05]. For more details see Chapter 3.2.1.
- **TER (Translation Edit Rate):** TER measures the number of edits required to change the machine-generated translation into the reference translation. These edits include insertions, deletions, and substitutions. A lower TER indicates a closer match to the reference translation. TER is particularly useful for understanding the effort needed to post-edit translations in professional settings [Sno+06].
- **BERTScore:** Similar to its application in summarization, BERTScore uses contextual embeddings to compare the similarity between the generated translation and the reference translation. By evaluating translations at a semantic level, BERTScore can capture nuances that $n-$gram based metrics like BLEU might miss. This makes it particularly valuable for assessing translations that require deep contextual understanding [Zha+19]. For more details see Chapter 3.2.1

While automatic metrics are valuable for their efficiency and objectivity, they are not without limitations. For example, these metrics may struggle to accurately evaluate translations with significant linguistic or cultural differences from the reference. Additionally, they may not adequately penalize translations that are grammatically correct but semantically incorrect or culturally inappropriate.

4.2.2 Human Evaluation

Human evaluation remains the gold standard for assessing translation quality [Koe09], as it allows for a nuanced understanding of the translation's accuracy, fluency, and appropriateness. Common methods of human evaluation include:

- **Adequacy and Fluency Ratings:** Human evaluators rate translations on scales of adequacy (how well the translation conveys the meaning of the source text) and fluency (how natural and grammatically correct the translation is in the target language). This method provides direct insights into the quality of the translation and its suitability for practical use [Cal+07].
- **Direct Assessment (DA):** DA involves human raters evaluating translations on an absolute scale, typically from 0 to 100, without reference to other translations. This method has gained popularity for its simplicity and effectiveness in capturing human judgment of translation quality [GBM15].

Human evaluation, while comprehensive, is time-consuming and resource-intensive. It is often used in combination with automatic metrics to provide a balanced assessment of translation quality.

4.3 The Role of Large Language Models in Translation

Large Language Models (LLMs) have revolutionized the field of machine translation by leveraging vast amounts of multilingual data and sophisticated architectures [Vas+17] to produce high-quality translations. Unlike traditional models, LLMs can generate translations that are not only accurate but also contextually and culturally appropriate.

4.3.1 Contextual Understanding

LLMs excel in understanding context across long text sequences, enabling them to translate complex sentences and paragraphs more accurately. For instance, transformer-based models, such as GPT and BERT, use self-attention mechanisms to capture dependencies between words regardless of their distance in the text. This allows them to handle syntactic variations and word order differences between languages more effectively [Vas+17].

4.3.2 Handling Low-Resource Languages

One of the significant challenges in machine translation is the translation of low-resource languages, for which there is limited training data available [Zop+16]. LLMs, pre-trained on large multilingual corpora, can transfer knowledge from high-resource languages to low-resource ones, improving translation quality in these challenging cases. Additionally, techniques like fine-tuning on specific language pairs or domains can further enhance the performance of LLMs in low-resource settings.

4.3.3 Preserving Cultural and Idiomatic Expressions

LLMs are also capable of preserving cultural nuances and translating idiomatic expressions more effectively than traditional models. By leveraging large amounts of data, these models learn to recognize and appropriately translate phrases that may not have direct equivalents in the target language, ensuring that the translation is not only accurate but also culturally sensitive.

4.3.4 Challenges and Future Directions

Despite their strengths, LLMs in machine translation are not without challenges. Issues such as model bias, hallucinations, and the high computational cost of training and inference remain areas of concern. Future research may focus on addressing these challenges through methods such as model distillation [HVD15], debiasing techniques [MPR21], and the development of more efficient architectures.

Overall, the integration of LLMs into machine translation systems has significantly improved the quality and applicability of machine-generated translations, paving the way for more effective and widespread use of MT in real-world applications.

Experimental Setup and Results 5

In this section, we describe the experimental setup used to evaluate our model in summarization and machine translation tasks, including the datasets, training procedures, and evaluation protocols. We also present the results of our experiments.

5.1 Hardware & Software

For our experiments, we are using Tensor Process Units (TPUs) as a hardware, specifically 8 TPUv4-8 units. That is because, TPUs have the fastest completion time than the alternative options like Central Processing Units (CPUs) or Graphics Processing Units (GPUs), and given the magnitude of the experiments that needed to be run, fast completion time was crucial.

One of the limitations of utilizing TPUs as hardware for machine learning experiments is that the code required for such tasks extends beyond frameworks like *PyTorch*[1] or APIs provided by *Hugging Face*[2]. In this work, we have selected *Python* version 3.10[3] as the programming language, and our experiments are developed using the *PyTorch Lightning* framework[4]. *PyTorch Lightning* provides an abstraction that alleviates the need to write different versions of code depending on the underlying hardware, allowing us to concentrate more on the scientific aspects of the code. These aspects include the design of the model architecture, the implementation of training, validation, and testing procedures, as well as the preprocessing and distribution of the dataset across TPU cores in each TPUv4-8, facilitating distributed learning. In addition, the code used for running the experiments on TPUs is available on GitHub[5] for reproducibility and further exploration.

5.2 Datasets and Preprocessing

We used several benchmark datasets for summarization tasks, including the CNN/DailyMail [SLM17] [Her+15] dataset, the XSum dataset [NCL18], and the SAMSum dataset [Gli+19] and for the translation tasks we used the Flores [NLL22] and IWSLT [Cet+17] datasets. We summarize the characteristics of those datasets below.

5.2.1 CNN/DailyMail Dataset

The CNN/DailyMail dataset comprises over 300k unique news articles in English, written by journalists at CNN and DailyMail. This dataset is categorized under abstractive

[1] https://pytorch.org/docs/stable/index.html
[2] https://huggingface.co/docs
[3] https://www.python.org/downloads/release/python-31012/
[4] https://lightning.ai/docs/pytorch/stable/
[5] https://github.com/PavlosPo/nlp-optimizers-aueb-2.git

summarization. Additionally, Bordia and Bowman [BB19] reported that it exhibits minimal gender bias, aligning well with the specific requirements of our experiment.

For preprocessing, we selected a subset of 30k articles for training, 3k for validation, and 3k for testing per split, using various seed numbers to generate different subsets from the original dataset. This approach allowed our model to encounter diverse examples, enhancing its exposure to the dataset.

Our batch size configuration was constrained by equipment resources. Each TPU core processed a batch size of 16, resulting in an effective batch size of 64 with four cores per TPU. This setup aligned with the dataset splits and met the computational limits of our available hardware.

5.2.2 XSum

The XSum dataset consists of real-world online articles from the British Broadcasting Corporation (BBC), with each article accompanied by a single-sentence summary, making it particularly well-suited for abstractive summarization tasks.

This dataset was partially included in the pretraining corpus of our chosen model (see Section 5.3). As described by Raffel et al. [Raf+19], the pretrained model was trained on the "Colossal Clean Crawled Corpus" (C4), a vast dataset containing diverse text. However, Dodge et al. [Dod+21] found that only about 15% of the XSum dataset overlaps with C4, meaning XSum was only partially encountered during pretraining. Despite this overlap, we included XSum in our evaluation to introduce variability and facilitate comparisons with fully unseen datasets, such as CNN/Daily Mail and SAMSum (see Sections 5.2.1 and 5.2.3).

For preprocessing, we adopted the same data split strategy used for CNN/DailyMail, selecting 30k articles for training, 3k for validation, and 3k for testing. We then exposed the model to a broader range of examples by generating splits based on varying seed numbers. The batch size per TPU core remains set at 16, resulting in the same total batch size of 64, aligning the total training steps with those used for the CNN/DailyMail dataset.

5.2.3 SAMSum

The SAMSum dataset [Gli+19] differs from the XSum and CNN/DailyMail datasets by containing conversational, messenger-style exchanges with corresponding summaries. These conversations, crafted by linguists fluent in English, reflect diverse language styles, including slang, emoticons, and typos. Created by the Samsung R&D Institute of Poland and distributed for research purposes, the dataset consists of 16k training examples. Summaries are concise third-person accounts of conversation content.

Given the dataset's smaller size, we used the entire dataset for each seed, maintaining a batch size of 16 per TPU core, or 64 in total. Examples were shuffled based on the seed number, yielding final splits of 14, 732 examples for training, and 816 examples each for validation and testing.

5.2.4 Flores

The Flores dataset, specifically the Flores-200 dataset [NLL22], is an updated version of Flores-101 [Goy+21]; [Guz+19] and is one of the datasets we selected for our translation task. Developed by Facebook, the Flores-101 and its expanded Flores-200 cover over 200 languages, providing a comprehensive multilingual resource.

Flores-200 consists of more than 3k professionally translated sentences sourced from English-language Wikimedia projects such as Wikinews, Wikijunior, and Wikivoyage, among others. Each English sentence was translated into the other 200+ languages by expert translators, creating a many-to-many multilingual dataset.

Facebook upholds a strict quality assurance process for Flores-200, ensuring high standards. The workflow (see Figure 5.1) involves professional translators and reviewers aligning on language standards, translating the dataset, performing automated checks, and conducting final reviews. If the quality assessment score exceeds 90%, the language is considered ready for inclusion in Flores-200.

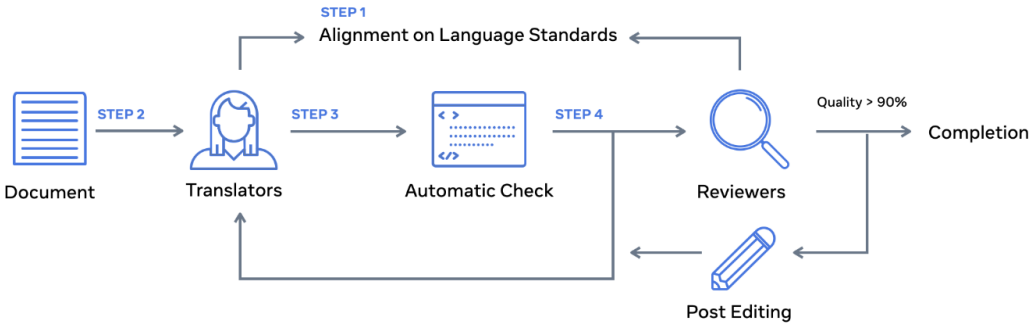

Fig. 5.1: A complex, multi-step workflow to ensure quality in the FLORES-200 dataset. First, professional translators and reviewers aligned on language standards. Next, translators translated all the Flores-200 sentences, followed by automated checks. Finally, independent reviewers assessed the quality, and based on their evaluation, some translations were sent for post-editing. [NLL22]

The average sentence length in Flores-200 is 21 words [NLL22], which aligns well with the T5-Small model's average generated length ($20 + 2$ tokens). We combined the two published splits (devtest and dev), then shuffled and partitioned them into 80% for training, 10% for validation, and 10% for testing. The batch size was set to 16 per TPU core, with four TPU cores per machine, consistent with the setup of the other datasets.

5.2.5 IWSLT 2017

The International Workshop on Spoken Language Translation (IWSLT) dataset [Cet+17] was another dataset we used for our machine translation task. Specifically, we used the 2017 version, a large dataset designed to support research in spoken language translation.

The IWSLT dataset was developed for annual IWSLT evaluation campaigns to improve machine translation, especially for spoken language. It includes various languages and

mirrors real-world translation situations, making it valuable for researchers and developers in natural language processing.

A key feature of the IWSLT dataset is its focus on spoken language, which brings challenges like handling speech recognition errors, informal language, and the variety in spoken conversations. The dataset includes transcripts of spoken language and their translations, covering languages such as English, German, French, and Romanian, which we use in our experiments. Most of the data comes from TED talks, which provide a wide range of topics and speaking styles. Using the IWSLT dataset in our model training adds diversity to our experiments due to the broad topics and different speaking styles found in the transcripts.

For preprocessing, we selected 10 training examples per language pair: English-German, English-French, and English-Romanian, resulting in a total of 30k training examples. For the validation and test subsets, we used the default splits provided by the Hugging Face API[6]. The validation subset has 888 examples per language pair (totaling 2.664), and the test subset includes over 8k examples per language pair, or about 25k test examples. We set the batch size to 16 per TPU core, and the data splits were shuffled and partitioned according to the seed number.

5.3 Models

The summarization model used in Gkouti et al.'s work [Gko+24] was DistilBERT [San+20], which is a smaller version of the BERT transformer model [Dev+18]. In this thesis we expand to more type of architectures, like the T5 family of models [Raf+19].

The smaller T5-Small model was chosen for the experiments. It is a transformer-based model that has an encoder and decoder Transformer and consists of 60 million parameters, 8-headed attention, and 6 layers each in the encoder and decoder respectfully. In addition, the input size is 512 input tokens, compared to 1.024 that the T5-Large has.

The architecture of all T5 model variations provides a simple way to train a single model on a wide variety of NLP tasks by converting them to sequence-to-sequence tasks, therefore using the same loss function and decoding procedures. Specifically, T5-small was pretrained on tasks like summarization and machine translation from English to German, French or Romanian.

We selected the smaller T5 model, for two reasons. The first reason, is to expand the variety of architectures in our experiments, by including an Encoder-Decoder Transformer model. The second reason is due to the small number of parameters it has, making it suitable for our equipment budget.

5.4 Choosing Optimizers

The selection of optimization algorithms is a critical aspect of our experimental design, directly influencing the performance and generalization capabilities of our summarization

[6] `https://huggingface.co/datasets/IWSLT/iwslt2017`

models. In this section, we outline our rationale for choosing specific optimizers, focusing on their relevance to our research objectives and their standing in the NLP community.

Our experiments involved five distinct optimizers: SGD as the baseline, SGDM (SGD with Momentum), Adam, as the most popular choice, and AdamW along with NAdam as improvements over Adam. The selection of these optimizers was driven by several key considerations:

- **SGD (Stochastic Gradient Descent):** Despite its simplicity, SGD (see Section 2.3.1) remains a cornerstone in optimization research. We included SGD to provide a fundamental baseline against which to measure the performance of more complex optimizers. Having said that, SGD was also chosen because we wanted to explore whether the simplicity of SGD offers any unexpected advantages in our summarization task, particularly in terms of generalization.
- **SGDM (SGD with Momentum):** The addition of SGDM (see Section 2.3.2) to our experiments was driven by its historical importance and continued relevance. Our goals were to assess the impact of momentum in a non-adaptive optimization context for summarization tasks. We also wanted to compare the performance of a classical momentum-based approach against more modern adaptive methods.
- **Adam (Adaptive Moment Estimation):** We chose Adam (see Section 2.4.4) primarily due to its widespread adoption in the NLP research community and its proven efficiency across a wide range of tasks. Our aim was to establish a strong baseline using a well-established optimizer..
- **AdamW:** We included AdamW (see Section 2.4.8) to investigate whether the reported benefits of decoupled weight decay offer significant advantages for text summarization or machine translation tasks. Additionally, comparing AdamW with standard Adam enables us to assess whether the added complexity of AdamW yields measurable improvements in our specific use case.
- **NAdam (Nesterov-accelerated Adaptive Moment Estimation):** We selected NAdam (see Section 2.4.5) to round out our comparison of adaptive methods. The inclusion of NAdam allowed us to evaluate whether the theoretical advantages of Nesterov momentum translate to practical improvements in our model training shceme. Another reason is to investigate whether the combination of Nesterov momentum and adaptive learning rates offers synergistic benefits not captured by Adam or SGDM individually.

In conclusion, focusing on SGD, SGDM, Adam, AdamW, and NAdam, our optimizer selection aims to provide a comprehensive yet focused comparison of optimization strategies in the context of text summarization. This selection allows us to investigate, firstly, the efficacy of adaptive versus non-adaptive methods in summarization and machine translation tasks. Secondly, the impact of different forms of momentum (classical and Nesterov) in both adaptive and non-adaptive contexts, and, thirdly, the potential benefits

of more recent developments in optimization algorithms (e.g., AdamW) compared to well-established methods.

5.5 Methodology in Experiments

Our experimental methodology is designed to systematically evaluate the impact of hyperparameter tuning on the performance of our model, particularly in the context of different optimizers. The experiments are structured into two distinct phases of hyperparameter tuning.

The first phase focuses exclusively on tuning the learning rate for each optimizer, which we will be referring to as "Basic Mode". This is a critical step, as the learning rate is one of the most influential hyperparameters, directly affecting the model's ability to converge during training. By isolating the learning rate, we can establish a clear understanding of its optimal value across different datasets and optimizers, providing a strong baseline for further tuning.

In the second phase, we expand the scope of hyperparameter tuning to include additional parameters such as beta coefficients, epsilon values, and momentum (where applicable). This phase aims to investigate whether fine-tuning these additional hyperparameters, beyond just the learning rate, offers any significant performance improvements in the tasks that we are considering. We will be referring to this type of training as "Full Mode". The inclusion of these parameters allows us to explore a more nuanced optimization landscape, potentially uncovering interactions between hyperparameters that could lead to better model performance.

To ensure the robustness and generalizability of our findings, we perform our experiments across multiple seeds. Specifically, for the CNN/DailyMail and XSum datasets, we generate four different subsets by splitting the data according to four different random seeds. These different seeds introduce variation not only in data splits but also in weight initializations. For the SAMSum dataset, which contains a smaller number of observations, we use the entire dataset for each seed but apply different shuffling strategies to introduce variability. When it comes to the Flores and IWSLT datasets, we use the same data splits as the ones provided by the Hugging Face API. This approach ensures that our models are exposed to diverse training examples and weight configurations, mitigating the risk of overfitting to any particular subset of data or initialization.

The training is conducted on TPUv4-8 hardware, leveraging its parallel processing capabilities to distribute the training workload across multiple cores. Each TPU has 4 cores, and we use a batch size of 16 per core, effectively yielding a batch size of 64 per training step. This setup not only accelerates the training process but also facilitates the handling of large datasets, which is crucial given the computational demands of hyperparameter tuning.

Hyperparameter tuning is performed using the Optuna framework[7], which is well-suited for automated optimization in high-dimensional spaces. We have chosen the base hyperparameter tuning strategy offered by Optuna, which is the same as the one used in Gkouti et al.'s work [Gko+24]. This strategy, called the TPESampler (Tree-structured Parzen Estimator sampler) [Wat23], utilizes Bayesian optimization through kernel fitting. The TPE sampler is used by default and aims to sample hyperparameter candidates by improving on the last trial's scores, expecting incremental (or marginal) improvements with each trial. Optuna's ability to dynamically adjust its search strategy based on intermediate results allows for more efficient exploration of the hyperparameter space. We log the results using a combination of TensorBoard[8] and WandB (Weights and Biases)[9], which provides real-time visualization of key metrics, aiding in the monitoring and analysis of the tuning process.

Upon completion of the hyperparameter tuning, the best-performing configurations are saved and used to train the final models. These models are then evaluated on a held-out test set that was not seen during training and hyperparameter-tuning, ensuring that the reported results reflect the models' true generalization capabilities. This two-phase experimental design, allows us to draw meaningful conclusions about the relative effectiveness of different optimization strategies in the context of text summarization and machine translation.

5.6 Hyperparameter Tuning

Hyperparameter tuning plays a pivotal role in optimizing the performance of machine learning models, particularly in complex tasks such as text summarization and machine translation. In this section, we discuss the experiments conducted to tune the hyperparameters of our models and the strategies employed to explore the hyperparameter search spaces effectively.

5.6.1 Basic Mode

The initial phase of our hyperparameter tuning focused exclusively on the learning rate for each dataset: CNN/DailyMail, XSum, SAMSum, IWSLT and Flores. Learning rate is a critical hyperparameter that significantly influences model convergence and overall performance. We conducted an extensive search over a range of learning rates to identify the optimal value for each dataset. This approach allowed us to establish a strong baseline for each model before proceeding to the the Full Mode in which we tune more available hyperparameters of each optimizer.

5.6.2 Full Mode

In contrast to the initial phase during which we only tune the learning rate, we expanded our hyperparameter search to include additional variables.

[7] `https://optuna.org`
[8] `https://www.tensorflow.org/tensorboard`
[9] `https://wandb.ai/`

For each optimizer, we defined a specific search space that included key hyperparameters such as learning rate, beta coefficients (which control the exponential decay rates for the moment estimates), epsilon (which prevents division by zero), and momentum (for optimizers that support it). The search spaces we run our experiments in, for each optimizer, are summarized in Table 5.1 and the default values of each optimizer are summarized in Table 5.2, which are the same default search spaces as in the Gkouti's et al work.

Tab. 5.1: Hyperparameter Search Space for Each Optimizer

Optimizer	Learning Rate	Beta1	Beta2	Epsilon	Momentum
Adam	$[1e-07, 0.001]$	$[0.8, 0.95]$	$[0.9, 0.99999]$	$[1e-09, 1e-07]$	N/A
AdamW	$[1e-07, 0.001]$	$[0.8, 0.95]$	$[0.9, 0.99999]$	$[1e-09, 1e-07]$	N/A
NAdam	$[1e-07, 0.001]$	$[0.8, 0.95]$	$[0.9, 0.99999]$	$[1e-09, 1e-07]$	$[0.0001, 0.01]$
SGD	$[1e-07, 0.001]$	N/A	N/A	N/A	N/A
SGDM	$[1e-07, 0.001]$	N/A	N/A	N/A	$[0.7, 0.99999]$

Tab. 5.2: Default Hyperparameters for Each Optimizer

Optimizer	Learning Rate	Beta1	Beta2	Epsilon	Momentum
Adam	$1e-3$	0.9	0.999	$1e-8$	N/A
AdamW	$1e-3$	0.9	0.999	$1e-8$	N/A
NAdam	$2e-3$	0.9	0.999	$1e-8$	$4e-3$
SGD	$1e-3$	N/A	N/A	N/A	N/A
SGDM	$1e-3$	N/A	N/A	N/A	0.9

5.6.3 Search Space Selection

The learning rate search space was set between $1e-07$ and 0.001 across all optimizers, ensuring that we captured a wide spectrum of possible optimal values without risking the model becoming unstable due to overly large learning rates. Similarly, the beta coefficients (Beta1 and Beta2) were explored within ranges that balance between faster convergence and stability.

The inclusion of momentum as a search parameter for NAdam and SGDM was intended to evaluate its impact on models trained on text summarization and machine translation tasks. Momentum is known to help models escape local minima and can improve convergence speed, making it a valuable hyperparameter to tune.

5.7 Hyperparameter Tuning Results

In Subsections 5.7.1 through 5.7.3, and Tables 5.3 through 5.17, we present the results of our hyperparameter search conducted across the SAMSum, XSum, CNN/DailyMail, Flores and IWSLT datasets.

5.7.1 SAMSum Dataset Results

Table 5.3 reports the learning rate results for each optimizer on the SAMSum dataset, aggregated over 4 different seed values. Among the optimizers, NAdam has the lowest mean learning rate, while SGDM has the highest.

Basic Mode

Tab. 5.3: Learning rates for each optimizer in the SAMSum Dataset, based on 4 different seed numbers

Optimizer	Learning Rate
Adam	$9.5 \times 10^{-4} (\pm 1.2 \times 10^{-5})$
AdamW	$9.7 \times 10^{-4} (\pm 8.8 \times 10^{-6})$
NAdam	$9.3 \times 10^{-4} (\pm 1.7 \times 10^{-5})$
SGD	$9.7 \times 10^{-4} (\pm 1.4 \times 10^{-5})$
SGDM	$9.8 \times 10^{-4} (\pm 5.8 \times 10^{-6})$

Full Mode

For the Full Mode results on the SAMSum dataset, Tables 5.4 and 5.5 present the mean values of the hyperparameters for each optimizer. The reported mean learning rates indicate that the adaptive optimizers favor smaller learning rates compared to the non-adaptive optimizers, a distinction not observed in Basic Mode.

Tab. 5.4: Hyperparameter Results for dataset SAMSum (Part 1)

Optimizer	Learning Rate	Beta1
Adam	$3.1 \times 10^{-4}(\pm 3.1 \times 10^{-5})$	$8.5 \times 10^{-1}(\pm 1.1 \times 10^{-2})$
AdamW	$4.8 \times 10^{-4}(\pm 2.7 \times 10^{-5})$	$8.9 \times 10^{-1}(\pm 6.5 \times 10^{-3})$
NAdam	$4.1 \times 10^{-4}(\pm 3.6 \times 10^{-5})$	$8.6 \times 10^{-1}(\pm 2.1 \times 10^{-2})$
SGD	$9.9 \times 10^{-4}(\pm 4.0 \times 10^{-6})$	N/A
SGDM	$9.8 \times 10^{-4}(\pm 5.8 \times 10^{-6})$	N/A

Tab. 5.5: Hyperparameter Results for dataset SAMSum (Part 2)

Optimizer	Beta2	Epsilon	Momentum
Adam	$9.4 \times 10^{-1}(\pm 7.4 \times 10^{-3})$	$6.5 \times 10^{-9}(\pm 2.6 \times 10^{-9})$	N/A
AdamW	$9.4 \times 10^{-1}(\pm 1.2 \times 10^{-2})$	$8.6 \times 10^{-9}(\pm 7.0 \times 10^{-9})$	N/A
NAdam	$9.2 \times 10^{-1}(\pm 1.4 \times 10^{-2})$	$5.3 \times 10^{-9}(\pm 3.1 \times 10^{-9})$	$7.0 \times 10^{-3}(\pm 9.9 \times 10^{-4})$
SGD	N/A	N/A	N/A
SGDM	N/A	N/A	$8.6 \times 10^{-1}(\pm 3.0 \times 10^{-2})$

5.7.2 XSum Dataset Results

Basic Mode

For the XSum dataset, Table 5.6 presents the mean learning rate for each optimizer. Overall, the mean learning rates for the adaptive optimizers and SGD are similar; however, the mean learning rate for SGDM is slightly lower than the others while having a higher deviation, too.

Tab. 5.6: Learning rates for each optimizer in the XSum Dataset, based on 4 different seed numbers

Optimizer	Learning Rate
Adam	$9.7 \times 10^{-4}(\pm 1.9 \times 10^{-5})$
AdamW	$9.1 \times 10^{-4}(\pm 1.9 \times 10^{-5})$
NAdam	$9.4 \times 10^{-4}(\pm 1.7 \times 10^{-5})$
SGD	$9.9 \times 10^{-4}(\pm 2.4 \times 10^{-6})$
SGDM	$7.9 \times 10^{-4}(\pm 7.2 \times 10^{-5})$

Full Mode

In Full Mode for the XSum dataset, the results are shown in Tables 5.7 and 5.8. As observed with the SAMSum dataset (see Section 5.7.1), the mean learning rates for the adaptive optimizers decreased compared to Basic Mode; this trend is also present in the XSum dataset. Interestingly, for the SGDM optimizer, the mean learning rate increased.

