POLITECNICO DI TORINO

MASTER's Degree in DATA SCIENCE AND ENGINEERING

MASTER's Degree Thesis

Beyond Cross-Entropy: Custom Loss Functions for Finetuning SLMs on Structured Recipe Generation

Supervisors	Candidate
Prof. Paolo GARZA	Mattia OTTOBORGO
Dott. Daniele REGE CAMBRIN	

DECEMBER 2025

Beyond Cross-Entropy: Custom Loss Functions for Finetuning SLMs on Structured Recipe Generation

Mattia Ottoborgo

Abstract

This thesis explores the use of custom loss functions to the finetuning of Small Language Models (SLMs) applied to recipe generation. With the exponential growth of deep learning, Large Language Models have proven themselves to have incredible text generation capabilities. Standard training frameworks, however, which employ Cross-Entropy loss, commonly disappoint in more demanding fields requiring high accuracy of facts and numbers, for instance the generation of procedural texts such as cooking recipes. This work tackles the intrinsic limitation of the standard solution that treats all the words indifferently, giving rise to the model's inability to appropriate the important but frequently statistically scarce ingredients of a recipe.

Construction of a valid recipe presents several challenges. It demands a model to generalize not only linguistic expertise but also procedural logic, recollection of facts about ingredients, and correct numerical reasoning for amounts, times, and temperatures. Incorrect generation of these crucial entities renders the output unusable and identifies a crucial disconnect between a model's textual coherence and its real-world practical utility. This paper contends that in order for this gap to be closed, the training objective itself needs to be modified to better accommodate the specific needs of the world. The thesis first provides an overview of the foundational concepts of modern NLP, from the Transformer architecture to the lifecycle and methodologies for finetuning language models, with a focus on Parameter-Efficient Finetuning (PEFT) via Low-Rank Adaptation (LoRA). It then details the design of a composite loss framework, augmenting the standard Cross-Entropy loss with one of three custom losses: Focal Loss, to address token imbalance; Dice Loss, to optimize for semantic overlap; and a novel Topological Loss, designed to measure the geometric similarity between the predicted and ground-truth ingredient lists in the embedding space.

Lastly, the thesis benchmarks the quality of several SLMs finetuned using such composite losses relative to a simple Cross-Entropy baseline. The models are assessed using a comprehensive set of evaluation metrics that includes typical NLP benchmarks and a series of ad-hoc measures designed to evaluate ingredient recall, numerical precision, and procedural correctness. The experiments document that training the objective using the domain-aware custom losses yields statistical gains that constitute a new and improved manner of finetuning language models for structured, fact-intensive generation tasks.