Tab. 5.7: Hyperparameter Results for dataset XSum (Part 1)

Optimizer	Learning Rate	Beta1
Adam	$4.3 \times 10^{-4} (\pm 3.4 \times 10^{-5})$	$8.5 \times 10^{-1} (\pm 2.5 \times 10^{-2})$
AdamW	$4.8 \times 10^{-4} (\pm 3.5 \times 10^{-5})$	$8.7 \times 10^{-1} (\pm 1.4 \times 10^{-2})$
NAdam	$4.5 \times 10^{-4} (\pm 1.9 \times 10^{-5})$	$8.9 \times 10^{-1} (\pm 2.3 \times 10^{-2})$
SGD	$9.9 \times 10^{-4} (\pm 1.9 \times 10^{-6})$	N/A
SGDM	$9.8 \times 10^{-4} (\pm 1.3 \times 10^{-5})$	N/A

Tab. 5.8: Hyperparameter Results for dataset XSum (Part 2)

Optimizer	Beta2	Epsilon	Momentum
Adam	$9.1 \times 10^{-1} (\pm 3.9 \times 10^{-3})$	$1.2 \times 10^{-8} (\pm 9.8 \times 10^{-9})$	N/A
AdamW	$9.5 \times 10^{-1} (\pm 1.3 \times 10^{-2})$	$6.4 \times 10^{-9} (\pm 1.6 \times 10^{-9})$	N/A
NAdam	$9.4 \times 10^{-1} (\pm 1.7 \times 10^{-2})$	$2.5 \times 10^{-8} (\pm 1.2 \times 10^{-8})$	$4.6 \times 10^{-3} (\pm 1.7 \times 10^{-3})$
SGD	N/A	N/A	N/A
SGDM	N/A	N/A	$8.0 \times 10^{-1} (\pm 3.8 \times 10^{-2})$

5.7.3 CNN/Dailymail Dataset Results

Basic Mode

Table 5.9 presents the mean learning rates for each optimizer on the CNN/DailyMail dataset in Basic Mode. The mean learning rates for all adaptive optimizers are similar to each other but notably smaller than those of the non-adaptive optimizers. This differs from the Basic Mode results for the SAMSum and XSum datasets (see Sections 5.7.1 and 5.7.2).

Tab. 5.9: Learning rates for each optimizer in the CNN/Dailymail Dataset

Optimizer	Learning Rate
Adam	$2.0 \times 10^{-4} (\pm 2.1 \times 10^{-5})$
AdamW	$2.5 \times 10^{-4} (\pm 3.1 \times 10^{-5})$
NAdam	$2.3 \times 10^{-4} (\pm 2.5 \times 10^{-5})$
SGD	$9.8 \times 10^{-4} (\pm 8.3 \times 10^{-6})$
SGDM	$9.7 \times 10^{-4} (\pm 3.3 \times 10^{-6})$

Full Mode

Tables 5.10 and 5.11 display the mean hyperparameter values for the CNN/DailyMail dataset in Full Mode. The mean learning rates for the adaptive optimizers continue to be lower than those for the non-adaptive optimizers, consistent with observations in the Full Mode of the XSum and SAMSum datasets.

Tab. 5.10: Hyperparameter Results for dataset CNN/Dailymail (Part 1)

Optimizer	Learning Rate	Beta1
Adam	$1.1 \times 10^{-4}(\pm 1.9 \times 10^{-5})$	$8.4 \times 10^{-1}(\pm 2.4 \times 10^{-2})$
AdamW	$1.1 \times 10^{-4}(\pm 5.8 \times 10^{-6})$	$8.8 \times 10^{-1}(\pm 2.3 \times 10^{-2})$
NAdam	$1.0 \times 10^{-4}(\pm 2.2 \times 10^{-5})$	$9.0 \times 10^{-1}(\pm 2.6 \times 10^{-2})$
SGD	$9.8 \times 10^{-4}(\pm 8.2 \times 10^{-6})$	N/A
SGDM	$9.1 \times 10^{-4}(\pm 2.7 \times 10^{-5})$	N/A

Tab. 5.11: Hyperparameter Results for dataset CNN/Dailymail (Part 2)

Optimizer	Beta2	Epsilon	Momentum
Adam	$9.3 \times 10^{-1}(\pm 4.4 \times 10^{-3})$	$3.2 \times 10^{-8}(\pm 2.2 \times 10^{-8})$	N/A
AdamW	$9.3 \times 10^{-1}(\pm 4.4 \times 10^{-3})$	$7.2 \times 10^{-9}(\pm 2.1 \times 10^{-9})$	N/A
NAdam	$9.3 \times 10^{-1}(\pm 6.4 \times 10^{-3})$	$2.5 \times 10^{-8}(\pm 1.5 \times 10^{-8})$	$3.2 \times 10^{-3}(\pm 1.2 \times 10^{-3})$
SGD	N/A	N/A	N/A
SGDM	N/A	N/A	$8.7 \times 10^{-1}(\pm 2.2 \times 10^{-2})$

5.7.4 Flores Dataset Results

Basic

Table 5.12 reports the mean learning rate values for the Flores dataset in Basic Mode. Interestingly, the adaptive optimizers exhibit higher mean values, while the non-adaptive optimizers have lower ones, which is the opposite of the trends observed in the Basic Mode for the SAMSum, XSum, and CNN/DailyMail datasets.

Tab. 5.12: Learning rates for each optimizer in the Flores Dataset

Optimizer	Learning Rate
Adam	$9.6 \times 10^{-4}(\pm 2.8 \times 10^{-5})$
AdamW	$9.5 \times 10^{-4}(\pm 1.9 \times 10^{-5})$
NAdam	$9.0 \times 10^{-4}(\pm 3.4 \times 10^{-5})$
SGD	$7.6 \times 10^{-5}(\pm 5.1 \times 10^{-6})$
SGDM	$2.5 \times 10^{-4}(\pm 2.4 \times 10^{-4})$

Full Mode

Tables 5.13 and 5.14 present the hyperparameter search results in Full Mode for the Flores dataset. The mean learning rates for all optimizers fall within a similar range; however, SGDM shows the highest variability in error, while SGD has the lowest mean value. This pattern differs from the Full Mode results observed in the previous datasets.

Tab. 5.13: Hyperparameter Results for dataset Flores (Part 1)

Optimizer	Learning Rate	Beta1
Adam	$3.4 \times 10^{-4} (\pm 8.2 \times 10^{-5})$	$8.2 \times 10^{-1} (\pm 8.4 \times 10^{-3})$
AdamW	$2.1 \times 10^{-4} (\pm 2.8 \times 10^{-5})$	$8.4 \times 10^{-1} (\pm 9.7 \times 10^{-3})$
NAdam	$3.0 \times 10^{-4} (\pm 4.2 \times 10^{-5})$	$8.6 \times 10^{-1} (\pm 2.0 \times 10^{-2})$
SGD	$8.0 \times 10^{-5} (\pm 5.6 \times 10^{-6})$	N/A
SGDM	$2.4 \times 10^{-4} (\pm 2.3 \times 10^{-4})$	N/A

Tab. 5.14: Hyperparameter Results for dataset Flores (Part 2)

Optimizer	Beta2	Epsilon	Momentum
Adam	$9.4 \times 10^{-1} (\pm 1.2 \times 10^{-2})$	$5.0 \times 10^{-8} (\pm 2.0 \times 10^{-8})$	N/A
AdamW	$9.2 \times 10^{-1} (\pm 5.0 \times 10^{-3})$	$1.9 \times 10^{-8} (\pm 9.2 \times 10^{-9})$	N/A
NAdam	$9.2 \times 10^{-1} (\pm 1.6 \times 10^{-2})$	$1.4 \times 10^{-8} (\pm 2.4 \times 10^{-9})$	$2.8 \times 10^{-3} (\pm 1.0 \times 10^{-3})$
SGD	N/A	N/A	N/A
SGDM	N/A	N/A	$7.9 \times 10^{-1} (\pm 3.2 \times 10^{-2})$

5.7.5 IWSLT Dataset Results

Basic Mode

Table 5.15 reports the mean learning rate values for the IWSLT dataset in Basic Mode. The adaptive optimizers have lower mean values than the non adaptive optimizers.

Tab. 5.15: Learning rates for each optimizer in the IWSLT Dataset

Optimizer	Learning Rate
Adam	$3.1 \times 10^{-4} (\pm 2.7 \times 10^{-5})$
AdamW	$3.3 \times 10^{-4} (\pm 1.3 \times 10^{-5})$
NAdam	$3.2 \times 10^{-4} (\pm 8.2 \times 10^{-6})$
SGD	$9.9 \times 10^{-4} (\pm 0.0)$
SGDM	$9.7 \times 10^{-4} (\pm 8.7 \times 10^{-6})$

Full Mode

Tables 5.16 and 5.17 display the mean hyperparameter values for the IWSLT dataset in Full Mode. The mean learning rates for the adaptive optimizers are lower than those for the non-adaptive optimizers, consistent with results from the SAMSum, XSum, and CNN/DailyMail datasets. The only exception is the Flores dataset, where both adaptive and non-adaptive optimizers exhibited similar mean learning rate values.

Tab. 5.16: Hyperparameter Results for dataset IWSLT (Part 1)

Optimizer	Learning Rate	Beta1
Adam	$1.1 \times 10^{-4} (\pm 4.1 \times 10^{-6})$	$8.8 \times 10^{-1} (\pm 1.5 \times 10^{-2})$
AdamW	$1.2 \times 10^{-4} (\pm 6.3 \times 10^{-6})$	$8.4 \times 10^{-1} (\pm 1.3 \times 10^{-2})$
NAdam	$1.3 \times 10^{-4} (\pm 1.7 \times 10^{-5})$	$8.6 \times 10^{-1} (\pm 1.9 \times 10^{-2})$
SGD	$9.6 \times 10^{-4} (\pm 2.0 \times 10^{-5})$	N/A
SGDM	$8.0 \times 10^{-4} (\pm 1.3 \times 10^{-4})$	N/A

Tab. 5.17: Hyperparameter Results for dataset IWSLT (Part 2)

Optimizer	Beta2	Epsilon	Momentum
Adam	$9.4 \times 10^{-1} (\pm 1.3 \times 10^{-2})$	$1.4 \times 10^{-8} (\pm 6.7 \times 10^{-9})$	N/A
AdamW	$9.6 \times 10^{-1} (\pm 1.3 \times 10^{-2})$	$2.0 \times 10^{-8} (\pm 7.3 \times 10^{-9})$	N/A
NAdam	$9.5 \times 10^{-1} (\pm 1.0 \times 10^{-2})$	$3.7 \times 10^{-9} (\pm 1.2 \times 10^{-9})$	$6.5 \times 10^{-3} (\pm 1.2 \times 10^{-3})$
SGD	N/A	N/A	N/A
SGDM	N/A	N/A	$8.8 \times 10^{-1} (\pm 2.7 \times 10^{-2})$

5.8 Training Results and Discussion

In this section, we present the training curves based on the optimal hyperparameter configurations obtained from the hyperparameter search. The results are organized by dataset, training mode (Basic Mode vs. Full Mode), and optimizer. Each optimizer's performance is visualized with curves representing the mean values, with shaded areas denoting the standard deviation (STD) across multiple seeds. Notably, we observed that the optimizers SGD and SGDM required more than five epochs to deliver meaningful results. Therefore, while their results with five epochs are included for comparison with adaptive optimizers, a more detailed discussion of their performance over extended epochs is deferred to Section 5.10.

We now elaborate on the metrics employed in the following subsections. First, we plot the training loss (cross-entropy loss on the y-axis and steps on the x-axis) to track the model's learning progress. In addition, validation loss (cross-entropy loss on the y-axis and steps on the x-axis) is plotted.

For performance evaluation, we focus on ROUGE metrics, particularly the ROUGE-1 F-measure, as it balances precision and recall, providing a more holistic view of performance. While ROUGE-1 Recall and ROUGE-1 Precision are related metrics, we prioritize the F-measure for its ability to encapsulate both precision and recall (see Section 3.2.1).

Similarly, we include the ROUGE-2 F-measure, which considers bigram overlaps. By focusing on the F-measure for both ROUGE-1 and ROUGE-2, we provide a consistent evaluation metric that accounts for precision-recall trade-offs.

Finally, we report the BERTScore F1 metric. We compute this using the DeBERTa[10] model, with the F1 score automatically calculated based on the weighted relevance of words in context, this functionality of BERTScore was explained in earlier sections.

We start with the CNN/Dailymail's results.

[10] https://huggingface.co/microsoft/deberta-large-mnli

5.8.1 CNN/Dailymail

Basic Mode

As observed in the training mode where only the learning rate was tuned (Basic Mode), the adaptive optimizers NAdam, AdamW, and Adam exhibited similar performance in both training loss and validation loss (see Figures 5.2 and 5.3). In contrast, the performance of the SGD and SGDM optimizers, while similar to each other, lagged behind the adaptive optimizers in terms of both training and validation loss.

An interesting observation arises when examining the ROUGE-1, ROUGE-2 (F-measure), and BERTScore F1 metrics (see Figures 5.4, 5.5, and 5.6). Among the adaptive optimizers, NAdam demonstrates the fastest initial convergence. However, while NAdam reaches optimal performance slightly ahead of the others, all adaptive optimizers ultimately converge to similar performance levels, making this early advantage of NAdam insignificant due to the fast follow-up in performance by the other adaptive optimizers. Regarding the SGD and SGDM optimizers, their metrics are either not displayed or appear as zero on the graphs due to extremely slow progress or negligible values.

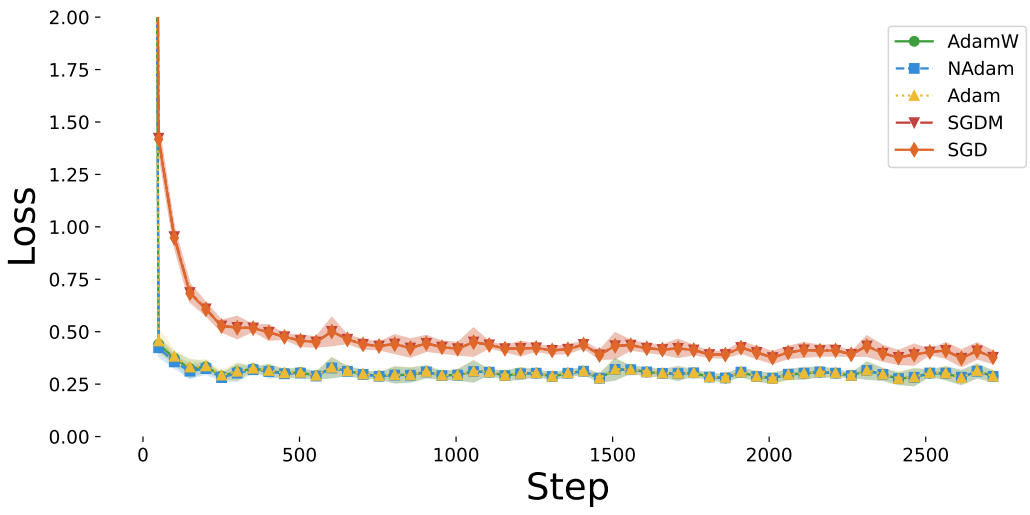

Fig. 5.2: Training Loss over 5 epochs for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

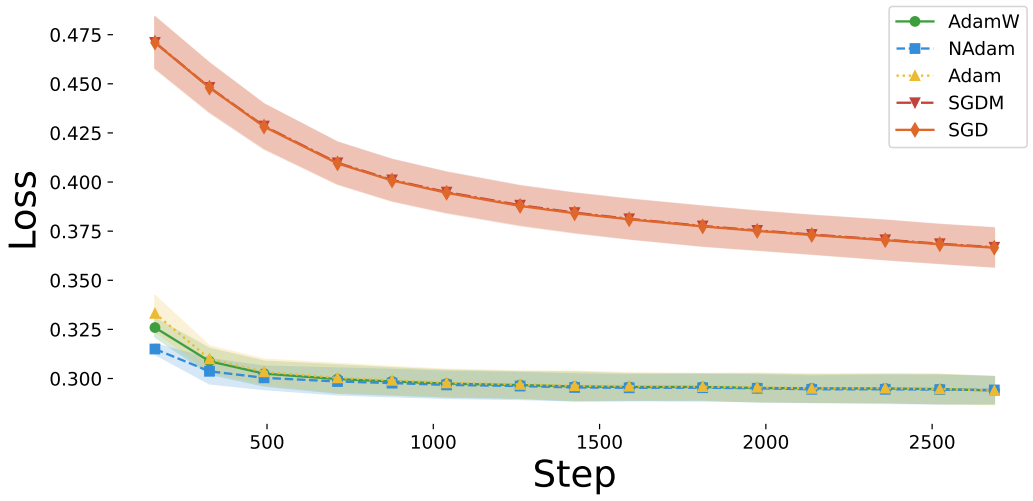

Fig. 5.3: Validation Loss over 5 epochs for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

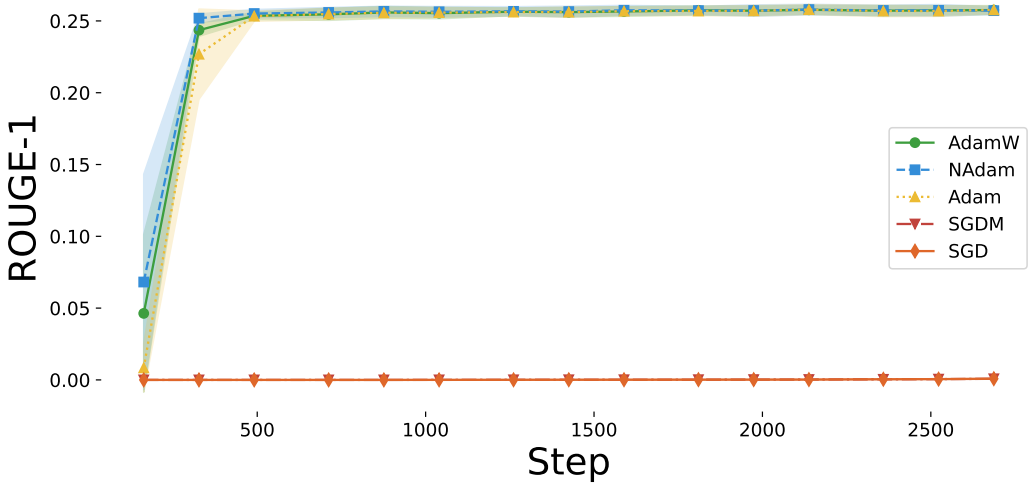

Fig. 5.4: Rouge 1 - F Measure, during validation, over 5 epochs for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

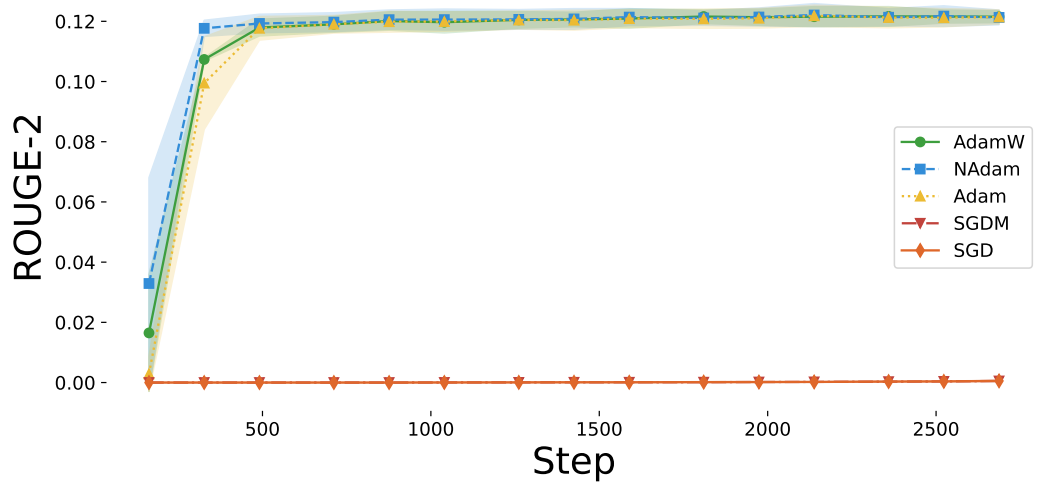

Fig. 5.5: Rouge 2 - F Measure, during validation, over 5 epochs for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

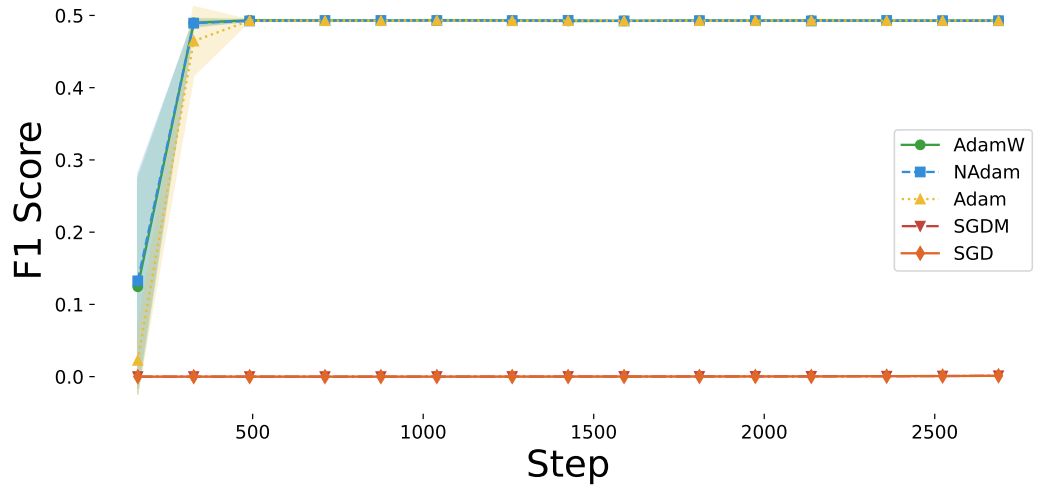

Fig. 5.6: F1 BERTScore, during validation, over 5 epochs for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

Full Mode

In the Full Mode, where additional hyperparameters were tuned, the adaptive optimizers NAdam, AdamW, and Adam showed even more similar performance than in the Basic Mode, both in terms of training loss and validation loss (see Figures 5.7 and 5.8). Interestingly, the performance of the SGDM optimizer improved compared to the Basic Mode, achieving a training loss of 0.5 and a validation loss of 0.4 in nearly half the steps. Notably, there is a distinction between SGD and SGDM in this setting, as SGDM reduces both the training and validation loss more quickly, making it the faster of the two. Although we retrained the SGD optimizer, it was only tuned for the learning rate hyperparameter and was included here solely for comparison purposes.

When analyzing the ROUGE-1, ROUGE-2 (F-measure), and BERTScore F1 metrics (see Figures 5.9, 5.10, and 5.11), we observe that all adaptive optimizers converge to similar performance within approximately the same number of steps as in the Basic Mode. In contrast, using the SGD optimizer we fail to report any meaningful values, likely due to poor performance or insignificant metric values. However, using SGDM optimizer we report significant results around the $1.3K$ step, showing improved performance compared to the Basic Mode, though it still lags far behind the adaptive optimizers.

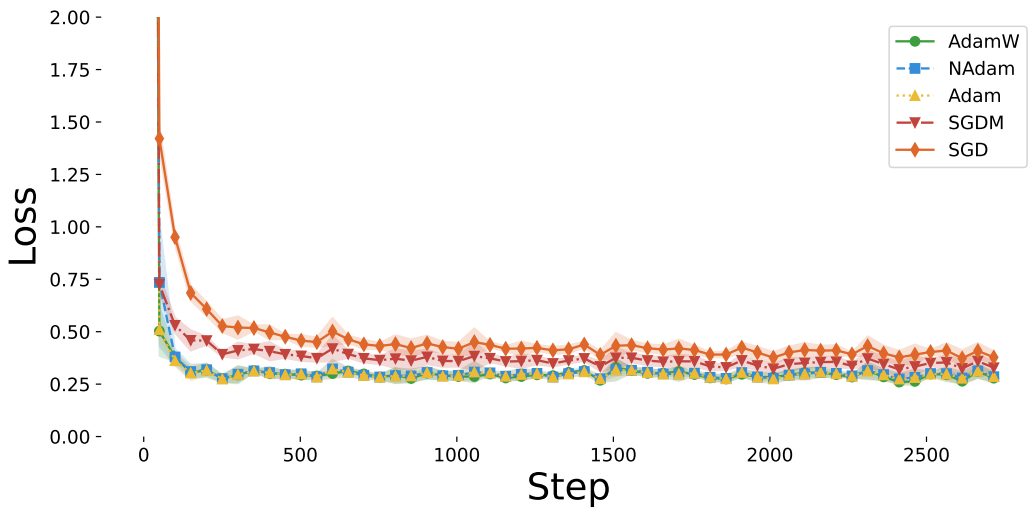

Fig. 5.7: Training Loss over 5 epochs for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

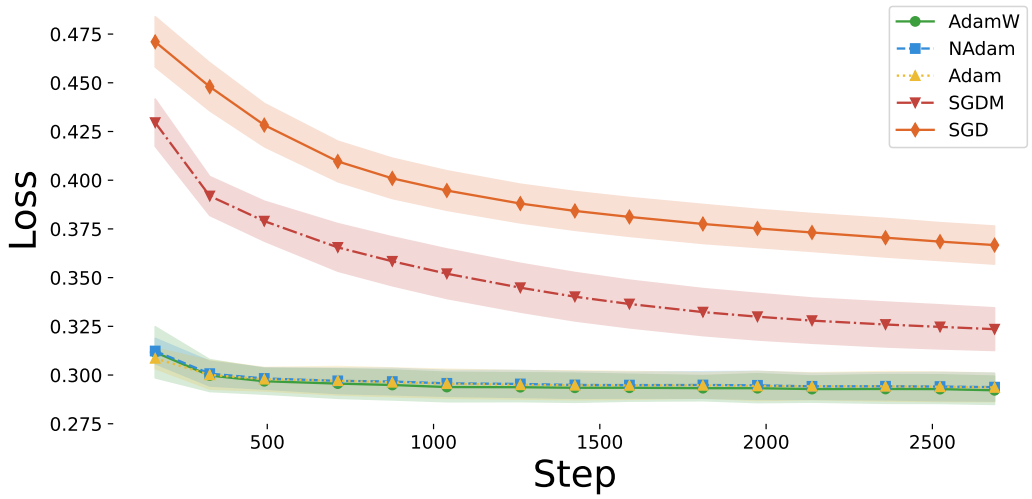

Fig. 5.8: Validation Loss score, over 5 epochs for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

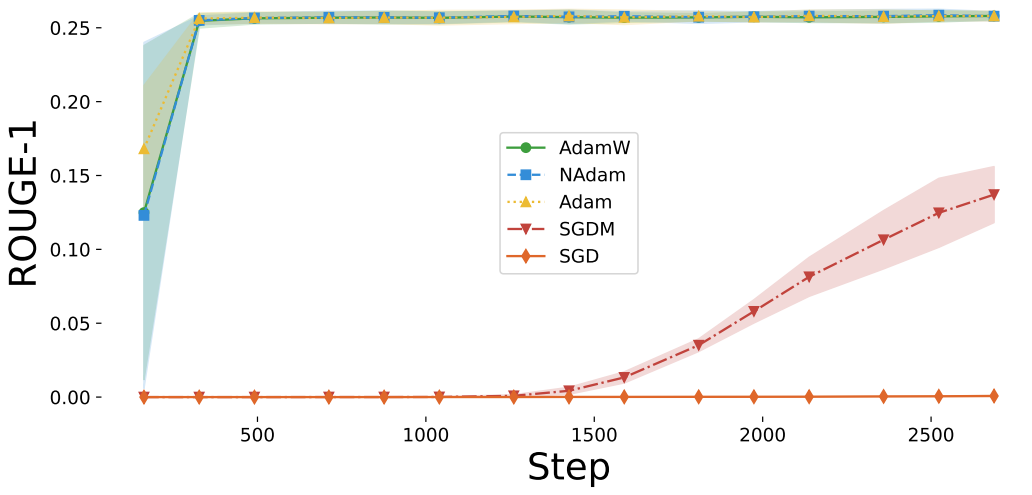

Fig. 5.9: Rouge 1 - F Measure score, during validation, over 5 epochs for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

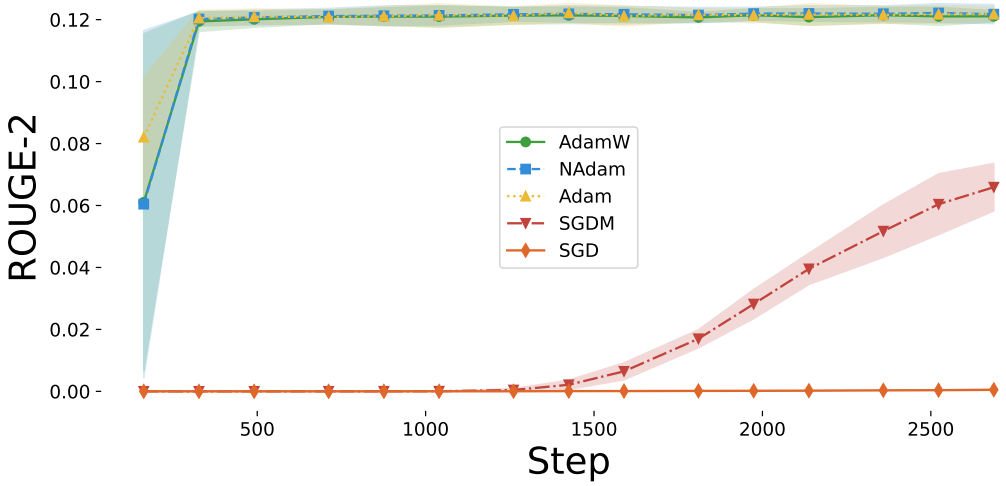

Fig. 5.10: Rouge 2 - F Measure score, during validation, over 5 epochs for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

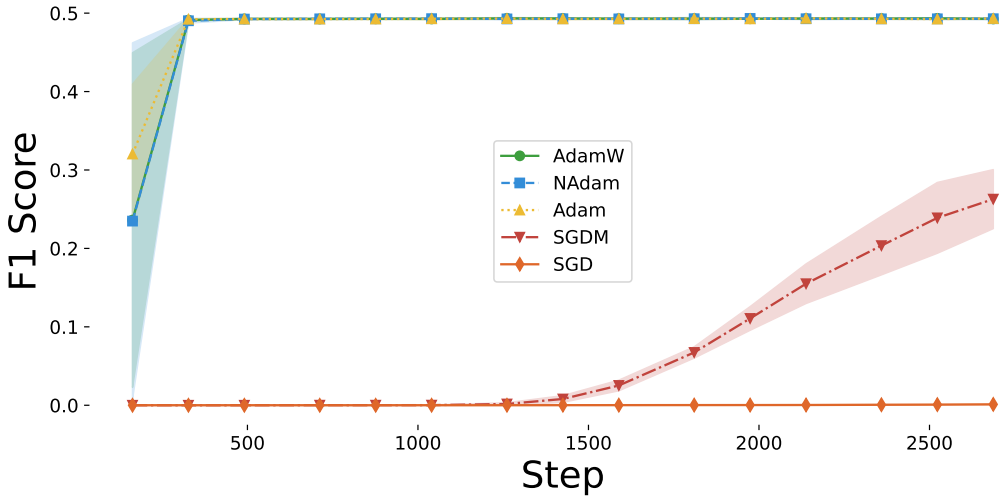

Fig. 5.11: F1 BERTScore score, during validation, over 5 epochs for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

5.8.2 SAMSum

Basic Mode

As observed in the Basic Mode using the SAMSum dataset, the adaptive optimizers NAdam, AdamW, and Adam exhibited similar performance in both training loss and validation loss (see Figures 5.12 and 5.13). In contrast, the performance of the SGD and SGDM optimizers, while comparable to each other, lagged behind the adaptive optimizers in terms of both training and validation loss.

When examining the ROUGE-1, ROUGE-2 (F-measure), and BERTScore F1 metrics (see Figures 5.14, 5.15, and 5.16), NAdam demonstrates the fastest initial convergence. However, while NAdam reaches optimal performance slightly ahead of the other adaptive optimizers, all ultimately converge to similar performance levels. As for the SGD and SGDM optimizers, their metrics are either not displayed or appear as zero in the graphs, owing to their extremely slow progress or negligible values reported during this phase.

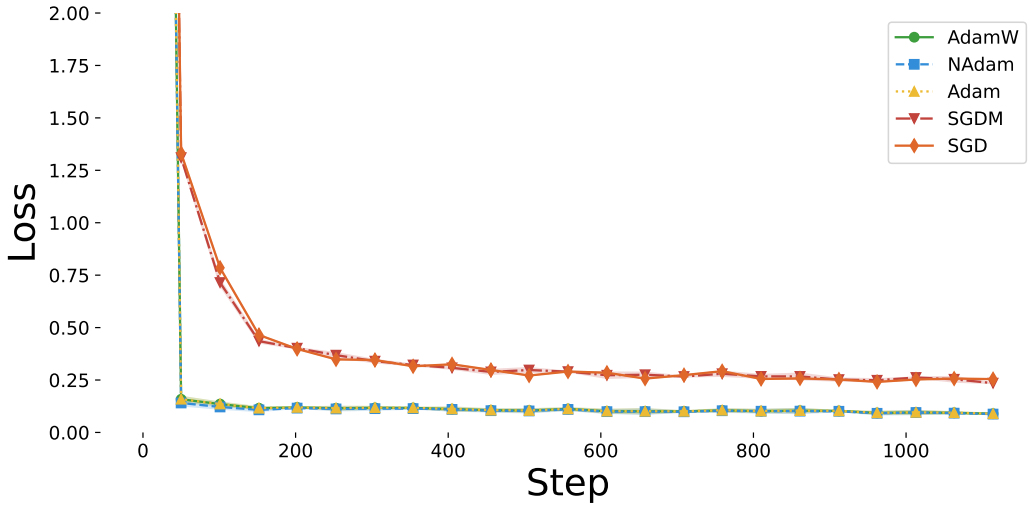

Fig. 5.12: Training Loss over 5 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

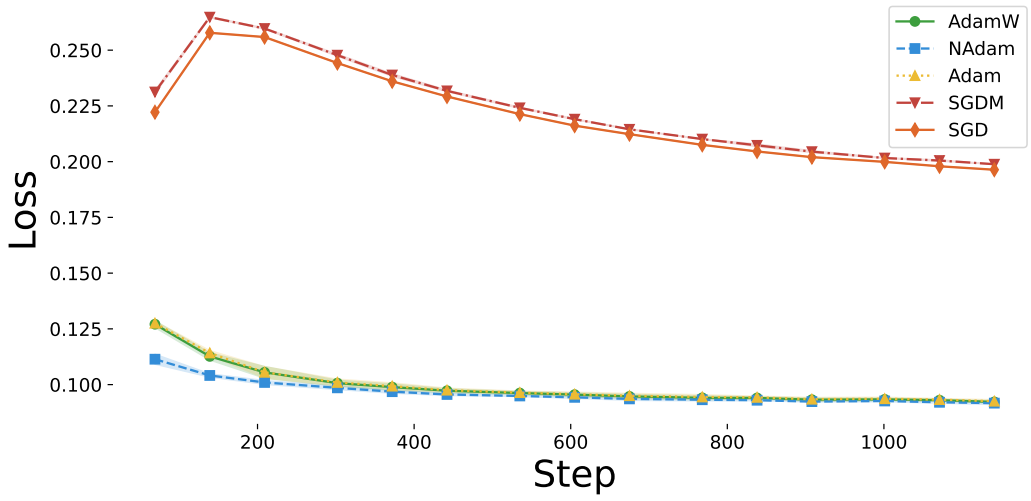

Fig. 5.13: Validation Loss over 5 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

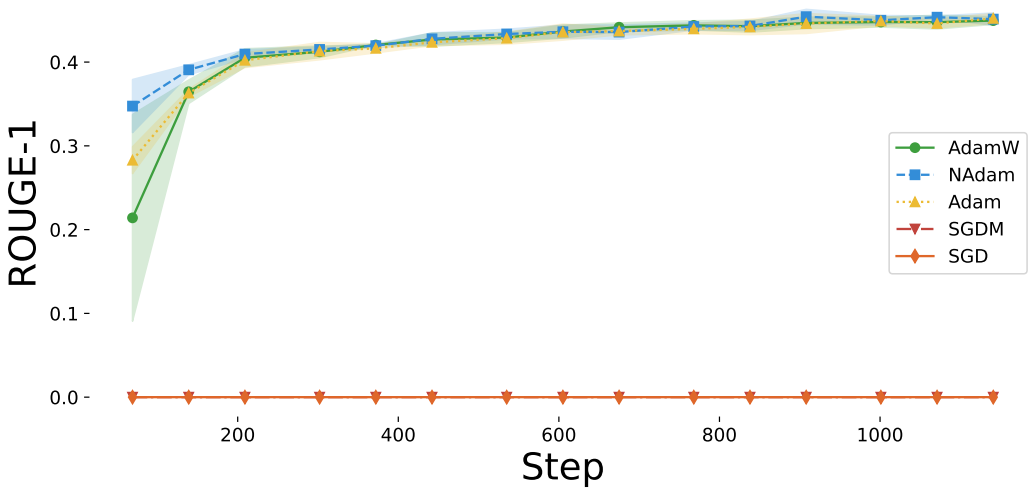

Fig. 5.14: Rouge 1 - F Measure score, during validation, over 5 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

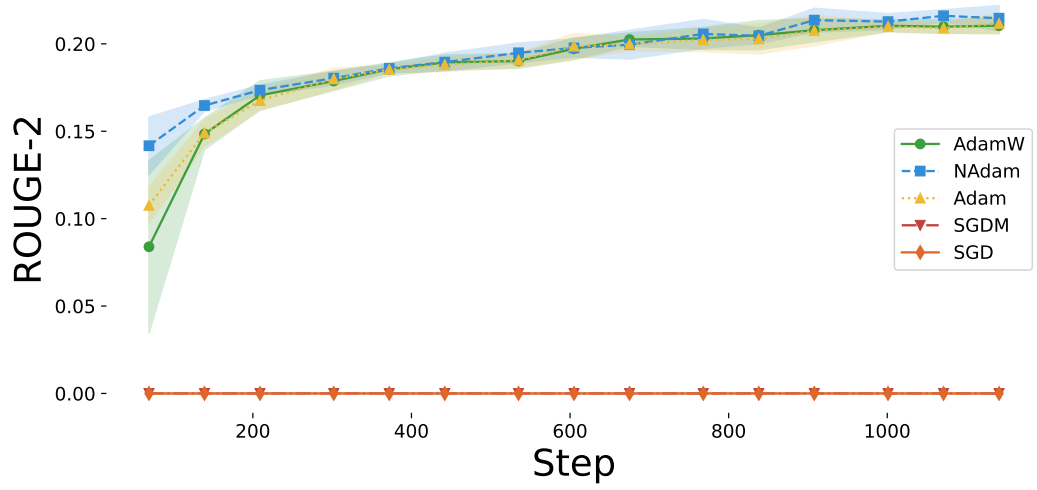

Fig. 5.15: Rouge 2 - F Measure score, during validation, over 5 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

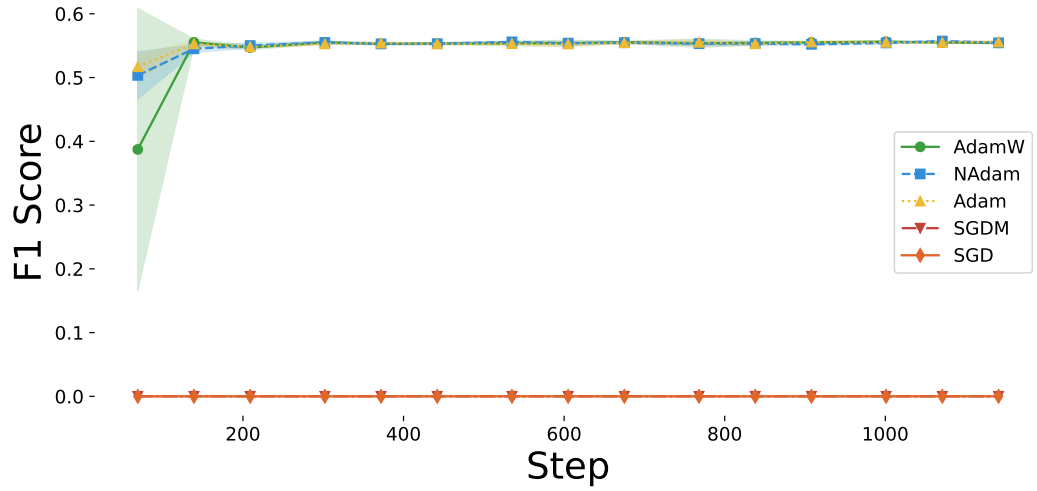

Fig. 5.16: F1 BERTScore score, during validation, over 5 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

Full Mode

In the Full Mode, where additional hyperparameters were tuned, the adaptive optimizers NAdam, AdamW, and Adam exhibited even more similar performance than in the Basic Mode, both in terms of training loss and validation loss (see Figures 5.17 and 5.18). While SGDM showed a notable improvement compared to the Basic Mode—reaching a training loss of 0.5 and a validation loss of 0.2 in nearly half the steps, around the 400 step mark—SGD's performance remained similar to that of the Basic Mode, as it was only retrained with learning rate adjustments. SGD, again, was included here solely for comparison purposes.

When analyzing the ROUGE-1, ROUGE-2 (F-measures), and BERTScore F1 metrics (see Figures 5.19, 5.20, and 5.21), we observe that all adaptive optimizers converge to similar performance within approximately the same number of steps as in the Basic Mode. NAdam shows a slight advantage in convergence speed compared to the other optimizers, but this difference is not significant. In contrast, the SGD and SGDM optimizers fail to report any meaningful values, likely due to poor performance or insignificant metric values.

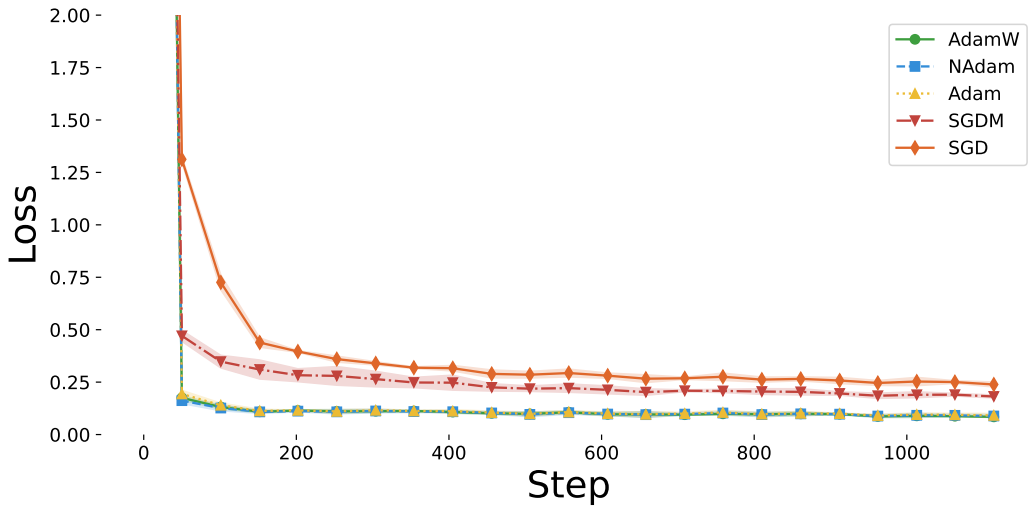

Fig. 5.17: Training Loss score over 5 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

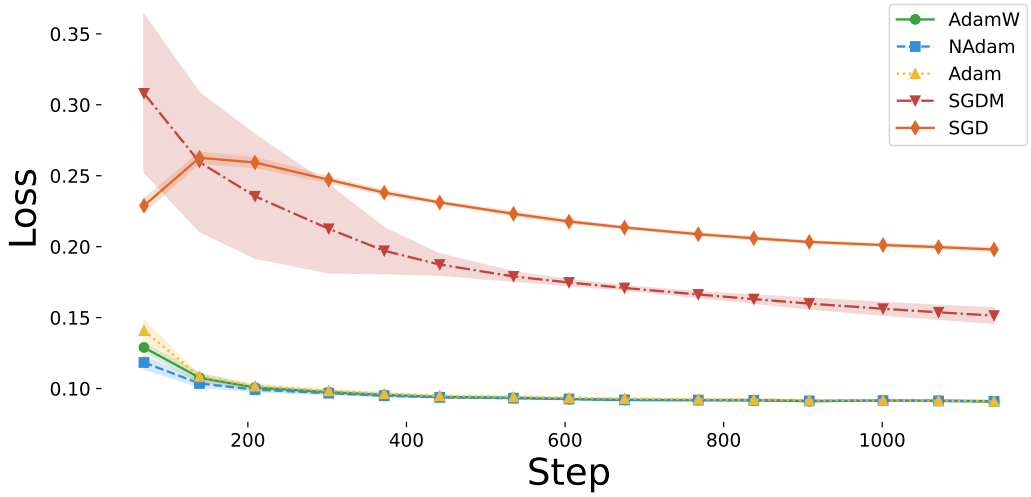

Fig. 5.18: Validation Loss score over 5 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

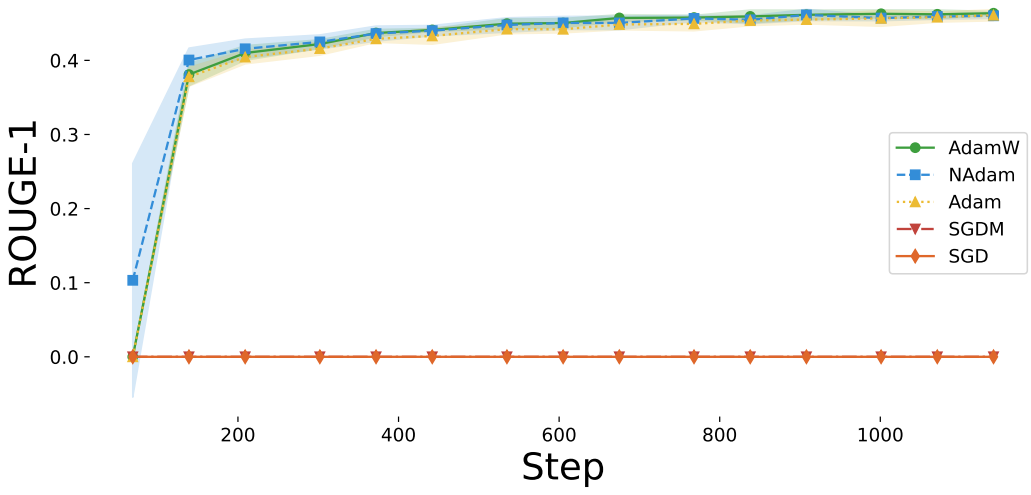

Fig. 5.19: Rouge 1 - F Measure score, during validation, over 5 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

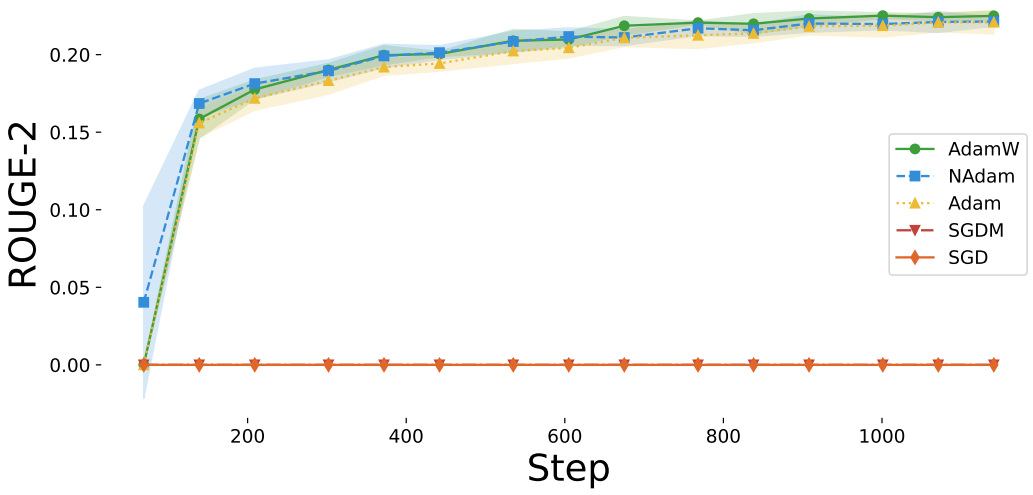

Fig. 5.20: Rouge 2 - F Measure score, during validation, over 5 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

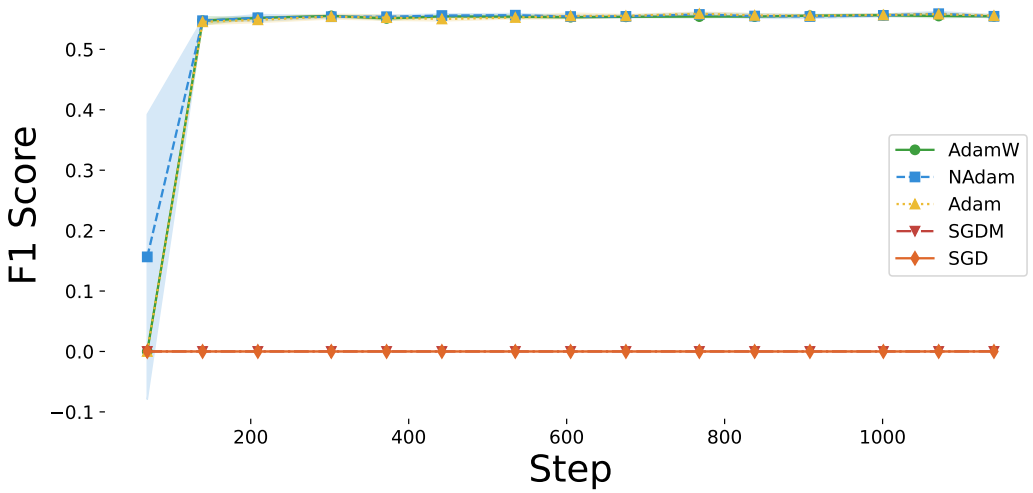

Fig. 5.21: F1 BERTScore, during validation, over 5 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

5.8.3 XSum

Basic Mode

As observed in the training mode where only the learning rate was tuned, using the XSum dataset, the adaptive optimizers NAdam, AdamW, and Adam exhibited similar performance in both training loss and validation loss (see Figures 5.22 and 5.23). In contrast, the performance of the SGD and SGDM optimizers, while similar, lagged behind the adaptive optimizers in terms of both training and validation loss.

When examining the ROUGE-1, ROUGE-2 (F-measure), and BERTScore F1 metrics (see Figures 5.24, 5.25, and 5.26), all the adaptive optimizers have the same performance, with a slight bump of Adam in step 1,200 for unknown reasons), while the non-adaptive optimizers are not reporting any values, thus they do not appear on the graphs.

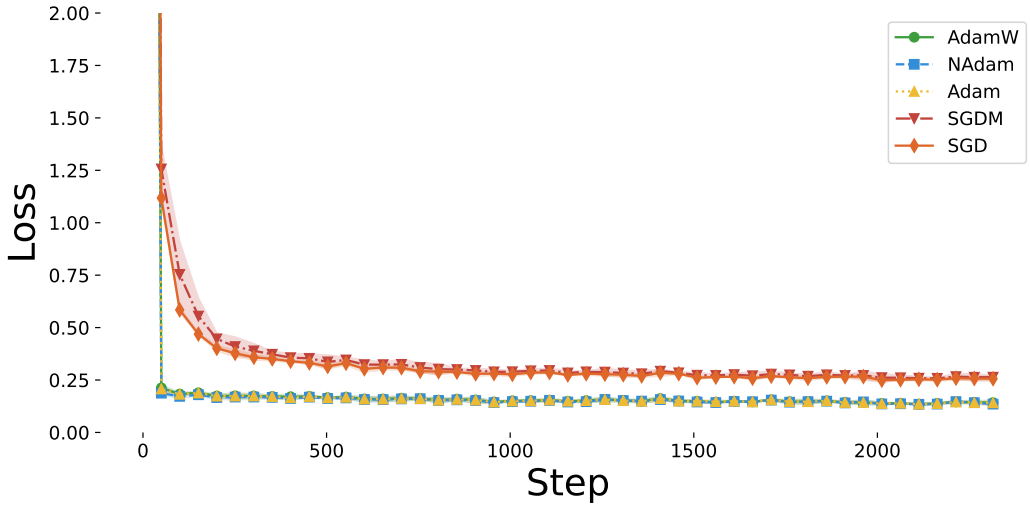

Fig. 5.22: Training Loss over 5 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

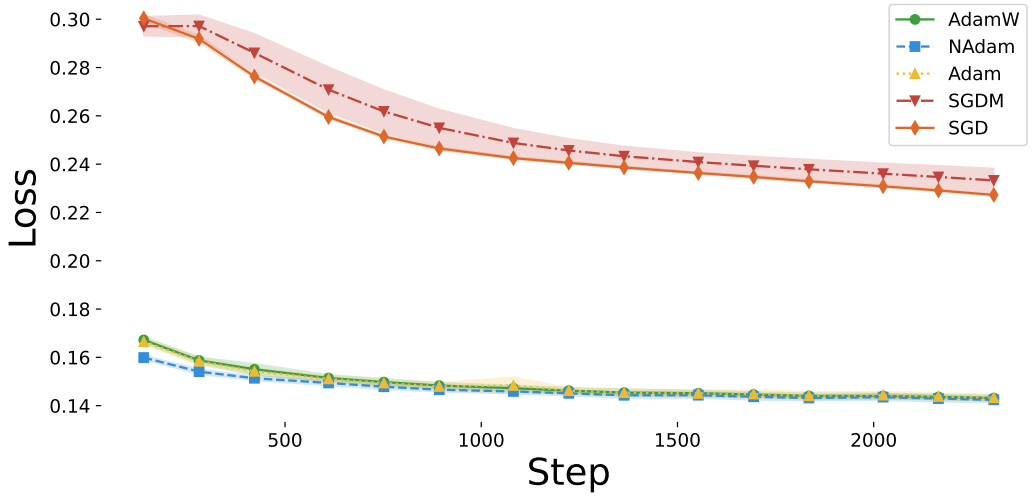

Fig. 5.23: Validation Loss over 5 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

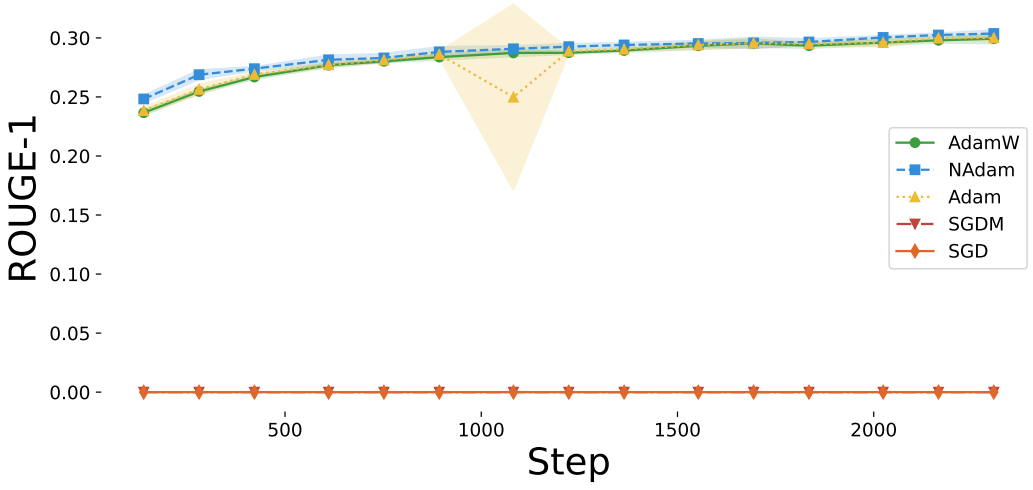

Fig. 5.24: Rouge 1 - F Measure, during validation, over 5 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

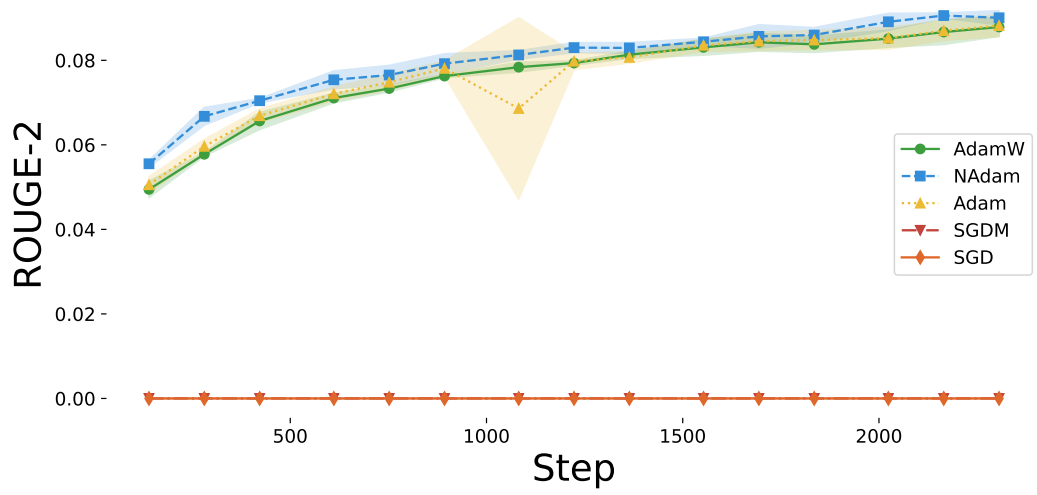

Fig. 5.25: Rouge 2 - F Measure, during validation, over 5 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

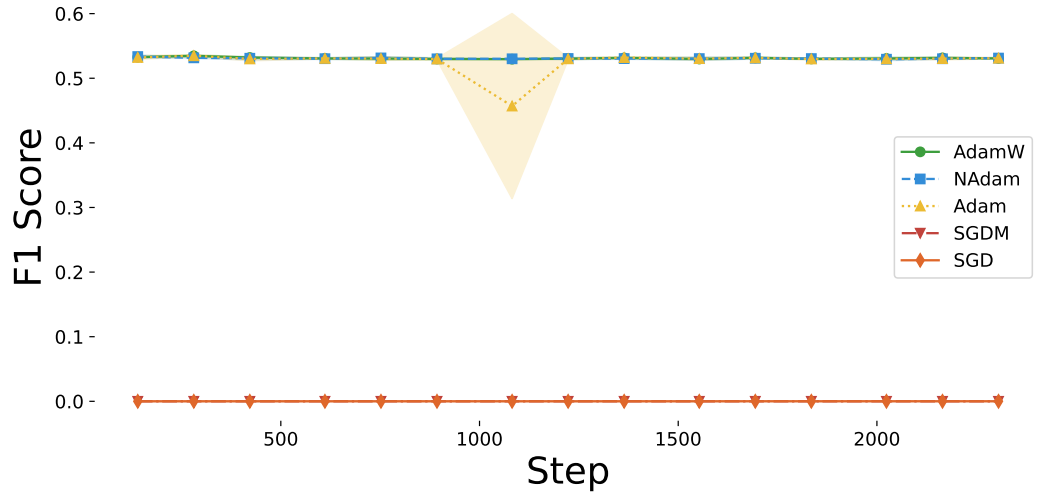

Fig. 5.26: F1 BERTScore, during validation, over 5 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

Full Mode

In the Full Mode, where additional hyperparameters were tuned, the adaptive optimizers NAdam, AdamW, and Adam exhibited even more similar performance than in the Basic Mode, both in terms of training loss and validation loss (see Figures 5.27 and 5.28), making no distinction in performance among them. While SGDM showed a notable improvement compared to the Basic Mode—achieving a training loss of 0.5 and a validation loss of 0.25 in nearly half the steps, reaching this milestone around the 200 step mark—SGD's performance remained similar to that of the Basic Mode, as it was only retrained with learning rate adjustments. SGD, again, was included here solely for comparison purposes.

When analyzing the ROUGE-1, ROUGE-2 (F-measure), and BERTScore F1 metrics (see Figures 5.29, 5.30, and 5.31), we observe that all adaptive optimizers converge to similar performance within approximately the same number of steps as in the Basic Mode. In contrast, the SGD and SGDM optimizers fail to report any meaningful values, likely due to poor performance or insignificant metric values.

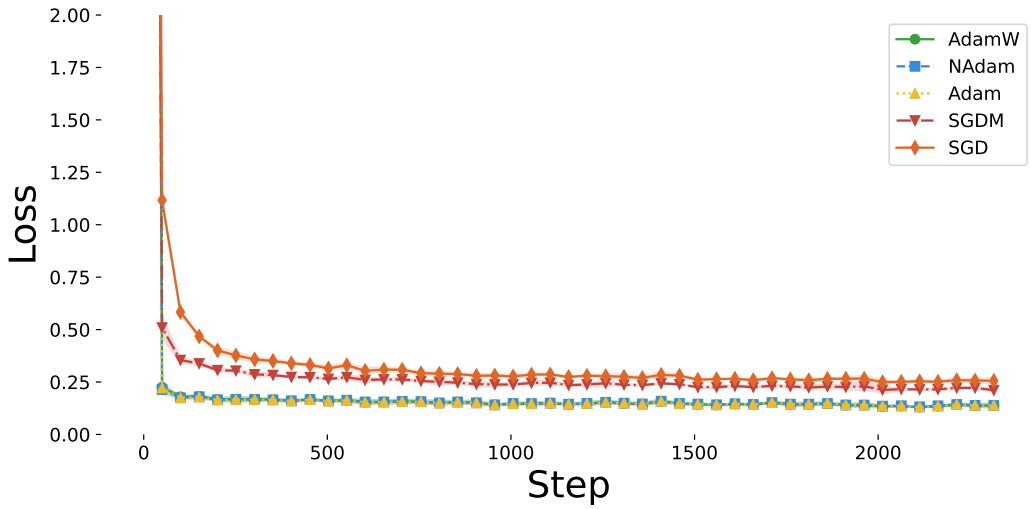

Fig. 5.27: Training Loss over 5 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

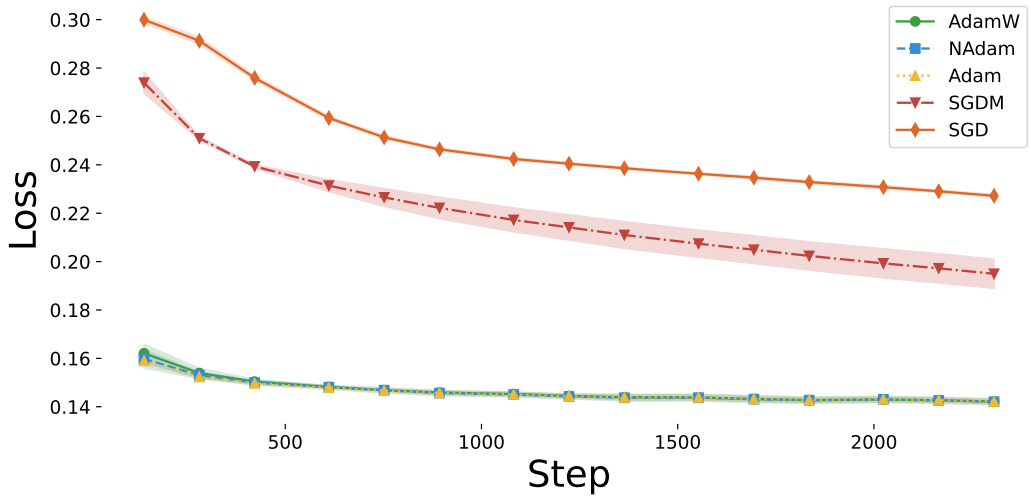

Fig. 5.28: Validation Loss over 5 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

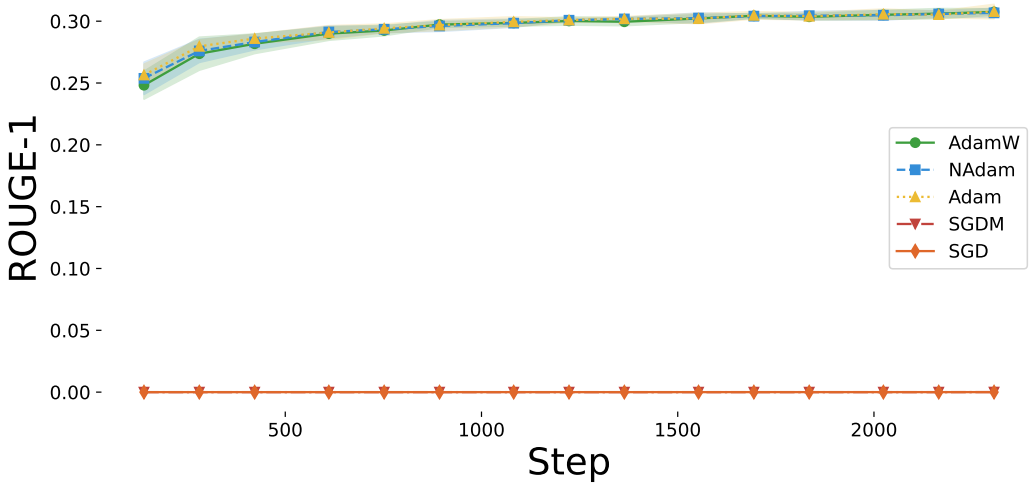

Fig. 5.29: Rouge 1 - F Measure score, during validation, over 5 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

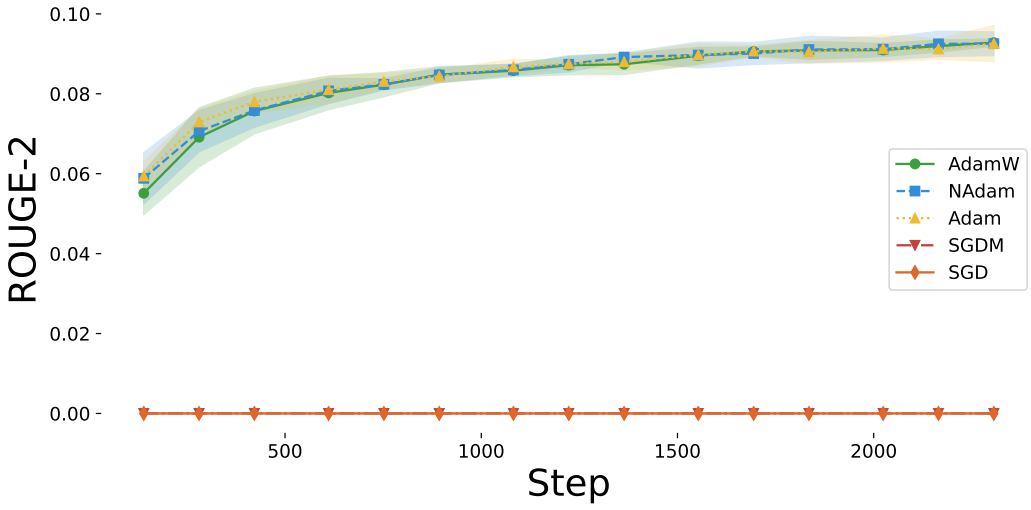

Fig. 5.30: Rouge 2 - F Measure score, during validation, over 5 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

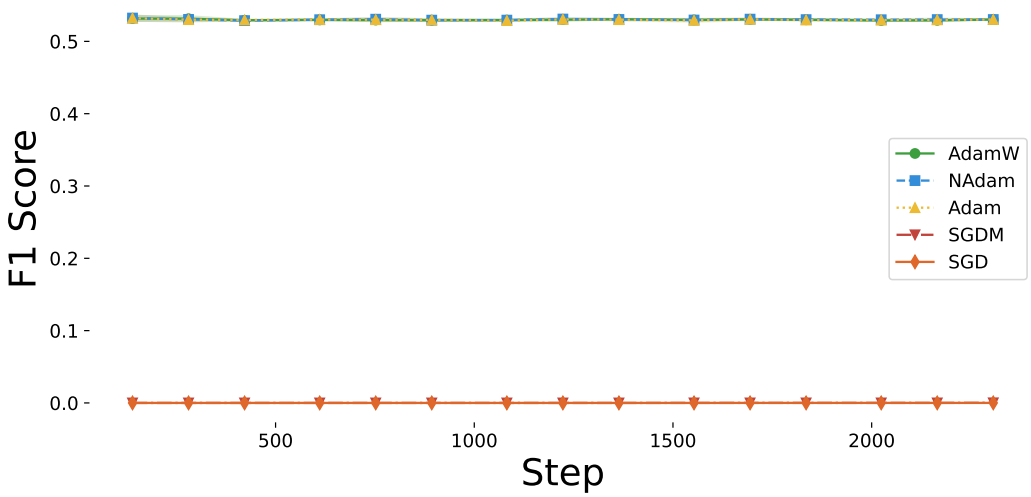

Fig. 5.31: F1 BERTScore score, during validation, over 5 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

5.8.4 IWSLT Dataset

Basic Mode

When it comes to the IWSLT dataset and in the training mode where only the learning rate was tuned (Basic Mode), the adaptive optimizers NAdam, AdamW, and Adam exhibited similar performance in both training, validation loss and F1 score (see Figures 5.32, 5.33 and 5.34). In contrast, the performance of the SGD and SGDM optimizers, while similar to each other, lagged behind the adaptive optimizers in terms of both training and validation loss, in addition, at F1 score, they are not reporting any values.

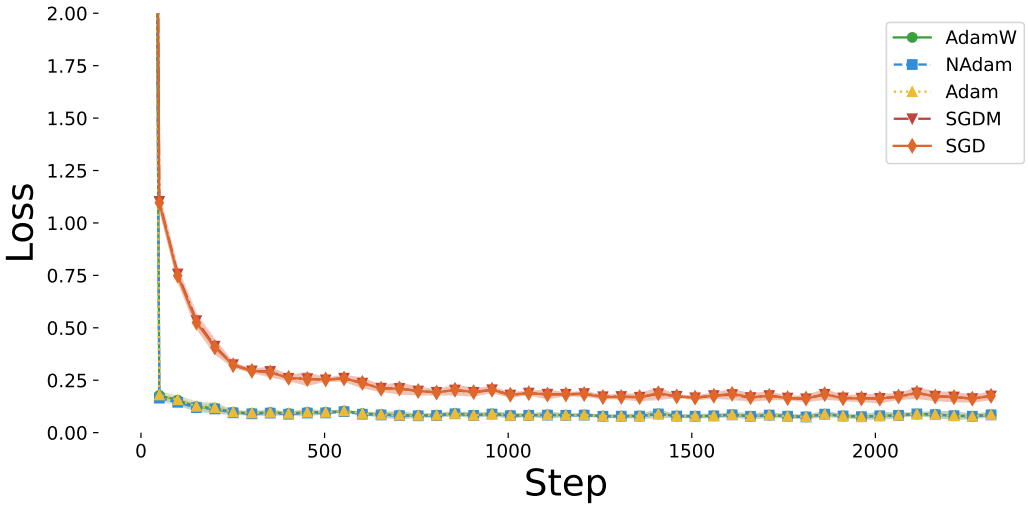

Fig. 5.32: Training Loss over 5 epochs for the IWSLT dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

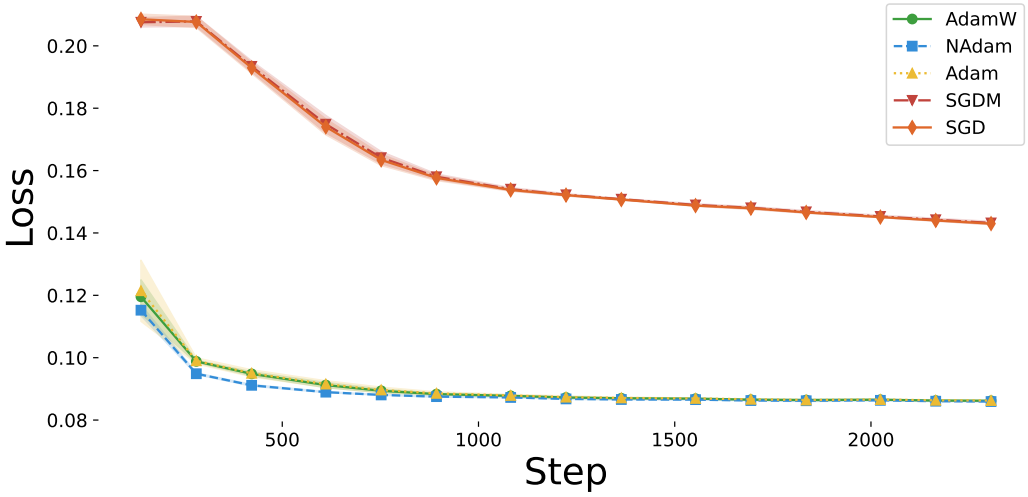

Fig. 5.33: Validation Loss over 5 epochs for the IWSLT dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

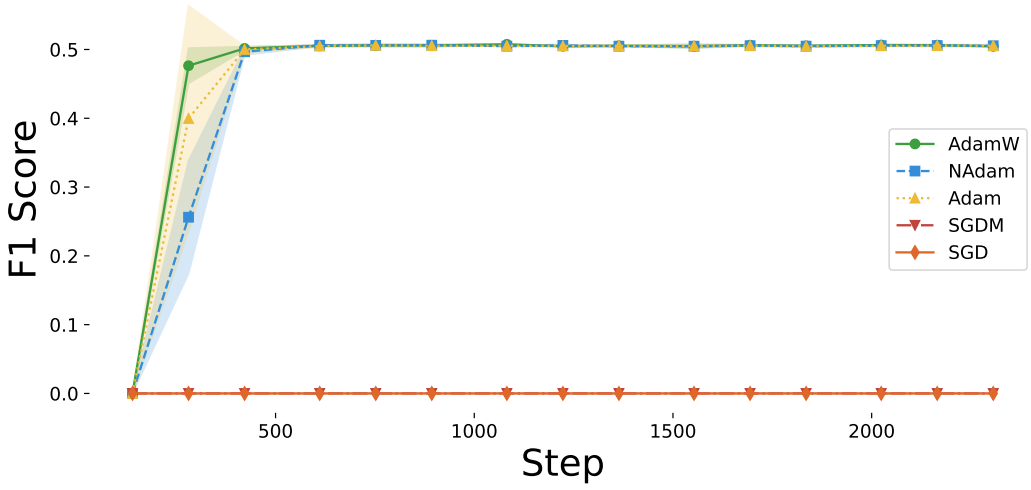

Fig. 5.34: F1 BERTScore, during validation, over 5 epochs for the IWSLT dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

Full Mode

In the Full Mode, the adaptive optimizers NAdam, AdamW, and Adam showed once again a similar performance between them, both in terms of training loss and validation loss (see Figures 5.35 and 5.36).

When analyzing the BERTScore F1 metrics (see Figure 5.37), we observe that all adaptive optimizers converge to similar performance within approximately the same number of steps as in the Basic Mode. Additionally, SGD is not reporting any values, and SGDM is reporting values only in the step 2300, which is near the end of the training process.

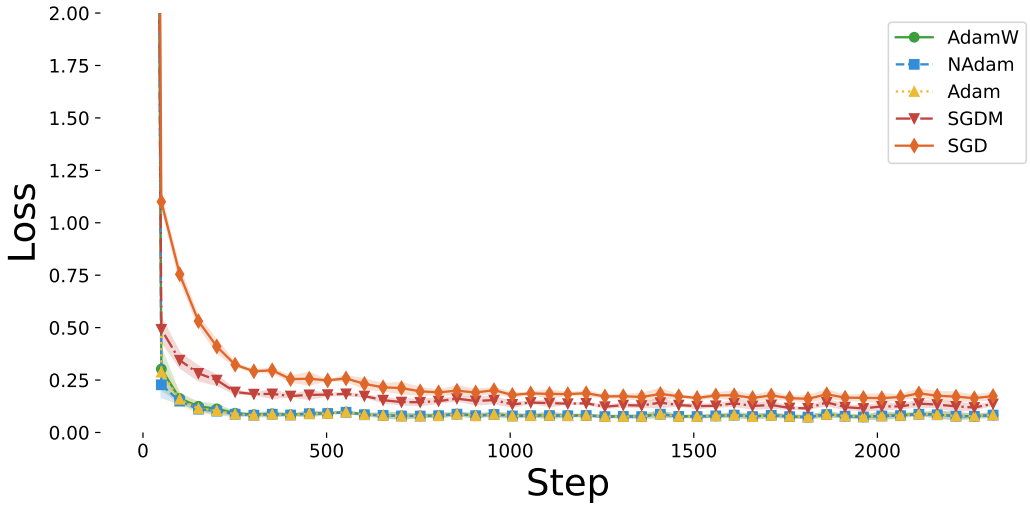

Fig. 5.35: Training Loss over 5 epochs for the IWSLT dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

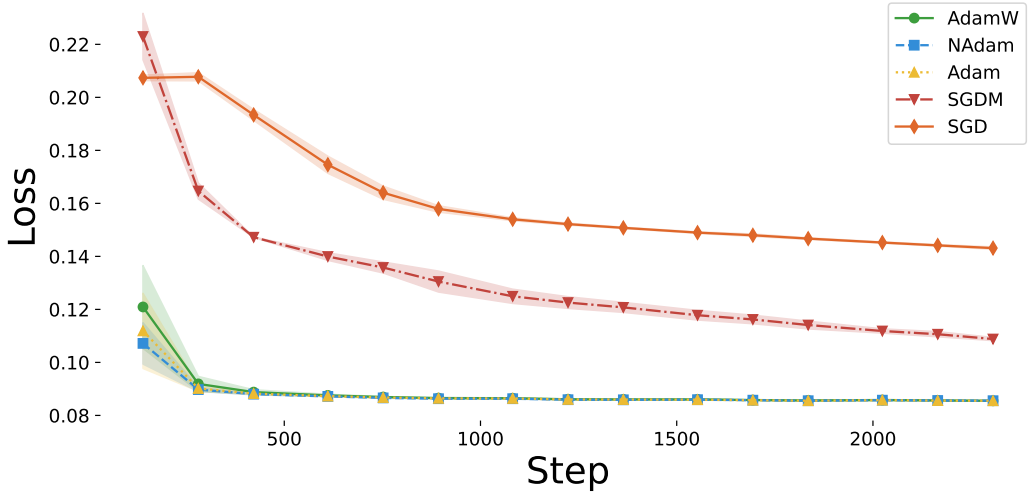

Fig. 5.36: Validation Loss score, over 5 epochs for the IWSLT dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

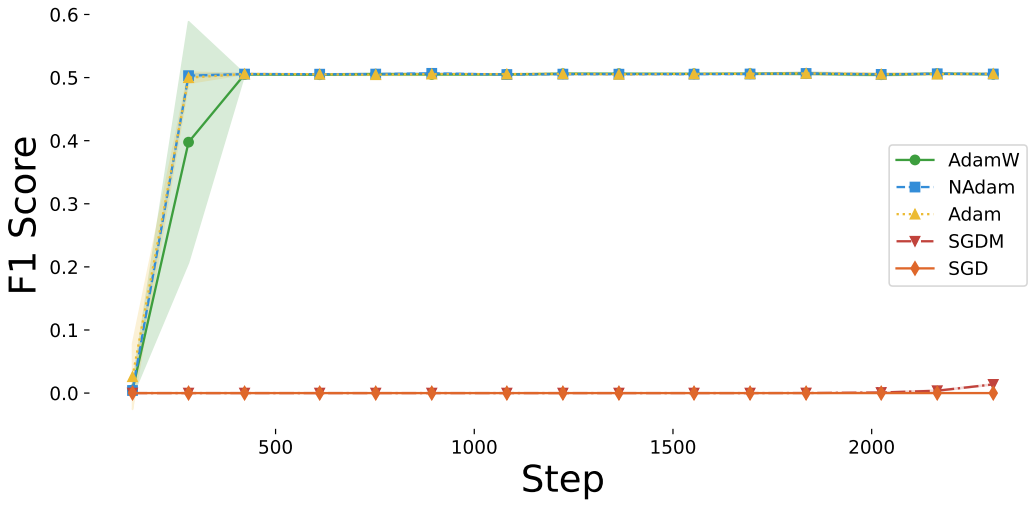

Fig. 5.37: F1 BERTScore score, during validation, over 5 epochs for the IWSLT dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

5.8.5 Flores Dataset

Basic Mode

As observed in the Basic Mode using the Flores dataset, the adaptive optimizers NAdam, AdamW, and Adam exhibited similar performance in both training loss and validation loss (see Figures 5.38 and 5.39). In general, the performance between SGD and SGDM, lack behind the adaptive optimizers, and interestingly even though SGDM is lacking behind of SGD in the start of the training process with the training and validation loss metrics, SGDM gets ahead of SGD in the BERTScore f1 metric. This phenomenon only occurs in this dataset.''

When it comes to the F1 (BERTScore) (see Figure 5.40) we can see that while adaptive optimizers behaving with the same overall performance, SGD and SGDM, have a decreasing rate of performance while training.

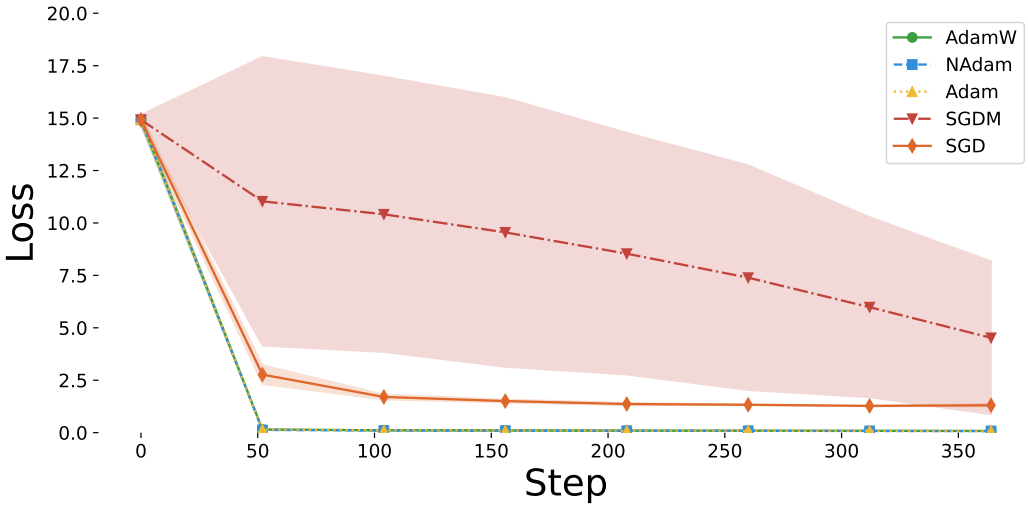

Fig. 5.38: Training Loss over 5 epochs for the Flores dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

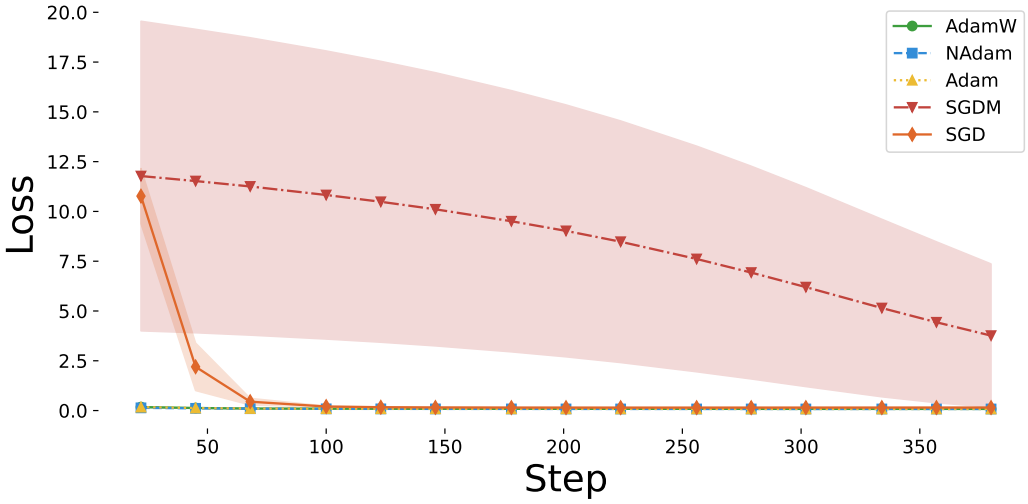

Fig. 5.39: Validation Loss over 5 epochs for the Flores dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

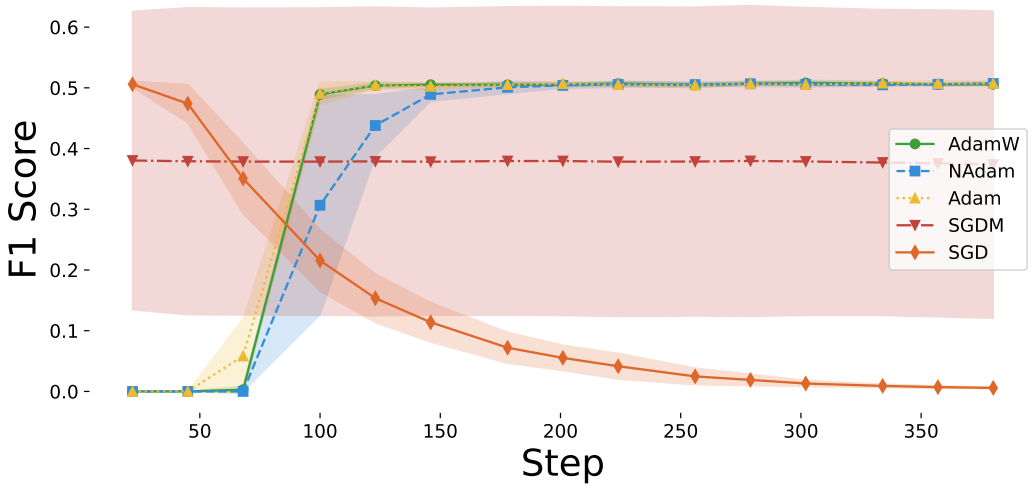

Fig. 5.40: F1 BERTScore score, during validation, over 5 epochs for the Flores dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

Full Mode

In the Full Mode, where additional hyperparameters were tuned, the adaptive optimizers NAdam, AdamW, and Adam exhibited even more similar performance than in the Basic Mode, both in terms of training loss and validation loss (see Figures 5.41 and 5.42).

When analyzing BERTScore F1 metric (see Figure 5.43), we observe that all adaptive optimizers converge to similar performance within approximately the same number of steps as in the Basic Mode. In contrast, the SGD and SGDM optimizers continuing to have a decreasing rate of performance during training, showcasing their lack to compete against the adaptive optimizers.

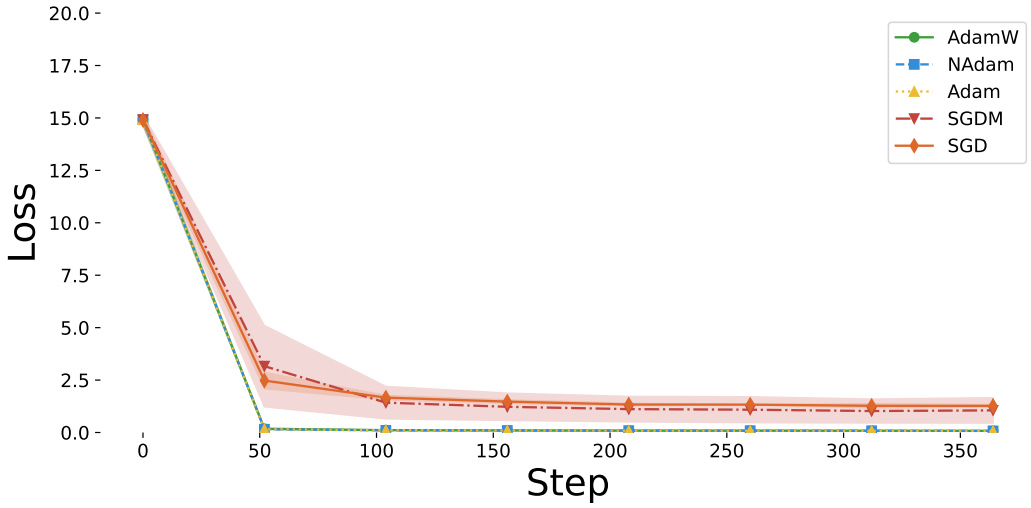

Fig. 5.41: Training Loss score over 5 epochs for the Flores dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

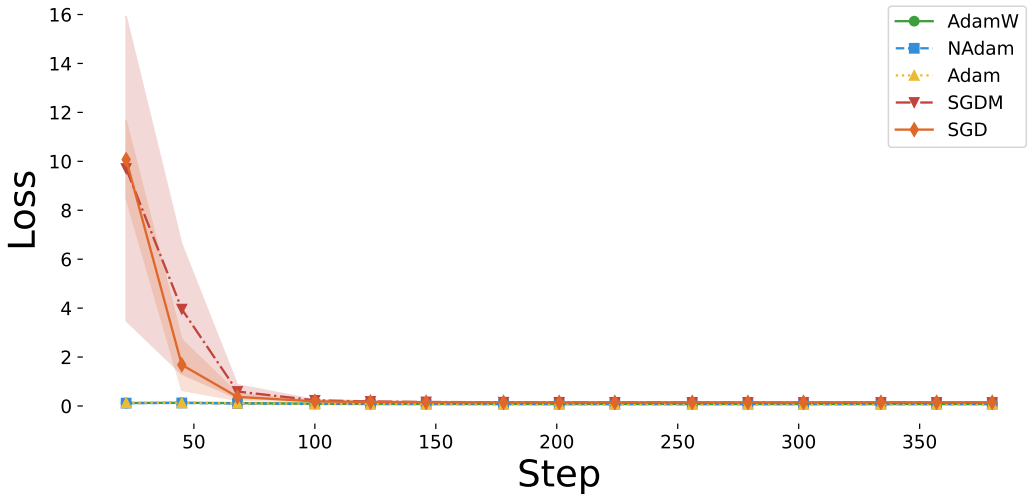

Fig. 5.42: Validation Loss score over 5 epochs for the Flores dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

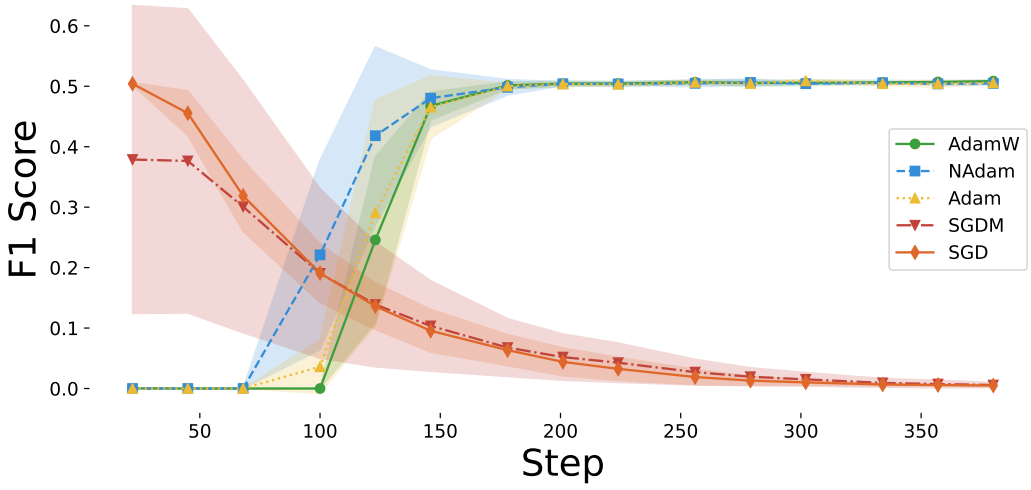

Fig. 5.43: F1 BERTScore, during validation, over 5 epochs for the Flores dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

5.9 Test Results and Discussion

In this section, we present the performance of each optimizer on a test subset that has never been seen by the models. The evaluation metrics include Test Cross Entropy Loss (Test Loss), BERTScore F1, ROUGE-1, and ROUGE-2 (F-measure). Each model was tested using a checkpoint taken during training, selected based on the lowest validation loss, and then evaluated on a completely unseen test subset of the respective datasets. We begin by reporting the results for the CNN/DailyMail dataset.

5.9.1 CNN/Dailymail Test Results

Basic Mode

As shown in Tables 5.18 and 5.19, the adaptive optimizers NAdam, AdamW, and Adam exhibit similar performance across all metrics, with differences appearing only at the fourth decimal place. In contrast, the SGDM and SGD optimizers demonstrate similar performance to each other across all metrics but fall behind the adaptive optimizers in overall performance.

Tab. 5.18: This table presents the mean values of each optimizer's performance, averaged over four different seeds, for the CNN/Dailymail dataset in learning rate tuning mode only. The table contains the results for each optimizer's learning rate. Due to space constraints, this is Part 1 of 2.

Metrics	NAdam	SGD	SGDM
Test Loss	0.2936 ($\pm$0.0011)	0.3663 ($\pm$0.0017)	0.3665 ($\pm$0.0015)
Test F1 (BERTScore)	0.4926 ($\pm$0.0006)	0.0017 ($\pm$0.0004)	0.0016 ($\pm$0.0007)
Test Rouge 1 (F Measure)	0.2581 ($\pm$0.0024)	0.0011 ($\pm$0.0002)	0.001 ($\pm$0.0004)
Test Rouge 2 (F Measure)	0.1222 ($\pm$0.0022)	0.0006 ($\pm$0.0001)	0.0006 ($\pm$0.0002)

Tab. 5.19: This table continues from Part 1, showing the mean values of each optimizer's performance, averaged over four seeds, for the CNN/Dailymail dataset in learning rate tuning mode only. This table captures the remaining optimizers.

Metrics	AdamW	Adam
Test Loss	0.2932 ($\pm$0.0013)	0.2934 ($\pm$0.0013)
Test F1 (BERTScore)	0.4925 ($\pm$0.0004)	0.493 ($\pm$0.0005)
Test Rouge 1 (F Measure)	0.2572 ($\pm$0.0026)	0.2575 ($\pm$0.0021)
Test Rouge 2 (F Measure)	0.1211 ($\pm$0.0011)	0.1216 ($\pm$0.0014)

Full Mode

In the Full Mode, with more hyperparameters tuned for each optimizer, (see Tables 5.20 and 5.21) we observe an improvement in the performance of all optimizers. However, the key observation is that the improvement in the adaptive optimizers (NAdam, AdamW, and Adam) is only in the third or fourth decimal place, indicating that while there is some improvement, it is minimal. In contrast, the SGDM optimizer shows a significant performance boost, with its BERTScore F1 increasing from 0.0016 in the Basic Mode to 0.2691 in the Full Mode. Improvements are also observed in the other metrics for SGDM, whereas SGD's performance remains largely unchanged.

Tab. 5.20: This table presents the mean values of each optimizer's performance, averaged over four different seeds, for the CNN/Dailymail dataset in Full Mode. The table includes results for each optimizer's hyperparameters and learning rates. Due to space constraints, this is Part 1 of 2.

Metrics	NAdam	SGD	SGDM
Test Loss	0.2932 (±0.0012)	0.3664 (±0.0014)	0.3233 (±0.0042)
Test F1 (BERTScore)	0.4931 (±0.0007)	0.0015 (±0.0008)	0.2691 (±0.037)
Test Rouge 1 (F Measure)	0.2586 (±0.0025)	0.0009 (±0.0005)	0.1406 (±0.018)
Test Rouge 2 (F Measure)	0.1225 (±0.0002)	0.0005 (±0.0003)	0.0679 (±0.0067)

Tab. 5.21: This table continues from Part 1, showing the mean values of each optimizer's performance, averaged over four seeds, for the CNN/Dailymail dataset in Full Mode. This table captures additional hyperparameters and optimizers.

Metrics	AdamW	Adam
Test Loss	0.2928 (±0.0016)	0.2931 (±0.0013)
Test F1 (BERTScore)	0.4919 (±0.007)	0.4925 (±0.001)
Test Rouge 1 (F Measure)	0.2587 (±0.0027)	0.2587 (±0.0026)
Test Rouge 2 (F Measure)	0.1225 (±0.0019)	0.1226 (±0.0017)

5.9.2 SAMSum

Basic Mode

In the SAMSum dataset, using the Basic Mode, we observe that using the adaptive optimizers NAdam, AdamW, and Adam we reported very similar results across all metrics (see Tables 5.22 and 5.23). On the other hand, the SGD and SGDM optimizers reported zero values for most metrics, with the exception of Test Loss, where SGDM showed a slight advantage over SGD. However, both still lag significantly behind the performance of the adaptive optimizers.

Tab. 5.22: This table presents the mean values of each optimizer's performance, averaged over four different seeds, for the SAMSum dataset in learning rate tuning mode only. The table contains the results for each optimizer's learning rate. Due to space constraints, this is Part 1 of 2.

Metrics	NAdam	SGD	SGDM
Test Loss	0.1413 (±0.001)	0.2268 (±0.0013)	0.1947 (±0.0077)
Test F1 (BERTScore)	0.5305 (±0.0015)	0	0
Test Rouge 1 (F Measure)	0.3094 (±0.001)	0	0
Test Rouge 2 (F Measure)	0.0939 (±0.0004)	0	0

Tab. 5.23: This table continues from Part 1, showing the mean values of each optimizer's performance, averaged over four seeds, for the SAMSum dataset in learning rate tuning mode only. This table captures the remaining optimizers.

Metrics	AdamW	Adam
Test Loss	0.1414 (±0.0011)	0.1413 (±0.001)
Test F1 (BERTScore)	0.5302 (±0.0011)	0.5307 (±0.0013)
Test Rouge 1 (F Measure)	0.3089 (±0.0006)	0.3082 (±0.0016)
Test Rouge 2 (F Measure)	0.0945 (±0.0008)	0.0934 (±0.0015)

Full Mode

In the Full Mode, where additional hyperparameters were tuned for each optimizer, we observe an overall improvement in performance for both adaptive and non-adaptive optimizers. Specifically, there is a significant boost in ROUGE-1 and ROUGE-2 (F-measure) scores (see Tables 5.24 and 5.25). For instance, the ROUGE-1 score for NAdam increased from 0.3094 to 0.4527, representing a 46% improvement, with a similar increase observed in ROUGE-2. This trend is also evident for AdamW and Adam. However, when it comes to the BERTScore metric, we see an improvement but this performance boost is less than 1% (from 0.53 to 0.55 in general for all adaptive optimizers). With BERTScore as the metric to evaluate our model's contextual performance, the observed improvement of less than 1% is relatively minor.

In contrast, while the SGDM and SGD optimizers show improved Test Loss scores, they still report zero values for most other metrics. The exception is SGDM, which achieved a low but measurable F1 BERTScore of 0.00017.

Tab. 5.24: This table presents the mean values of each optimizer's performance, averaged over four different seeds, for the SAMSum dataset in Full Mode. The table includes results for each optimizer's hyperparameters and learning rates. Due to space constraints, this is Part 1 of 2.

Metrics	NAdam	SGD	SGDM
Test Loss	0.0943 (± 0.001)	0.2015 (± 0.0016)	0.1552 (± 0.0057)
Test F1 (BERTScore)	0.553 (± 0.001)	0	0.00017 (± 0.0003)
Test Rouge 1 (F Measure)	0.4527 (± 0.0065)	0	0
Test Rouge 2 (F Measure)	0.215 (± 0.0066)	0	0

Tab. 5.25: This table continues from Part 1, showing the mean values of each optimizer's performance, averaged over four seeds, for the SAMSum dataset in Full Mode. This table captures additional hyperparameters and optimizers.

Metrics	AdamW	Adam
Test Loss	0.0948 (± 0.0008)	0.0944 (± 0.0008)
Test F1 (BERTScore)	0.553 (± 0.0008)	0.5558 (± 0.0038)
Test Rouge 1 (F Measure)	0.4495 (± 0.0042)	0.4521 (± 0.0027)
Test Rouge 2 (F Measure)	0.2126 (± 0.0023)	0.2127 (± 0.0031)

5.9.3 XSum

Basic Mode

In the XSum dataset, using the Basic Mode, where only the learning rate was tuned for each optimizer, we observe similar performance across all test metrics for the adaptive optimizers NAdam, AdamW, and Adam. NAdam shows slightly better scores, though the differences are minor (see Tables 5.26 and 5.27). In contrast, the SGD and SGDM optimizers lag behind the adaptive optimizers, reporting only test loss values while showing zero values for the other metrics. SGDM performs slightly better than SGD in terms of test loss.

Tab. 5.26: This table presents the mean values of each optimizer's performance, averaged over four different seeds, for the XSum dataset in learning rate tuning mode only. The table contains the results for each optimizer's learning rate. Due to space constraints, this is Part 1 of 2.

Metrics	NAdam	SGD	SGDM
Test Loss	0.14228 (±0.0071)	0.2268 (±0.0013)	0.2329 (±0.0055)
Test F1 (BERTScore)	0.5309 (±0.0001)	0	0
Test Rouge 1 (F Measure)	0.3033 (±0.0023)	0	0
Test Rouge 2 (F Measure)	0.0905 (±0.0017)	0	0

Tab. 5.27: This table continues from Part 1, showing the mean values of each optimizer's performance, averaged over four seeds, for the XSum dataset in learning rate tuning mode only. This table captures the remaining optimizers.

Metrics	AdamW	Adam
Test Loss	0.1424 (±0.0009)	0.1424 (±0.0007)
Test F1 (BERTScore)	0.5312 (±0.00021)	0.5304 (±0.0014)
Test Rouge 1 (F Measure)	0.3007 (±0.0016)	0.3006 (±0.0009)
Test Rouge 2 (F Measure)	0.0891 (±0.0007)	0.0885 (±0.0006)

Full Mode

In the Full Mode, where additional hyperparameters were tuned for each optimizer, we observe similar performance across all metrics (see Tables 5.28 and 5.29). Among the adaptive optimizers, there is no significant difference in performance that favors one over the others. For the SGD and SGDM optimizers, SGDM demonstrates better test loss scores, although both SGD and SGDM report zero values for all other metrics, continuing to lag behind the adaptive optimizers.

Tab. 5.28: This table presents the mean values of each optimizer's performance, averaged over four different seeds, for the XSum dataset in Full Mode. The table includes results for each optimizer's hyperparameters and learning rates. Due to space constraints, this is Part 1 of 2.

Metrics	NAdam	SGD	SGDM
Test Loss	0.1413 (±0.001)	0.2268 (±0.0013)	0.1947 (±0.0077)
Test F1 (BERTScore)	0.5305 (±0.0015)	0	0
Test Rouge 1 (F Measure)	0.3094 (±0.001)	0	0
Test Rouge 2 (F Measure)	0.0939 (±0.0004)	0	0

Tab. 5.29: This table continues from Part 1, showing the mean values of each optimizer's performance, averaged over four seeds, for the XSum dataset in Full Mode. This table captures additional hyperparameters and optimizers.

Metrics	AdamW	Adam
Test Loss	0.1414 (±0.0011)	0.1413 (±0.001)
Test F1 (BERTScore)	0.5305 (±0.0015)	0.5307 (±0.0013)
Test Rouge 1 (F Measure)	0.3089 (±0.006)	0.3082 (±0.0016)
Test Rouge 2 (F Measure)	0.0945 (±0.0008)	0.0934 (±0.0015)

5.9.4 IWSLT

Basic Mode

In the IWSLT dataset, and in the Basic Mode, we observe similar performance across all metrics for the adaptive optimizers NAdam, AdamW, and Adam (see Tables 5.30 and 5.31). On the other hand, the SGD and SGDM optimizers lag behind the adaptive optimizers, reporting only test loss values while showing zero values for the other metrics.

Tab. 5.30: This table presents the mean values of each optimizer's performance, averaged over four different seeds, for the IWSLT dataset in learning rate tuning mode only. The table contains the results for each optimizer's learning rate. Due to space constraints, this is Part 1 of 2.

Metrics	NAdam	SGD	SGDM
Test Loss	0.0665 (±0.0011)	0.1177 (±0.0014)	0.1179 (±0.0014)
Test F1 (BERTScore)	0.5070 (±0.0005)	0	0
Test Rouge 1 (F Measure)	0.5400 (±0.0012)	0	0
Test Rouge 2 (F Measure)	0.3512 (±0.0034)	0	0

Tab. 5.31: This table continues from Part 1, showing the mean values of each optimizer's performance, averaged over four seeds, for the IWSLT dataset in learning rate tuning mode only. This table captures the remaining optimizers.

Metrics	AdamW	Adam
Test Loss	0.0667 (±0.0012)	0.0668 (±0.0011)
Test F1	0.5082 (±0.0009)	0.5073 (±0.0014)
Test Rouge 1 (F Measure)	0.5402 (±0.0022)	0.5399 (±0.0022)
Test Rouge 2 (F Measure)	0.3504 (±0.0038)	0.3501 (±0.0036)

Full Mode

In the Full Mode, among the adaptive optimizers, we observe similar performance across all metrics (see Tables 5.32 and 5.33) meaning there is no significant difference in performance that favors one over the other adaptive optimizers. However, they do not gain a significant performance boost compared to the Basic Mode. When it comes to the SGD and SGDM optimizers, SGDM demonstrates better test loss scores than SGD, and SGDM manage to report some small values among the metrics, compared to the values of SGD and SGDM when trained in Basic Mode.

Tab. 5.32: This table presents the mean values of each optimizer's performance, averaged over four different seeds, for the IWSLT dataset in Full Mode. The table includes results for each optimizer's hyperparameters and learning rates. Due to space constraints, this is Part 1 of 2.

Metrics	NAdam	SGD	SGDM
Test Loss	0.0661 (±0.0010)	0.1178 (±0.0015)	0.0854 (±0.0011)
Test F1	0.5076 (±0.0005)	0.0000 (±0.0000)	0.0120 (±0.0004)
Test Rouge 1 (F Measure)	0.5434 (±0.0014)	0.0000 (±0.0000)	0.0102 (±0.0008)
Test Rouge 2 (F Measure)	0.3543 (±0.0011)	0.0000 (±0.0000)	0.0065 (±0.0005)

Tab. 5.33: This table continues from Part 1, showing the mean values of each optimizer's performance, averaged over four seeds, for the IWSLT dataset in Full Mode. This table captures additional hyperparameters and optimizers.

Metrics	AdamW	Adam
Test Loss	0.0661 (±0.0011)	0.0661 (±0.0010)
Test F1	0.5073 (±0.0017)	0.5064 (±0.0014)
Test Rouge 1 (F Measure)	0.5430 (±0.0019)	0.5436 (±0.0016)
Test Rouge 2 (F Measure)	0.3546 (±0.0023)	0.3543 (±0.0026)

5.9.5 FLORES

Basic Mode

When it comes to the Flores dataset and in Basic Mode, we can see in Tables 5.34 and 5.35, that we have similar performance across all adaptive optimizers, while the non-adaptive optimizers still lack behind from the adaptive ones. SGD fails to report any meaningful values across all metrics while the SGDM, have better results when it comes to the BERTScore F1, ROUGE 1 (F measure) and ROUGE 2 (F Measure) scores compared to SGD, has a very high test loss (cross entropy loss) compared to every other optimizer used.

Tab. 5.34: This table presents the mean values of each optimizer's performance, averaged over four different seeds, for the Flores dataset in learning rate tuning mode only. The table contains the results for each optimizer's learning rate. Due to space constraints, this is Part 1 of 2.

Metrics	NAdam	SGD	SGDM
Test Loss	0.0767 (±0.0019)	0.1491 (±0.0031)	3.7096 (±3.5263)
Test F1	0.5043 (±0.0033)	0.0474 (±0.0253)	0.3793 (±0.2406)
Test Rouge 1 (F Measure)	0.4891 (±0.0081)	0.0445 (±0.0246)	0.3501 (±0.2243)
Test Rouge 2 (F Measure)	0.3280 (±0.0055)	0.0279 (±0.0168)	0.2256 (±0.1466)

Tab. 5.35: This table continues from Part 1, showing the mean values of each optimizer's performance, averaged over four seeds, for the Flores dataset in learning rate tuning mode only. This table captures the remaining optimizers.

Metrics	Adamw	Adam
Test Loss	0.0791 (±0.0016)	0.0791 (±0.0016)
Test F1	0.5082 (±0.0033)	0.5042 (±0.0022)
Test Rouge 1 (F Measure)	0.4836 (±0.0090)	0.4838 (±0.0086)
Test Rouge 2 (F Measure)	0.3206 (±0.0040)	0.3209 (±0.0039)

Full Mode

In the Full Mode, where additional hyperparameters were tuned for each optimizer, we observe similar performance across the adaptive optimizers and all metrics (see Tables 5.36 and 5.37). Among the adaptive optimizers, there is no significant difference in performance that favors one over the other and no significant improvement compared to the Basic Mode. When it comes to the SGD and SGDM optimizers, SGDM demonstrates better test loss scores than Basic Mode, but still lacks behind from all the adaptive optimizers in the Full Mode. In Addition, SGDM lacks behind SGD as well, an exception when considering all the other datasets.

Tab. 5.36: This table presents the mean values of each optimizer's performance, averaged over four different seeds, for the Flores dataset in Full Mode. The table includes results for each optimizer's hyperparameters and learning rates. Due to space constraints, this is Part 1 of 2.

Metrics	NAdam	SGD	SGDM
Test Loss	0.0756 (±0.0019)	0.1497 (±0.0022)	0.1416 (±0.0164)
Test F1	0.5046 (±0.0031)	0.0462 (±0.0235)	0.0392 (±0.0330)
Test Rouge 1 (F Measure)	0.4898 (±0.0070)	0.0424 (±0.0228)	0.0364 (±0.0300)
Test Rouge 2 (F Measure)	0.3281 (±0.0046)	0.0261 (±0.0155)	0.0230 (±0.0192)

Tab. 5.37: This table continues from Part 1, showing the mean values of each optimizer's performance, averaged over four seeds, for the Flores dataset in Full Mode. This table captures additional hyperparameters and optimizers.

Metrics	AdamW	Adam
Test Loss	0.0755 (±0.0017)	0.0759 (±0.0017)
Test F1	0.5047 (±0.0038)	0.5058 (±0.0015)
Test Rouge 1 (F Measure)	0.4907 (±0.0096)	0.4897 (±0.0080)
Test Rouge 2 (F Measure)	0.3290 (±0.0059)	0.3286 (±0.0042)

5.10 SGD vs. SGDM (20 Epochs)

In this chapter, we present the results of training only the SGD and SGDM optimizers for 20 epochs, as opposed to the 5 epochs used for the other optimizers. We observed that 5 epochs were insufficient for SGD and SGDM to produce meaningful values, that is why we extend the duration of the training allowing for a more comprehensive comparison between SGD and SGDM. This extended training will provide the necessary metrics to determine whether SGD and SGDM exhibit comparable performance to the results reported by Gkouti et al., who concluded that, in general, SGDM was better than SGD in the classification task.

5.10.1 CNN/Dailymail

Basic Mode

In the CNN/DailyMail dataset, Figures 5.44, 5.45, 5.46, 5.47, and 5.48 illustrate the performance of the SGD and SGDM optimizers when trained for 20 epochs in the Basic Mode, as opposed to the 5 epochs used in the previous Chapters 5.8 and 5.9. From these figures, it is evident that both SGD and SGDM optimizers exhibit similar performance across all metrics, with no significant distinction between the two. This contradicts the findings of Gkouti et al, that concluded that SGDM was better compared to SGD when the learning rate was the only tuned hyperparameter.

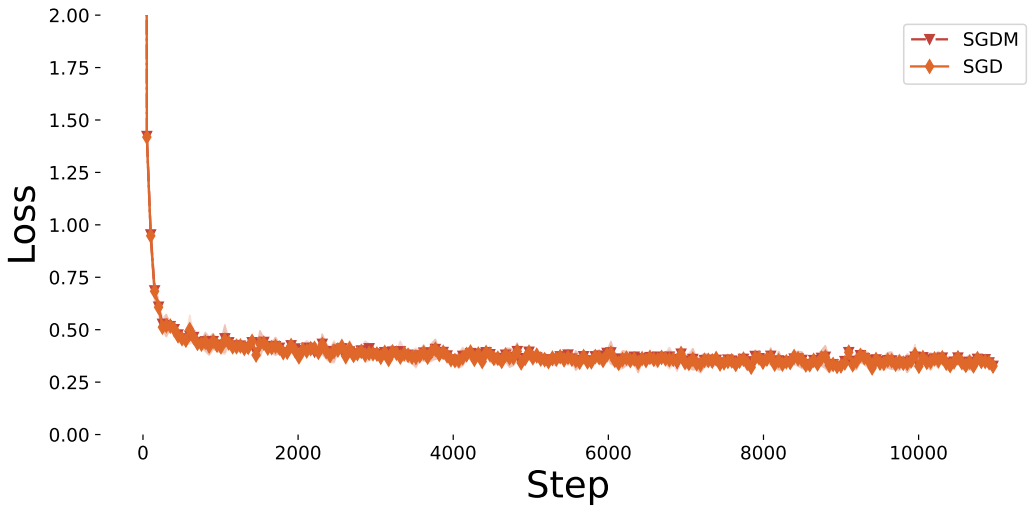

Fig. 5.44: Training Loss over 20 epochs for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

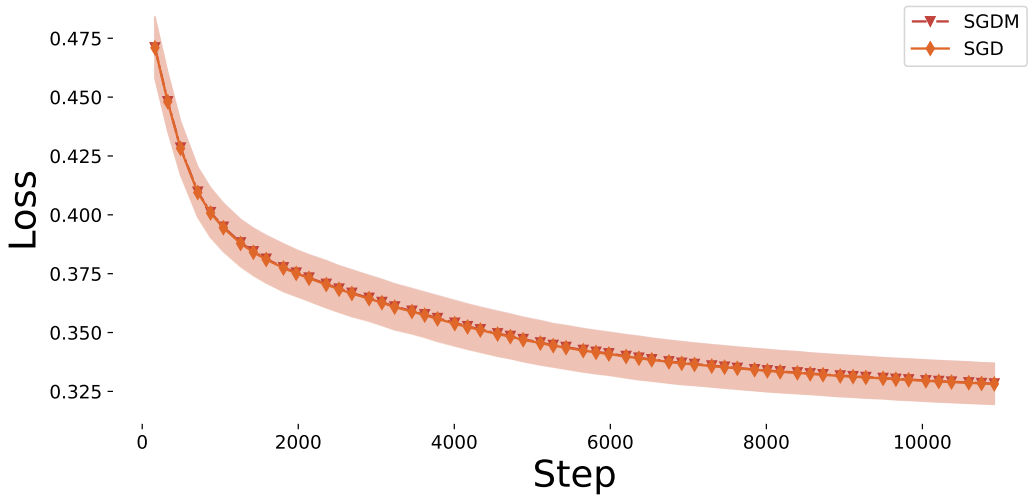

Fig. 5.45: Validation Loss over 20 epochs for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

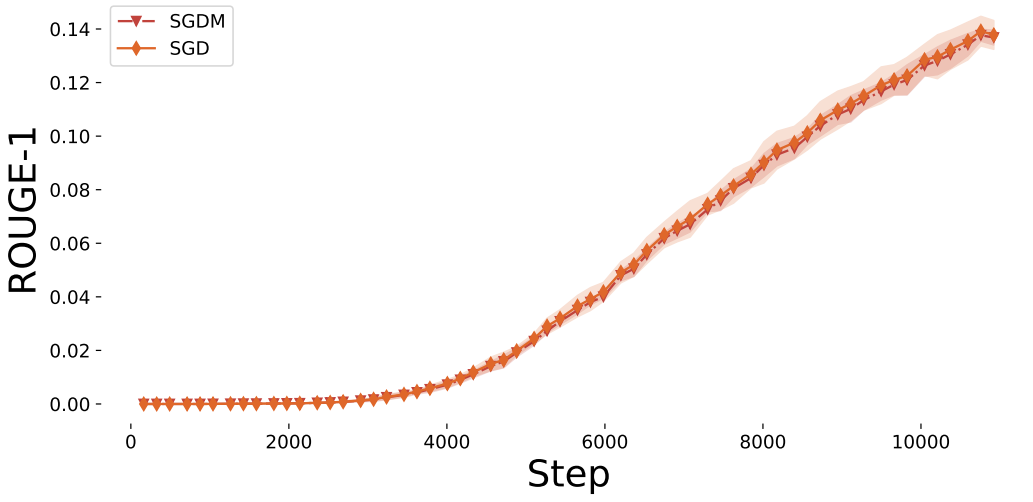

Fig. 5.46: Rouge 1 - F Measure, during validation, over 20 epochs for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

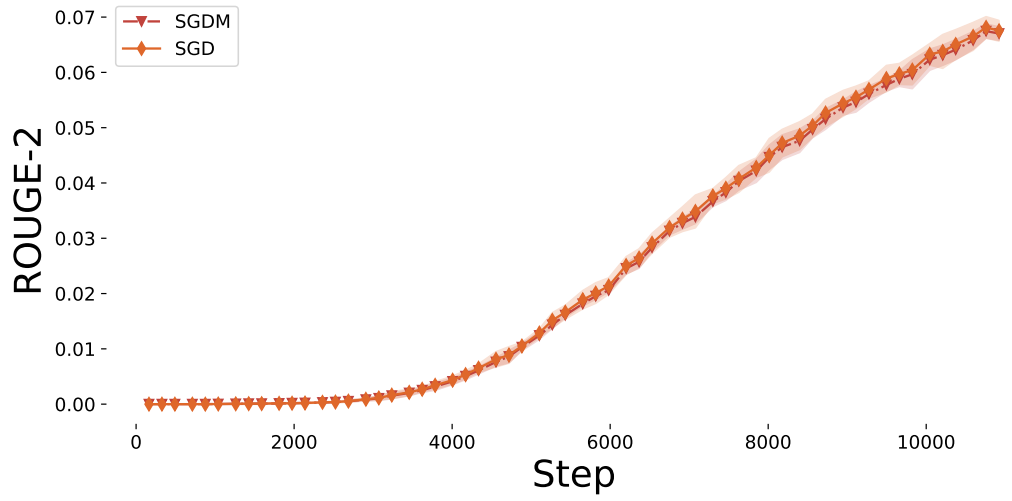

Fig. 5.47: Rouge 2 - F Measure, during validation, over 20 epochs for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

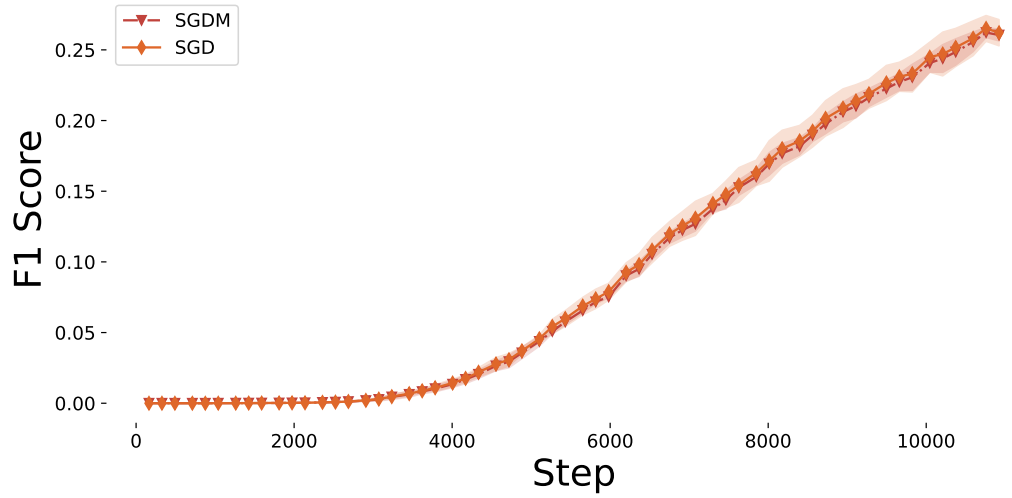

Fig. 5.48: F1 BERTScore, during validation, over 20 epochs for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

Full Mode

In Full Mode, SGDM shows a significant performance improvement compared to Basic Mode. Notably, SGDM outperforms SGD in this setup. Figures 5.49, 5.50, 5.51, 5.52, and 5.53 show that SGDM starts reporting meaningful values around step 1,700, while SGD only begins to do so around step 2,200. Additionally, SGDM achieves nearly double the scores of SGD and maintains higher values across all metrics even by step 2,500. These results indicate that SGDM performs significantly better than SGD when tuning includes more than just the learning rate. This finding differs from the conclusion of Gkouti et al., who reported no significant performance boost for SGD and SGDM with additional hyperparameter tuning. However, the observation that SGDM is superior to SGD remains consistent here.

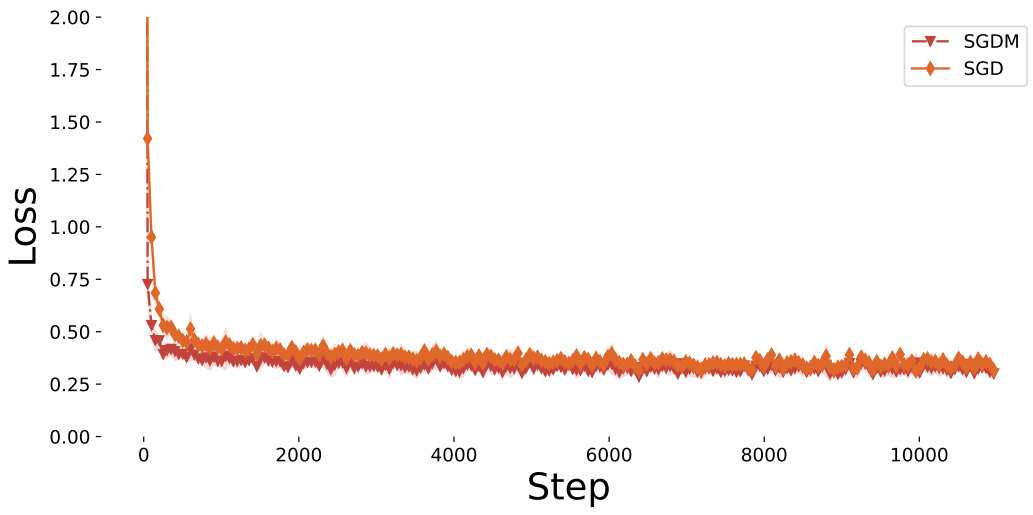

Fig. 5.49: Training Loss over 20 epochs, for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

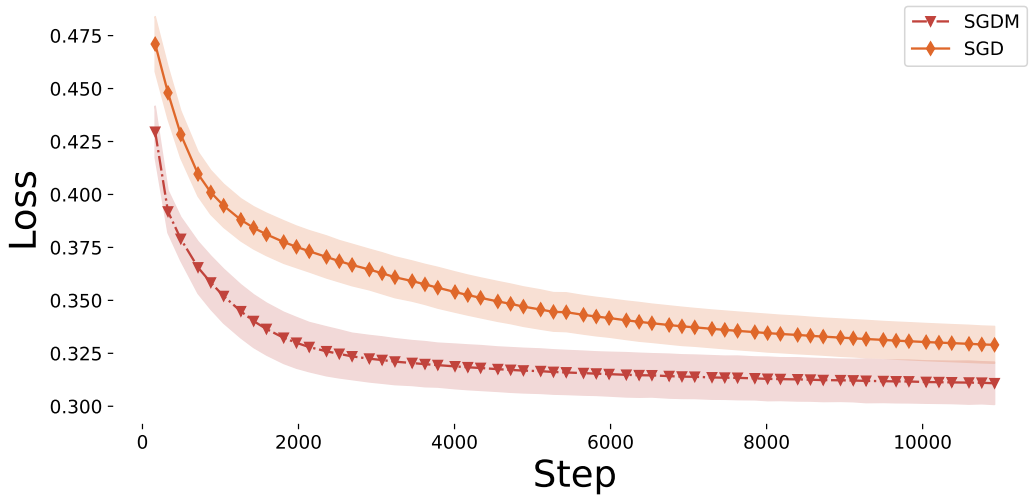

Fig. 5.50: Validation Loss score over 20 epochs, for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

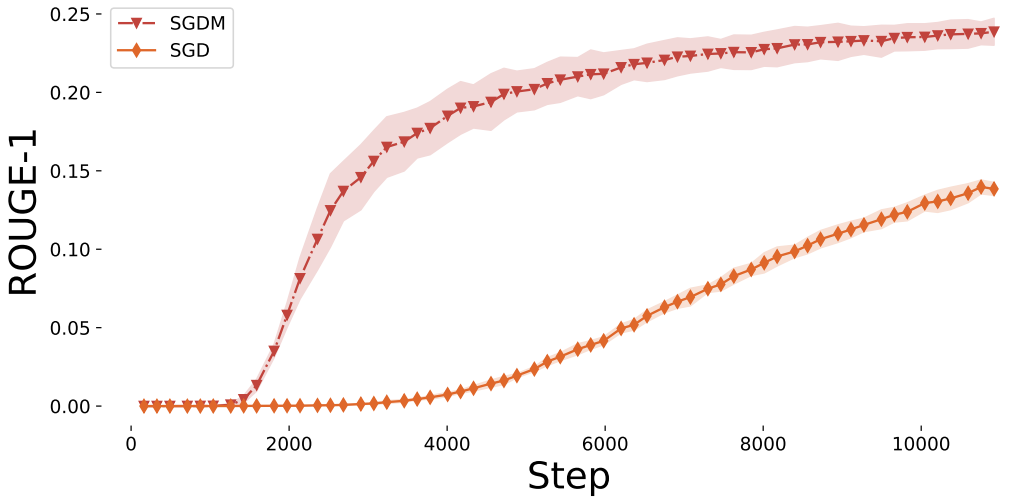

Fig. 5.51: Rouge 1 - F Measure score over 20 epochs, during validation, for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

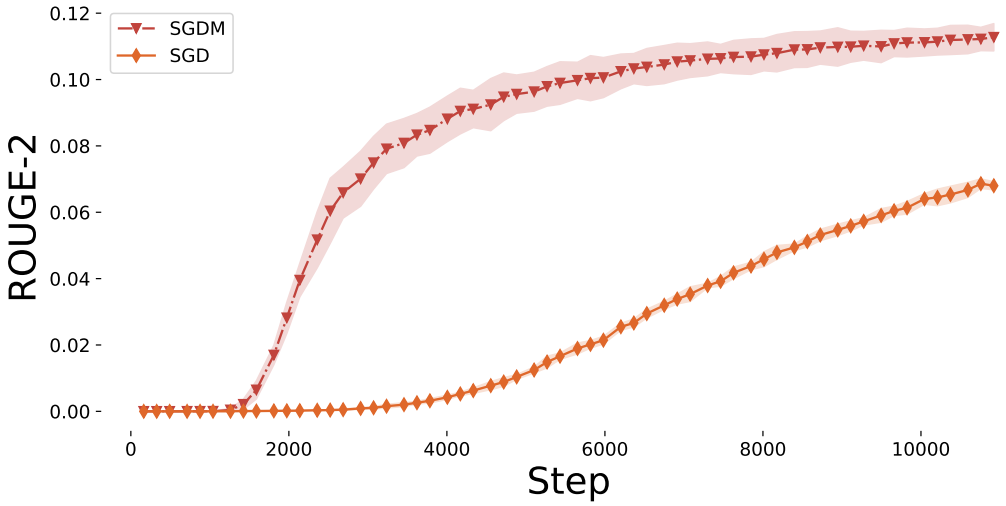

Fig. 5.52: Rouge 2 - F Measure score over 20 epochs, during validation, for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

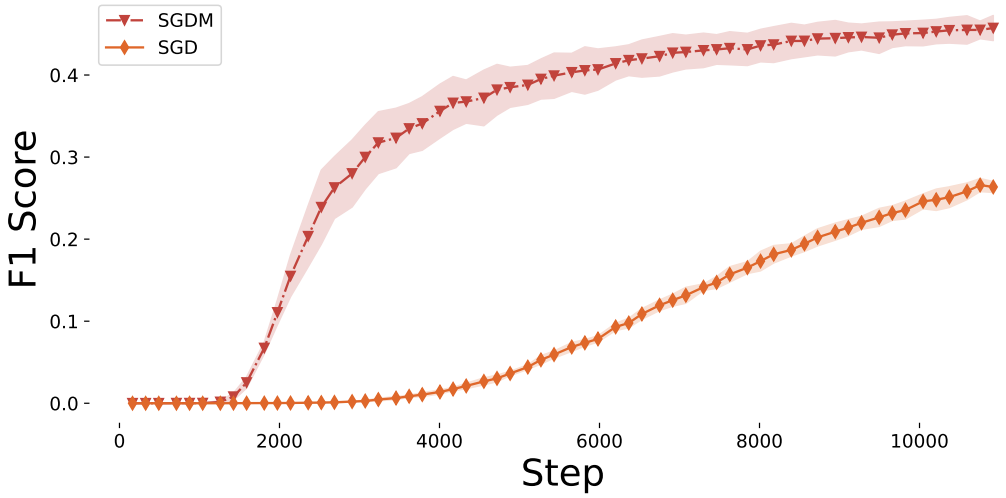

Fig. 5.53: F1 BERTScore score over 20 epochs, during validation, for the CNN/Dailymail dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

5.10.2 SAMSum

Basic Mode

For the SAMSum dataset, during the Basic Mode, we observe no notable distinction in performance between the SGD and SGDM optimizers (see Figures 5.54, 5.56, 5.57, and 5.58). An interesting observation is found in Figures 5.56, 5.57, and 5.58, where no values are reported for either SGD or SGDM, despite extending the training to 20 epochs. This suggests that, for this particular dataset, more training epochs may be required to observe any meaningful performance gains from these two optimizers.

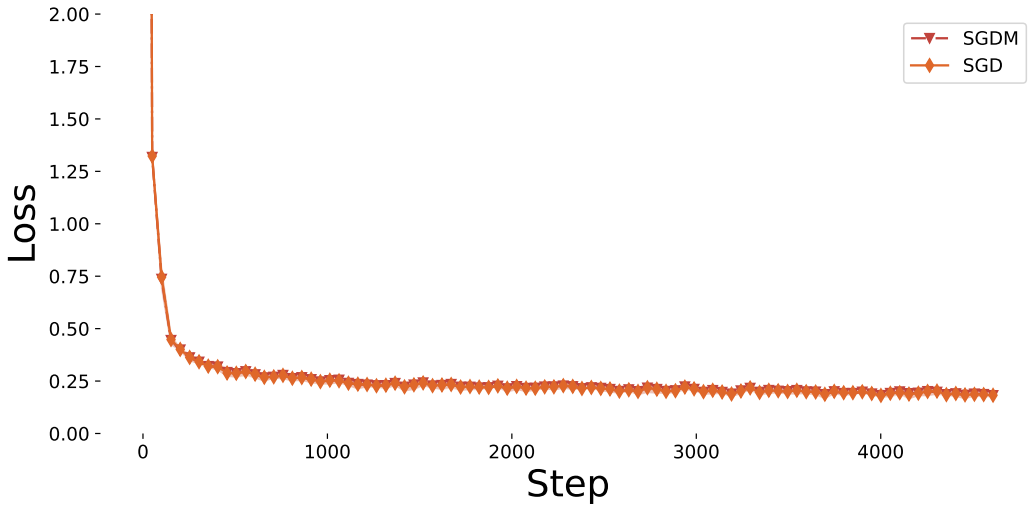

Fig. 5.54: Training Loss over 20 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

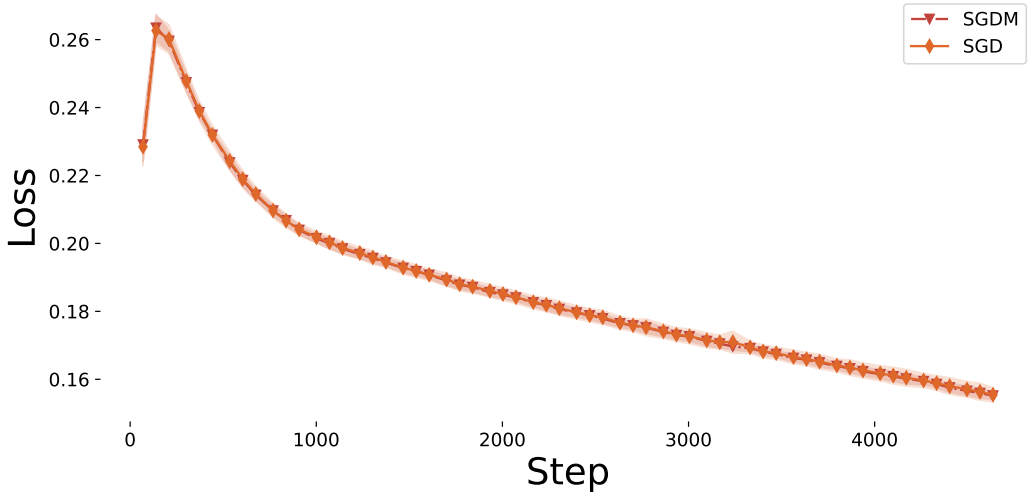

Fig. 5.55: Validation Loss over 20 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

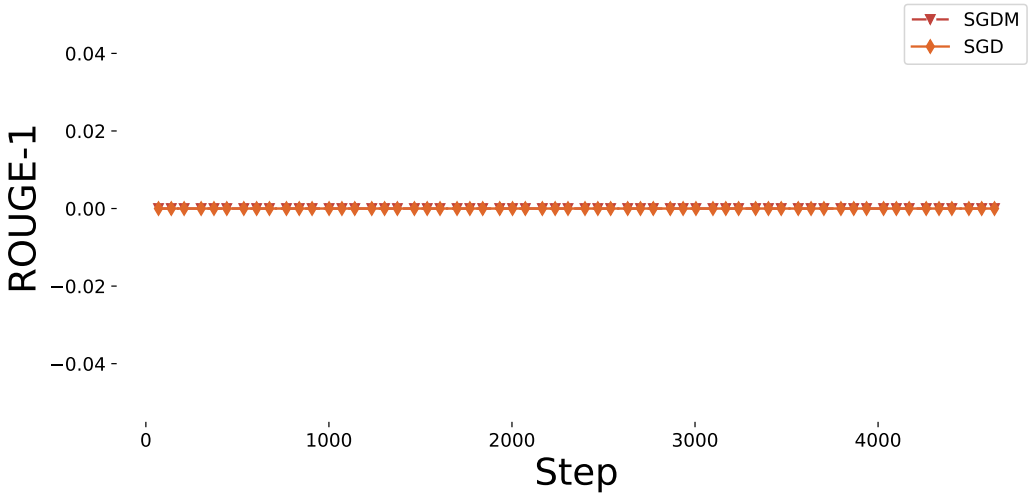

Fig. 5.56: Rouge 1 - F Measure score, during validation, over 20 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

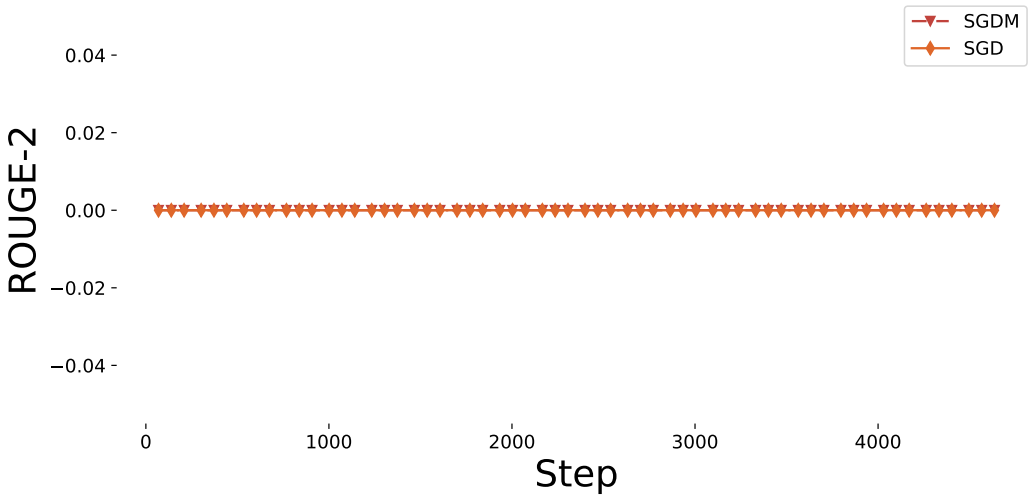

Fig. 5.57: Rouge 2 - F Measure score, during validation, over 20 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

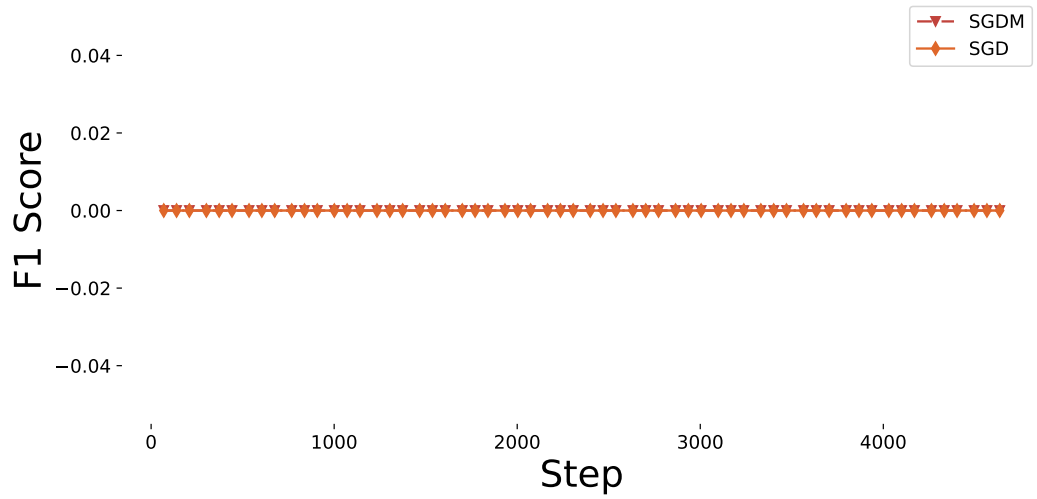

Fig. 5.58: F1 BERTScore score, during validation, over 20 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

Full Mode

In the Full Mode, we observe from Figures 5.59 and 5.60, that the SGDM optimizer significantly outperforms SGD in both training and validation loss. More importantly, the SGDM optimizer is now reporting values across all other metrics (see Figures 5.61, 5.62 and 5.63), a noticeable improvement compared to the Basic Mode where it reported negligible values. This improvement is evident even in the early stages of training, around the 1.500 step mark out of the 4.500 total steps, and continues to show rising values through to the final steps.

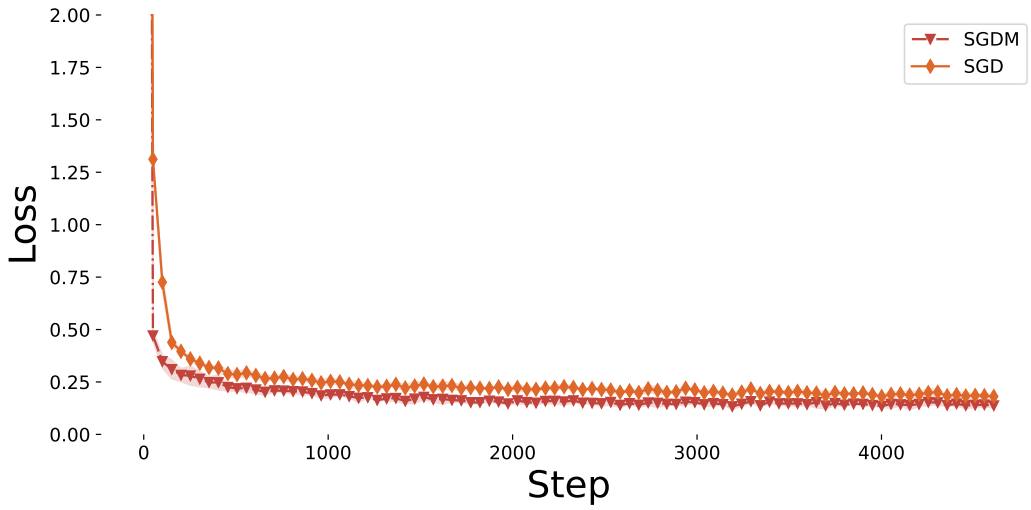

Fig. 5.59: Training Loss score over 20 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

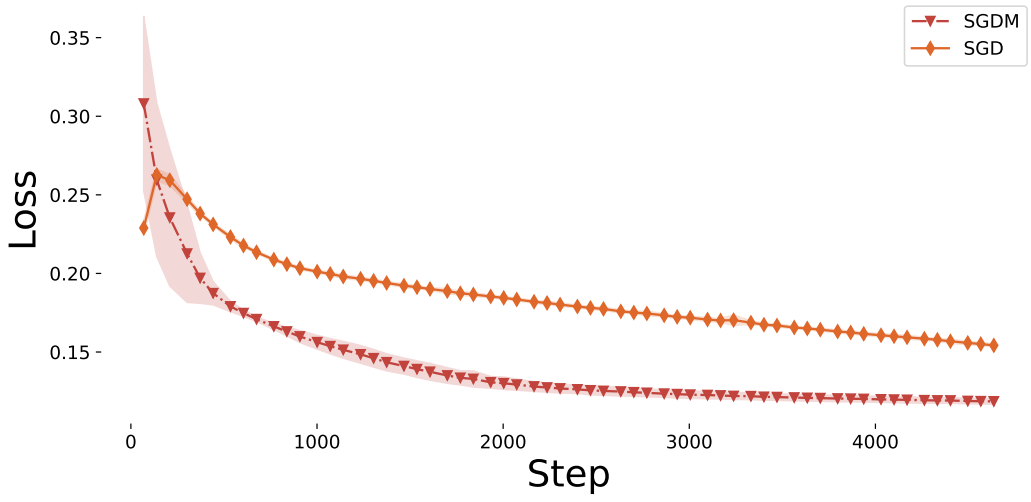

Fig. 5.60: Validation Loss score over 20 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

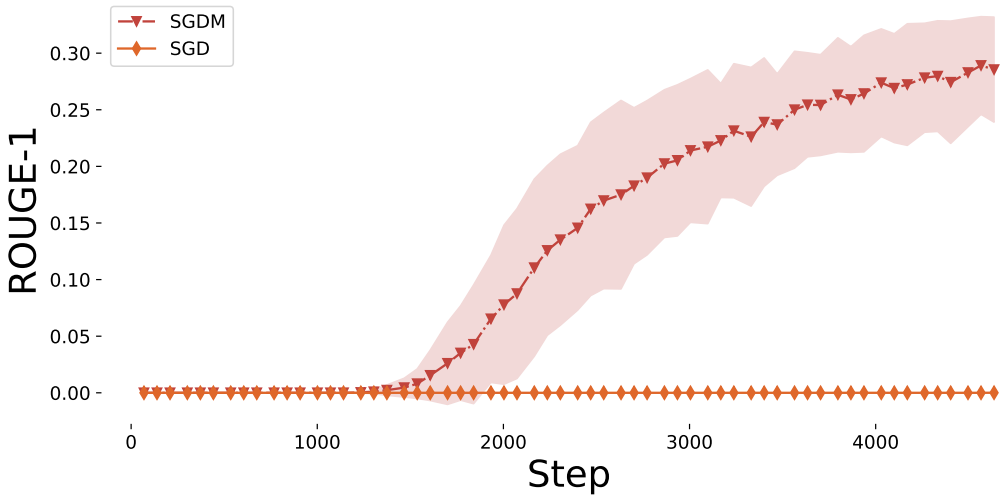

Fig. 5.61: Rouge 1 - F Measure score, during validation, over 20 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

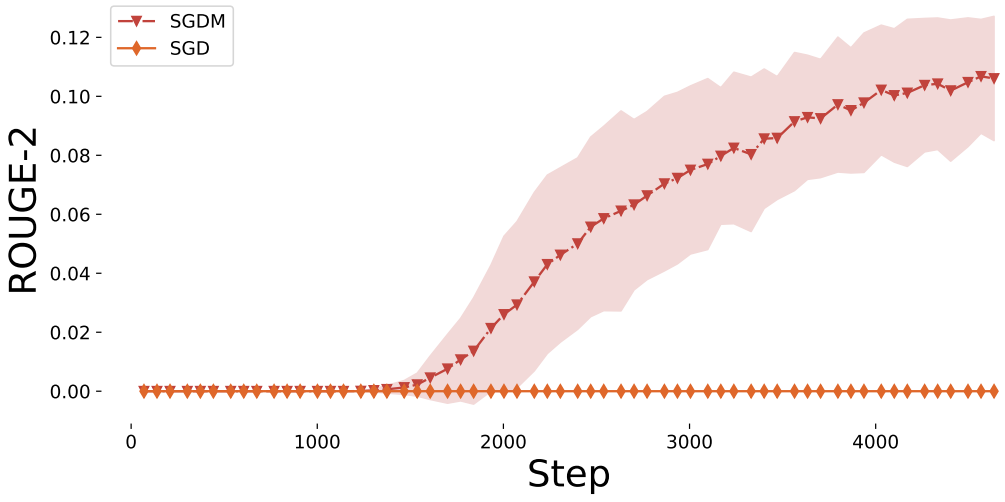

Fig. 5.62: Rouge 2 - F Measure score, during validation, over 20 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

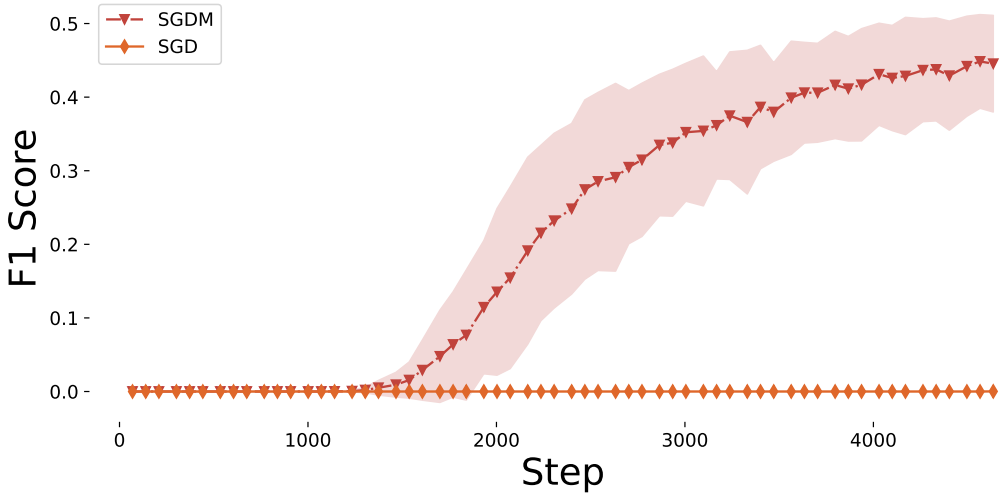

Fig. 5.63: F1 BERTScore score, during validation, over 20 epochs for the SAMSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode.

5.10.3 XSum

Basic Mode

In the XSum dataset, within the Basic Mode, we observe an interesting outcome (see Figures 5.64, 5.65, 5.66, 5.67 and 5.68). The SGD optimizer performs slightly better compared to SGDM in all metrics (except training loss). However, we must note that the difference in performance of those two are in the 3^{rd} decimal digit, which is negligible.

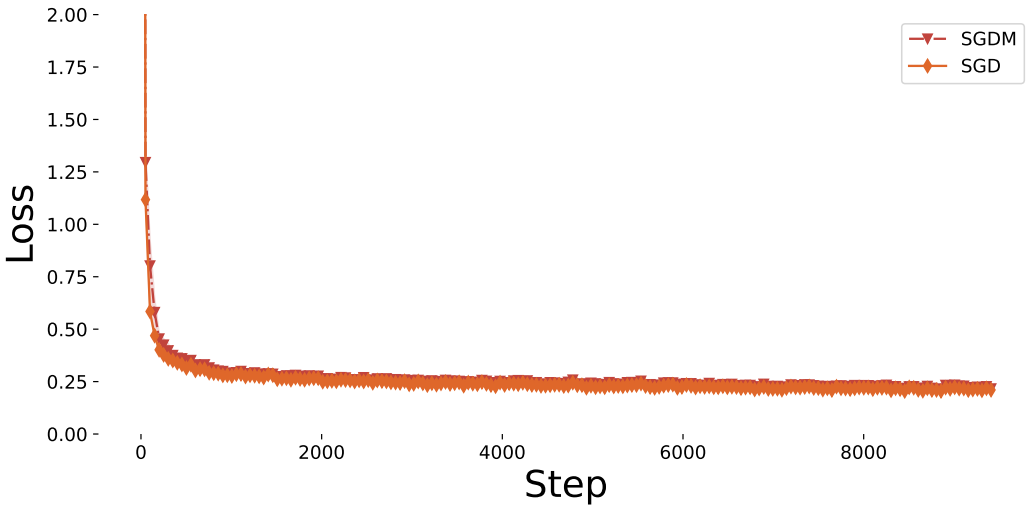

Fig. 5.64: Training Loss score over 20 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

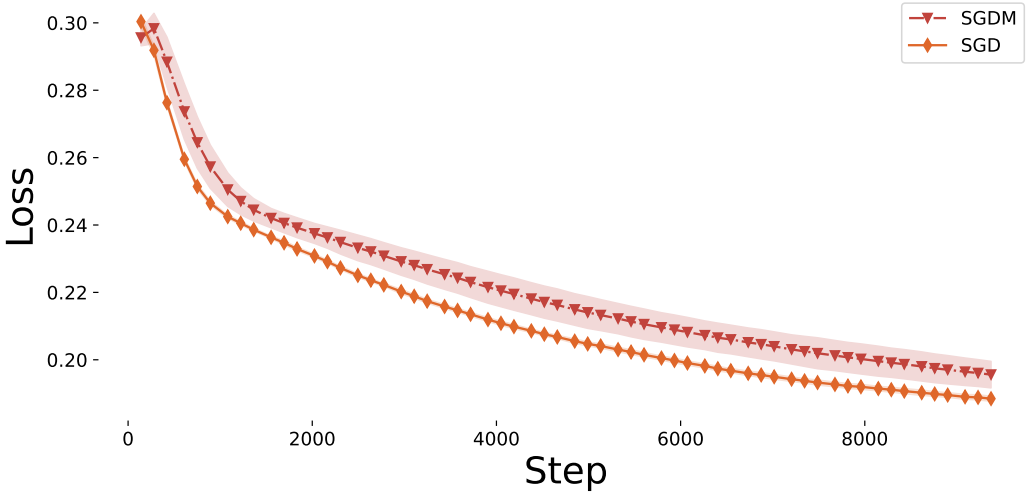

Fig. 5.65: Validation Loss score over 20 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

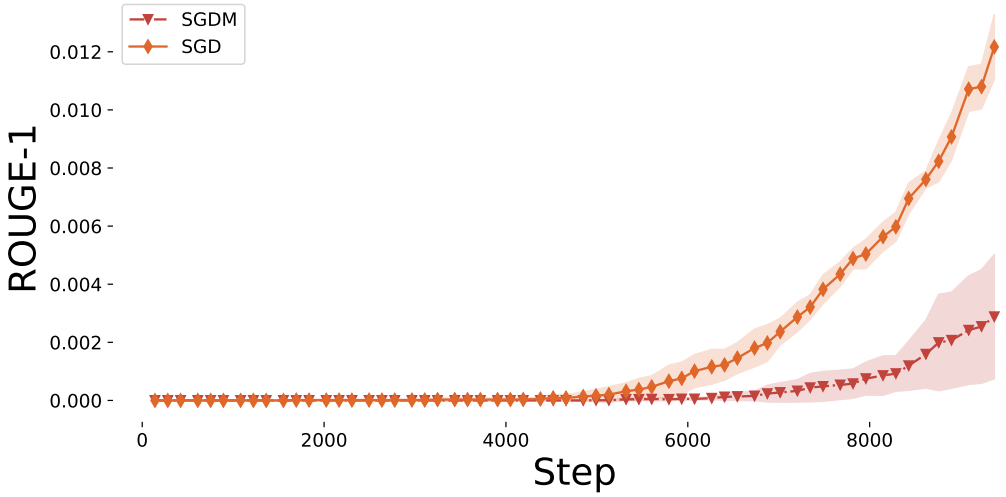

Fig. 5.66: Rouge 1 - F Measure score, during validation, over 20 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

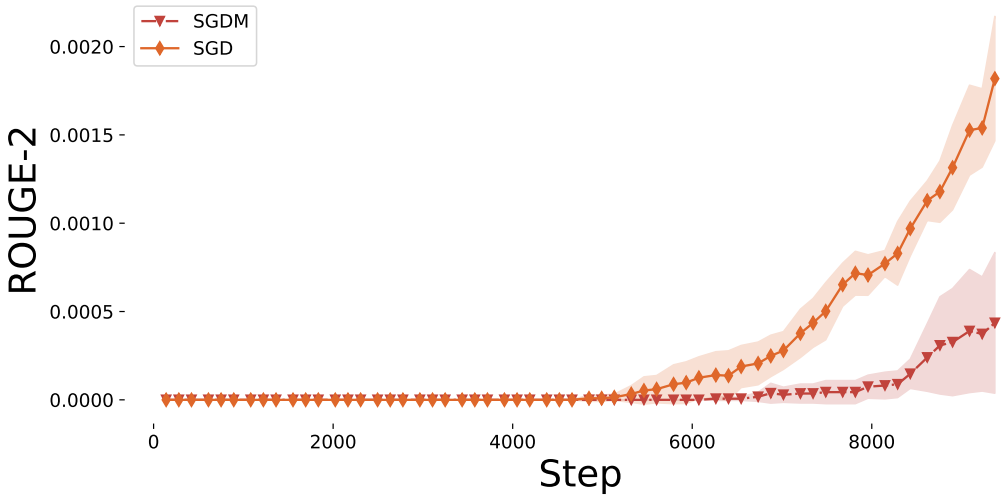

Fig. 5.67: Rouge 2 - F Measure score, during validation, over 20 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

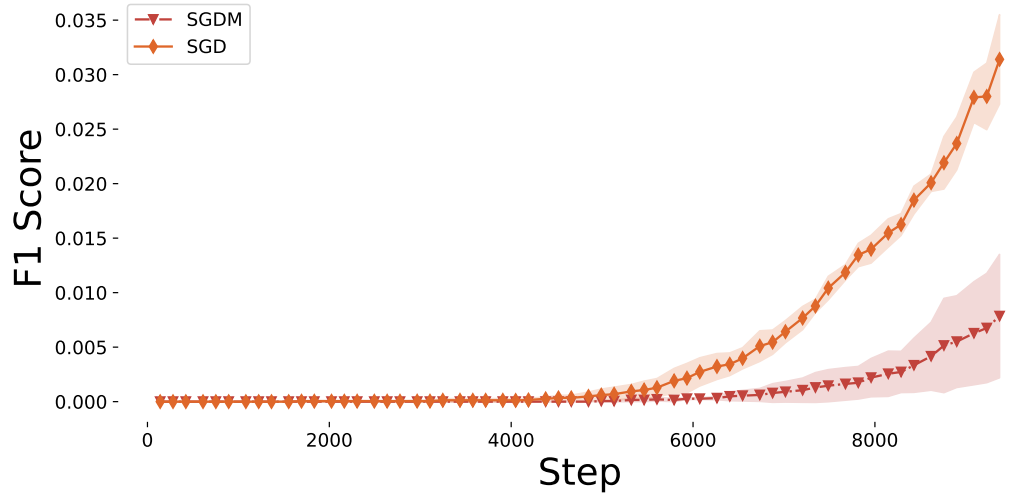

Fig. 5.68: F1 BERTScore score, during validation, over 20 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Basic Mode.

Full Mode

In the Full Mode, where additional hyperparameters were tuned for each optimizer, we see in Figures 5.69, 5.70, 5.71, 5.72, and 5.73 that SGDM outperforms plain SGD, showing better performance and reporting improved values compared to the Basic Mode. SGDM surpasses SGD significantly in metrics such as ROUGE-1, ROUGE-2, and BERTScore F1. In addition, SGD performs a little worse in the Full Mode than when only its learning rate was tuned.

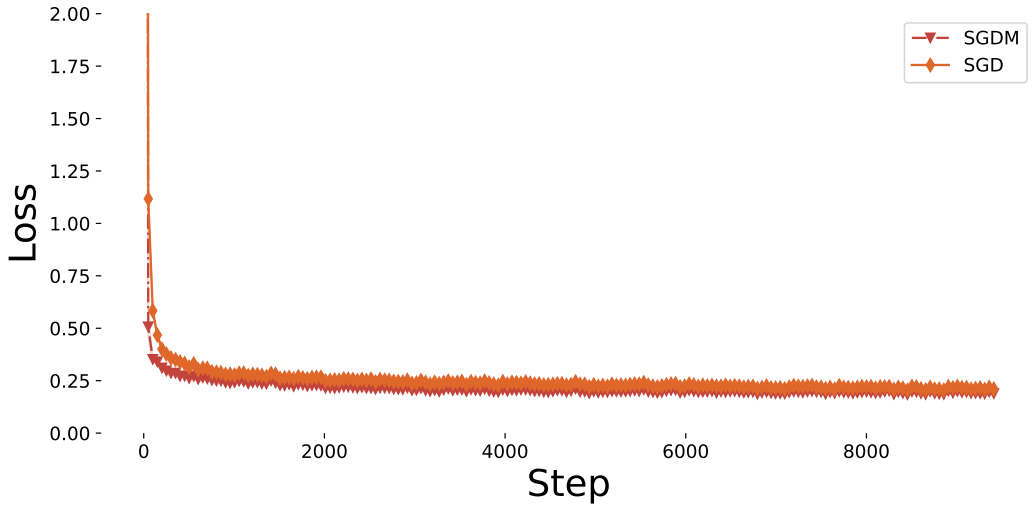

Fig. 5.69: Training Loss score, over 20 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode, where all hyperparameters were tuned for each optimizer.

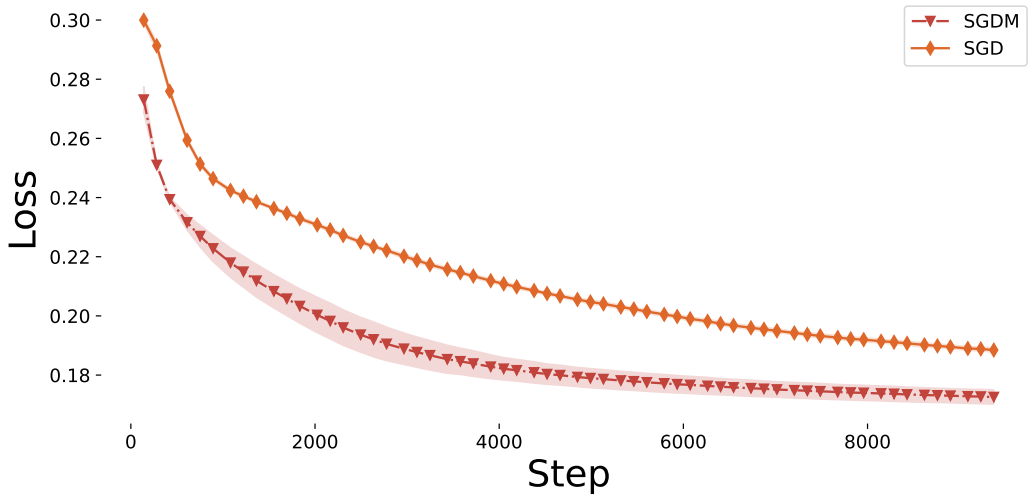

Fig. 5.70: Validation Loss score, over 20 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode, where all hyperparameters were tuned for each optimizer.

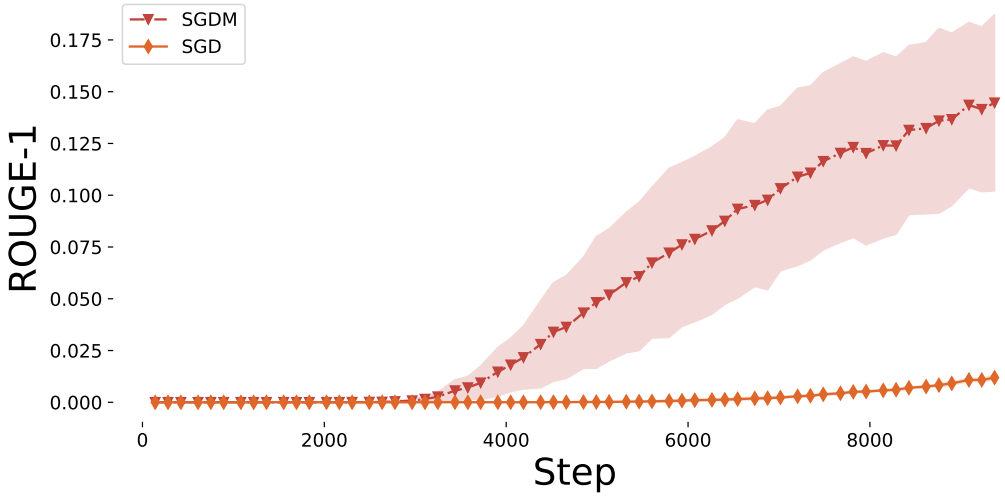

Fig. 5.71: Rouge 1 - F Measure score, during validation, over 20 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode, where all hyperparameters were tuned for each optimizer.

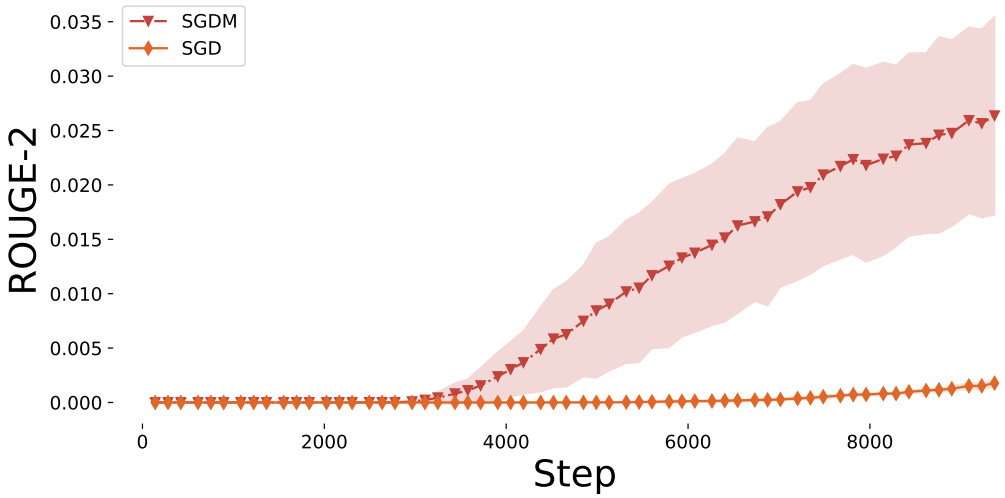

Fig. 5.72: Rouge 2 - F Measure score, during validation, over 20 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode, where all hyperparameters were tuned for each optimizer.

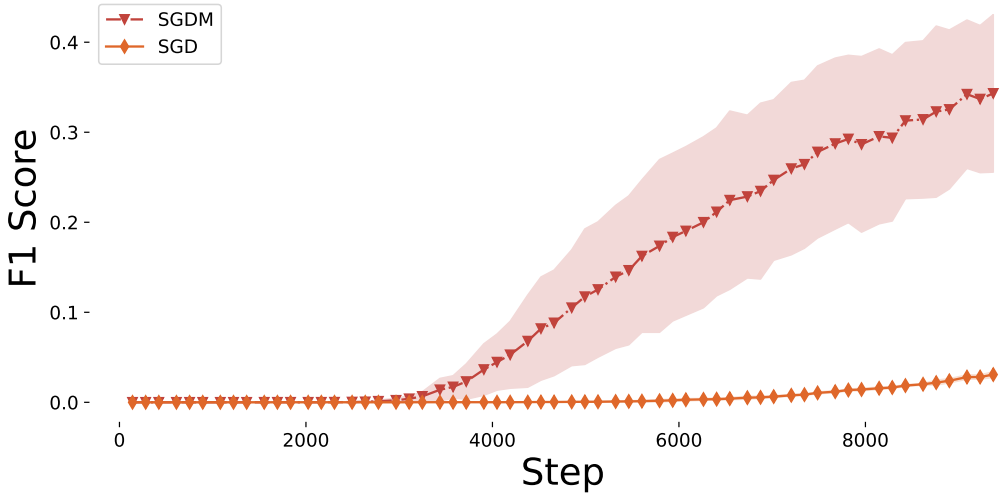

Fig. 5.73: F1 BERTScore score, during validation, over 20 epochs for the XSum dataset. The curve represents the mean performance of each optimizer, averaged across four different seeds, with shaded areas indicating the deviation across the seed runs. The training was conducted in Full Mode, where all hyperparameters were tuned for each optimizer.

Conclusions

6

This thesis builds upon the work of Gkouti et al. by examining the efficacy of hyperparameter tuning across various optimization algorithms in deep learning for Natural Language Processing (NLP) tasks. The primary objective was to investigate whether the observations regarding hyperparameter tuning and optimizer performance, originally made in encoder-only models and classification tasks, hold true across a wider spectrum of model architectures, tasks, and datasets. Specifically, we focused on an additional encoder-decoder model, the T5-Small, and the tasks of text summarization and machine translation, leveraging diverse datasets to ensure robust and generalizable conclusions.

Our results offer several key insights that both corroborate and extend the findings of Gkouti et al. For adaptive optimizers such as Adam, NAdam, and AdamW, and for the contextual metric BERTScore, there is no significant performance improvement when tuning all hyperparameters compared to tuning only the learning rate. However, in rare instances, we observed a performance boost in metrics like ROUGE-1 and ROUGE-2. It is important to note that these metrics lack the contextual depth of BERTScore, which should therefore be given greater consideration. An other key finding is that the performance of the adaptive optimizers is similar to each other, both in basic and full tuning modes, and they do not gain significant performance improvements when tuned in full mode. This finding has significant implications for the deep learning community, as it suggests that practitioners can achieve near-optimal performance with any of the adaptive optimizers by focusing solely on learning rate tuning, thereby saving substantial computational resources and time.

In contrast, our analysis of Stochastic Gradient Descent (SGD) and SGD with Momentum (SGDM) reveals a more nuanced picture. Both SGD and SGDM generally underperformed relative to adaptive optimizers across all of our experiments. However, SGDM exhibited a notable performance boost when additional hyperparameters were fine-tuned, compared to tuning only the learning rate. This underscores the importance of comprehensive hyperparameter tuning for SGDM when it is employed as the optimization algorithm. Furthermore, while SGD and SGDM performed similarly when only the learning rate was tuned, SGDM consistently outperformed SGD when multiple hyperparameters were optimized. These findings stand in stark contrast to those of Gkouti et al., who reported no significant performance gain when tuning beyond the learning rate for SGD and SGDM, whereas our results demonstrate the exact opposite.

Crucially, our research shows that these patterns hold across a broader range of NLP tasks and datasets than previously explored. By extending the investigation to include tasks such as text summarization and machine translation across diverse datasets, we demonstrate that some insights from Gkouti et al.'s work on classification tasks do not generalize to more complex NLP applications. While we agree that adaptive optimizers

perform similarly under both basic and full tuning modes and gain minimal performance improvement in full mode, we diverge significantly in our findings for SGD and SGDM. Specifically, our results indicate that SGD and SGDM exhibit similar performance when tuning only the learning rate, contradicting Gkouti et al.'s conclusion that SGDM is superior in this scenario. Moreover, SGDM gains a significant performance boost in full tuning mode, directly opposing Gkouti et al.'s claim of negligible improvement when tuning more than just the learning rate. Lastly, we found no evidence to support the assertion that SGDM can perform comparably to adaptive optimizers under full tuning; SGDM consistently lagged behind adaptive optimizers in both basic and full modes.

In conclusion, this thesis advances the discussion on optimization strategies in deep learning by providing empirical evidence to support targeted hyperparameter tuning approaches. Our findings suggest that for adaptive optimizers, focusing on learning rate tuning, and selecting any of them, is sufficient to achieve optimal or near-optimal performance. However, for traditional optimizers like SGDM, a comprehensive tuning strategy is essential to unlock their full potential. These insights not only offer practical guidance for efficient model training but also pave the way for further research into the theoretical foundations of optimizer behavior across diverse NLP tasks and model architectures.

Bibliography

[AK15] Mariette Awad and Rahul Khanna. "Machine Learning". In: *Efficient Learning Machines: Theories, Concepts, and Applications for Engineers and System Designers*. Berkeley, CA: Apress, 2015, pp. 1–18.

[AT03] Juan A. Alonso and Gregor Thurmair. "The Comprendium Translator system". In: *Proceedings of Machine Translation Summit IX: System Presentations*. New Orleans, USA, Sept. 2003.

[Ban+23] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, et al. "A Multitask, Multilingual, Multimodal Evaluation of ChatGPT on Reasoning, Hallucination, and Interactivity". In: *AACL 2023* (Nov. 2023). 45 pages, AACL 2023. arXiv: 2302.04023 [cs.CL].

[BB19] Shikha Bordia and Samuel R. Bowman. "Identifying and Reducing Gender Bias in Word-Level Language Models". In: *CoRR* abs/1904.03035 (2019). arXiv: 1904.03035.

[BL05] Satanjeev Banerjee and Alon Lavie. "METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments". In: *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*. Ed. by Jade Goldstein, Alon Lavie, Chin-Yew Lin, and Clare Voss. Ann Arbor, Michigan: Association for Computational Linguistics, June 2005, pp. 65–72.

[Bro+93] Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. "The Mathematics of Statistical Machine Translation: Parameter Estimation". In: *Computational Linguistics* 19.2 (1993). Ed. by Julia Hirschberg, pp. 263–311.

[Cal+07] Chris Callison-Burch, Cameron Fordyce, Philipp Koehn, Christof Monz, and Josh Schroeder. "(Meta-) evaluation of machine translation". In: (June 2007), pp. 136–158.

[Cau47] Augustin-Louis Cauchy. *Méthode générale pour la résolution des systèmes d'équations simultanées*. Gallica, 1847, pp. 536–538.

[Cet+17] Mauro Cettolo, Marcello Federico, Luisa Bentivogli, et al. "Overview of the IWSLT 2017 Evaluation Campaign". In: *Proceedings of the 14th International Conference on Spoken Language Translation*. Tokyo, Japan: International Workshop on Spoken Language Translation, Dec. 2017, pp. 2–14.

[Dev+18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *arXiv preprint arXiv:1810.04805* (Oct. 2018). State-of-the-art results on eleven NLP tasks. arXiv: 1810.04805 [cs.CL].

[DHS11] 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.

[Dod+21] Jesse Dodge, Maarten Sap, Ana Marasovic, et al. "Documenting the English Colossal Clean Crawled Corpus". In: *CoRR* abs/2104.08758 (2021). arXiv: 2104.08758.

[FTR09] Mikel L. Forcada, Francis M. Tyers, and Gema Ramírez-Sánchez. "The Apertium machine translation platform: Five years on". In: *Proceedings of the First International Workshop on Free/Open-Source Rule-Based Machine Translation*. Ed. by Juan Antonio Pérez-Ortiz, Felipe Sánchez-Martinez, and Francis M. Tyers. Alacant, Spain, Nov. 2009, pp. 3–10.

[GBM15] Yvette Graham, Timothy Baldwin, and Nitika Mathur. "Accurate Evaluation of Segment-level Machine Translation Metrics". In: Jan. 2015, pp. 1183–1191.

[Gko+24] Nefeli Gkouti, Prodromos Malakasiotis, Stavros Toumpis, and Ion Androutsopoulos. "Should I try multiple optimizers when fine-tuning a pre-trained Transformer for NLP tasks? Should I tune their hyperparameters?" In: *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Yvette Graham and Matthew Purver. St. Julian's, Malta: Association for Computational Linguistics, Mar. 2024, pp. 2555–2574.

[Gli+19] Bogdan Gliwa, Iwona Mochol, Maciej Biesek, and Aleksander Wawer. "SAMSum Corpus: A Human-annotated Dialogue Dataset for Abstractive Summarization". In: *Proceedings of the 2nd Workshop on New Frontiers in Summarization*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 70–79.

[Goy+21] Naman Goyal, Cynthia Gao, Vishrav Chaudhary, et al. "The FLORES-101 Evaluation Benchmark for Low-Resource and Multilingual Machine Translation". In: 2021.

[Gra15] Yvette Graham. "Re-evaluating Automatic Summarization with BLEU and 192 Shades of ROUGE". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Ed. by Lluís Màrquez, Chris Callison-Burch, and Jian Su. Lisbon, Portugal: Association for Computational Linguistics, Sept. 2015, pp. 128–137.

[Guz+19] Francisco Guzmán, Peng-Jen Chen, Myle Ott, et al. "Two New Evaluation Datasets for Low-Resource Machine Translation: Nepali-English and Sinhala-English". In: 2019.

[He+16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778.

[Her+15] Karl Moritz Hermann, Tomás Kociský, Edward Grefenstette, et al. "Teaching Machines to Read and Comprehend". In: *NIPS*. 2015, pp. 1693–1701.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[Hua+23] Lei Huang, Weijiang Yu, Weitao Ma, et al. "A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions". In: *arXiv preprint arXiv:2311.05232* (Nov. 2023). Work in progress; 49 pages. arXiv: 2311.05232 [cs.CL].

[HVD15] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. "Distilling the Knowledge in a Neural Network". In: *ArXiv* abs/1503.02531 (2015).

[JJ00] G.A. Jones and J.M. Jones. *Information and Coding Theory*. Springer Undergraduate Mathematics Series. Springer London, 2000.

[JL22] Samy Jelassi and Yuanzhi Li. "Towards understanding how momentum improves generalization in deep learning". In: *International Conference on Machine Learning*. 2022.

[KB14] Diederik Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations (ICLR)*. San Diega, CA, USA, 2014.

[Kin+75] J. Peter Kincaid, Robert P. Jr. Fishburne, Richard L. Rogers, and Brad S. Chissom. *Derivation Of New Readability Formulas (Automated Readability Index, Fog Count And Flesch Reading Ease Formula) For Navy Enlisted Personnel*. Tech. rep. 56. Institute for Simulation and Training, 1975.

[Koe09] Philipp Koehn. *Statistical Machine Translation*. Cambridge University Press, 2009.

[KW52] Jack Kiefer and Jacob Wolfowitz. "A Stochastic Approximation Method". In: *The Annals of Mathematical Statistics* 23.3 (1952), pp. 462–466.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), p. 436.

[LeC+98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[LH17] Ilya Loshchilov and Frank Hutter. "Fixing Weight Decay Regularization in Adam". In: *CoRR* abs/1711.05101 (2017). arXiv: 1711.05101.

[Lin04] 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.

[Lop08] Adam Lopez. "Statistical machine translation". In: *ACM Comput. Surv.* 40.3 (Aug. 2008).

[LTP22] Xin Liu, Wei Tao, and Zhisong Pan. "A Convergence Analysis of Nesterov's Accelerated Gradient Method in Training Deep Linear Neural Networks". In: *Inf. Sci.* 612 (2022), pp. 898–925.

[Luo+19] Liangchen Luo, Yuanhao Xiong, Yan Liu, and Xu Sun. "Adaptive Gradient Methods with Dynamic Bound of Learning Rate". In: *CoRR* abs/1902.09843 (2019). arXiv: 1902.09843.

[MB14] Matouš Macháček and Ondřej Bojar. "Results of the WMT14 Metrics Shared Task". In: *Proceedings of the Ninth Workshop on Statistical Machine Translation*. Ed. by Ondřej Bojar, Christian Buck, Christian Federmann, et al. Baltimore, Maryland, USA: Association for Computational Linguistics, June 2014, pp. 293–301.

[MPR21] Nicholas Meade, Elinor Poole-Dayan, and Siva Reddy. "An Empirical Survey of the Effectiveness of Debiasing Techniques for Pre-trained Language Models". In: *Annual Meeting of the Association for Computational Linguistics*. 2021.

[NCL18] Shashi Narayan, Shay B. Cohen, and Mirella Lapata. "Don't Give Me the Details, Just the Summary! Topic-Aware Convolutional Neural Networks for Extreme Summarization". In: *ArXiv* abs/1808.08745 (2018).

[Nes83] Yurii Nesterov. "A method for unconstrained convex minimization problem with the rate of convergence O(1/k)". In: *Doklady AN USSR* 269 (1983), pp. 543–547.

[NLL22] James Cross NLLB Team Marta R. Costa-jussà. "No Language Left Behind: Scaling Human-Centered Machine Translation". In: (2022).

[NP04] Ani Nenkova and Rebecca Passonneau. "Evaluating Content Selection in Summarization: The Pyramid Method". In: *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics: HLT-NAACL 2004*. Boston, Massachusetts, USA: Association for Computational Linguistics, May 2004, pp. 145–152.

[Pap+02] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. "BLEU: a method for automatic evaluation of machine translation". In: *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. ACL '02. Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, pp. 311–318.

[Pos18] Matt Post. "A Call for Clarity in Reporting BLEU Scores". In: *Proceedings of the Third Conference on Machine Translation: Research Papers*. Ed. by Ondřej Bojar, Rajen Chatterjee, Christian Federmann, et al. Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 186–191.

[Pri23] Simon J.D. Prince. *Understanding Deep Learning*. The MIT Press, 2023.

[Qia99] Ning Qian. "On the momentum term in gradient descent learning algorithms". In: *Neural Networks: The Official Journal of the International Neural Network Society* 12.1 (1999), pp. 145–151.

[Rad+19] Alec Radford, Jeff Wu, Rewon Child, et al. "Language Models are Unsupervised Multitask Learners". In: (2019).

[Raf+19] Colin Raffel, Noam Shazeer, Adam Roberts, et al. “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”. In: *arXiv preprint arXiv:1910.10683* (Oct. 2019). State-of-the-art results on various benchmarks. arXiv: 1910.10683 [cs.LG].

[RKK19] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. “On the Convergence of Adam and Beyond”. In: *CoRR* abs/1904.09237 (2019). arXiv: 1904.09237.

[RM51] Herbert Robbins and Sutton Monro. “A Stochastic Approximation Method”. In: *The Annals of Mathematical Statistics* 22.3 (1951), pp. 400–407.

[Sam59] A. L. Samuel. “Some Studies in Machine Learning Using the Game of Checkers”. In: *IBM Journal of Research and Development* 3.3 (1959), pp. 210–229.

[San+20] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. “DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter”. In: *Proceedings of the 5th Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS 2019*. 5 pages, 1 figure, 4 tables. Accepted at the 5th Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS 2019. Feb. 2020. arXiv: 1910.01108 [cs.CL].

[Sch+23] Hendrik Schuff, Lindsey Vanderlyn, Heike Adel, and Thang Vu. “How to do human evaluation: A brief introduction to user studies in NLP”. In: *Natural Language Engineering* 29.5 (Feb. 2023), pp. 1–24.

[See13] J. Seely. *Oxford Guide to Effective Writing and Speaking: How to Communicate Clearly*. OUP Oxford, 2013.

[SLM17] Abigail See, Peter J. Liu, and Christopher D. Manning. “Get To The Point: Summarization with Pointer-Generator Networks”. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, July 2017, pp. 1073–1083.

[Sno+06] Matthew Snover, Bonnie Dorr, Rich Schwartz, Linnea Micciulla, and John Makhoul. “A Study of Translation Edit Rate with Targeted Human Annotation”. In: *Proceedings of the 7th Conference of the Association for Machine Translation in the Americas: Technical Papers*. Cambridge, Massachusetts, USA: Association for Machine Translation in the Americas, Aug. 2006, pp. 223–231.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to Sequence Learning with Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger. Vol. 27. Curran Associates, Inc., 2014.

[TH12] Tijmen Tieleman and Geoffrey Hinton. *Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude*. Lecture 4. COURSERA: Neural Networks for Machine Learning. 2012.

[Tom77] Peter Toma. "Systran as a multilingual machine translation system". In: *Proceedings of the Third European Congress on Information Systems and Networks, Overcoming the language barrier*. 1977, pp. 569–581.

[Tor+19] Daniel Torregrosa, Nivranshu Pasricha, Maraim Masoud, et al. "Leveraging Rule-Based Machine Translation Knowledge for Under-Resourced Neural Machine Translation Models". In: *Proceedings of Machine Translation Summit XVII: Translator, Project and User Tracks*. Ed. by Mikel Forcada, Andy Way, John Tinsley, et al. Dublin, Ireland: European Association for Machine Translation, Aug. 2019, pp. 125–133.

[Van79] C.J. Van Rijsbergen. *Information Retrieval*. Butterworths, 1979.

[Vas+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, et al. "Attention is all you need". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010.

[Wat23] S. Watanabe. "Tree-structured Parzen estimator: Understanding its algorithm components and their roles for better empirical performance". In: *arXiv preprint arXiv:2304.11127* (2023).

[Zei12] Matthew D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method". In: *CoRR* abs/1212.5701 (2012). arXiv: 1212.5701.

[Zha+19] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. "BERTScore: Evaluating Text Generation with BERT". In: *CoRR* abs/1904.09675 (2019). arXiv: 1904.09675.

[Zop+16] Barret Zoph, Deniz Yuret, Jonathan May, and Kevin Knight. "Transfer Learning for Low-Resource Neural Machine Translation". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Ed. by Jian Su, Kevin Duh, and Xavier Carreras. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 1568–1575.

List of Acronyms

List of Figures

List of Tables

List of Algorithms