Table of Contents

1 **Introduction** 1

2 **Background** 3

2.1 Recipe 3

2.1.1 Recipes as Structured Knowledge 3

2.1.2 The Anatomy of a Recipe 4

2.1.3 Challenges in Recipe Generation 4

2.1.4 Evolution of NLP Tasks in the Recipe Domain 5

2.1.5 Conclusion: Why Recipe Generation is a Key Benchmark for Modern AI 6

2.2 Supervised Learning 6

2.2.1 Definition 6

2.2.2 Dataset and Goal 6

2.2.3 The Empirical Risk Minimization (ERM) Principle 7

2.2.4 The Learning Objective Function and Regularization 7

2.2.5 Core Components of the Supervised Learning Process 8

2.2.6 The Parametric Model: Large Language Models 8

2.2.7 The Loss Function: Sequence-Level Negative Log-Likelihood (NLL) 8

2.2.8 Optimization: Stochastic Gradient Descent and its Variants 9

2.2.9 Generalization and the Bias-Variance Tradeoff 10

2.2.10 Supervised Fine-Tuning (SFT) as the Foundation 11

2.3 NLP - Natural Language Processing 11

2.4 Generative AI 13

2.4.1 Generative vs. Discriminative Models 13

2.4.2 Architectural Landmarks 14

2.4.2.1 Early Deep Generative Models: VAEs and GANs 14

2.4.2.2 The Transformer's Role as a Universal Architecture 15

2.4.2.3 The GPT Family and Autoregressive Language Modeling 16

2.4.2.4 Diffusion Models and Stable Diffusion 16

2.4.3 Modalities of Generative AI 17

2.4.4 Applications and Impact 18

2.5 Transformer Architecture: principles and implementation 19

2.5.1 The Central Concept: Self-Attention 20
2.5.2 Formalizing the Attention Mechanism: Query, Key, and Value 20
2.5.3 Mathematics Formulation for Scaled Dot-Product Attention . 20
2.5.4 Multi-Head Attention: Parallel Attention 21
2.5.5 Positional Encoding: Reinserting Sequence Order 21
2.5.6 The Complete Transformer Block: Putting the Parts Together 21
2.5.7 The Original Architecture: Encoder and Decoder Stacks . . . 22
2.5.8 Architectural Variants and their Use Cases 22
2.5.9 Conclusion: Why the Transformer Dominates Modern NLP . 22
2.6 LLM - Large Language Models 23
2.6.1 What is an LLM? 23
2.6.2 From Statistical Models to Transformers 23
2.6.3 The Two-Stage Training Paradigm 24
2.6.4 Capabilities and Limitations 24
2.6.5 Taxonomy of LLMs 24
2.6.6 Conclusion 26
2.7 Small Language Models 27
2.7.1 Introduction: A New Paradigm of Efficiency and Specialization 27
2.7.2 What is a "Small" Language Model? 27
2.7.3 SLM Philosophy: Data Quality more than Data Volume . . . 27
2.7.4 Key Techniques for Creating High-Performing SLMs 27
2.7.5 The Central Trade-off: Generalist Giants or Specialist Experts? 28
2.7.6 The Benefits of Small Language Models 28
2.7.7 Limitation and Difficulty 29
2.7.8 SLMs' Ideal Use Cases 29
2.7.9 The Contribution of SLMs in This Research 29
2.7.10 Conclusion: The Future is Both Large and Small 30
2.8 Data Analysis Using Topological and Geometric Methods 31
2.8.1 Introduction: The Shape of Data, Beyond Traditional Statistics 31
2.8.2 From Data Points to Geometric Shapes: The Simplicial Complex 31
2.8.2.1 A Formal Definition 31
2.8.3 Homology: A Language for Describing Shape 31
2.8.3.1 The Algebra of Homology 32
2.8.4 The Scale Challenge and the Solution of Persistence 32
2.8.5 Persistent Homology in Detail 32
2.8.5.1 Birth, Death, and Persistence 32
2.8.5.2 The Persistence Diagram: Visualizing Topological Features 33
2.8.6 From Analyzing a Single Shape to Comparing Two: Optimal Transport 33
2.8.7 The Wasserstein Distance: Formalizing the Earth Mover's Distance 33
2.8.8 Sinkhorn Divergence: Approximating Optimal Transport . . 33

2.8.8.1 The Role of Entropic Regularization 33
2.8.8.2 The Final Formulation 34
2.8.9 The Connection with Topological Loss 34
2.8.10 Conclusion: A Bridge Between Topology and Deep Learning 34
2.9 Finetuning 34
2.9.1 What is Fine-Tuning? 34
2.9.2 Key Components of the Fine-Tuning Process 35
2.9.3 Challenges and Strategies 35
2.10 PEFT - Parameter Efficient Tuning 36
2.10.1 Introduction: The Efficiency Revolution 36
2.10.2 The Core Principle: Freezing the Giant, Training the Specialist 37
2.10.3 A Taxonomy of PEFT Methods 37
2.10.4 Additive Techniques: Injecting New, Trainable Modules . . . 37
2.10.5 A Deeper Look into Prompt-Based Tuning 38
2.10.6 Selective Methods: Finetuning a Small Subset of Existing Weights 38
2.10.7 Reparameterization Techniques: Another Change of Basis . . 39
2.10.8 Low-Rank Adaptation (LoRA) 39
2.10.9 Advantages of PEFT: Why It Is a Game-Changer 40
2.10.10 Disadvantages and Real-world Considerations 40
2.10.11 The Current PEFT Environment: Libraries and Tooling . . . 40
2.10.12 Conclusion: PEFT as the New Standard for Model Adaptation 40
2.11 LORA - Low-rank adaptation 41
2.11.1 Introduction: A Principled Approach to Parameter Efficiency 41
2.11.2 The Theoretical Foundation: The Intrinsic Rank Hypothesis 41
2.11.3 The Mechanics of Low-Rank Adaptation: Decomposing the Weight Update 42
2.11.4 Mathematical Formulation of the Forward Pass 42
2.11.5 The Trainable Parameters: A Quantitative Analysis 43
2.11.6 Key Hyperparameters: Rank (r) and Alpha (α) 43
2.11.7 Implementation in Transformer Models in Practice 44
2.11.8 Merging LoRA Weights for Zero-Latency Inference 44
2.11.9 Technical Advantages 44
2.11.10 Limitations and Theoretical Considerations 45
2.11.11 Conclusion: LoRA as an Optimal Trade-off 45
2.12 Quantization 45
2.12.1 The Model Deployment Problem 45
2.12.2 What is Quantization? The Principle of Lower Precision . . . 45
2.12.3 The Mathematics of Quantization: Affine Mapping 46
2.12.4 A Classification of Methods of Quantization 46
2.12.5 A Further Look into Post-Training Quantization (PTQ) . . . 47
2.12.6 Exploring Bit Depths: 8-bit to 4-bit and Beyond 47
2.12.7 The Core Benefits: Why Quantize a Model? 48
2.12.8 The Trade-off that is Unavoidable: Performance vs Efficiency 48

2.12.9 PEFT and Quantization: A Close Relationship 48
2.12.10 Conclusion: Quantization as a Standard for Deploying AI . . 49

3 Related Work 50
3.1 The Evolution of Recipe Generation as an NLG Task 50
3.1.1 Classical Sequence-to-Sequence Approaches and Early Baseline 50
3.1.2 Shifting Towards Transformer-Based LLMs and Adaptation . 51
3.1.3 Recipe Generation as Constructed and Constrained NLG . . 51
3.2 Culinary NLG Datasets and Benchmarks 52
3.2.1 Predecessor Datasets (Recipe1M+ and Visual Focus) 52
3.2.2 The RecipeNLG Dataset 52
3.2.3 Emerging Multimodal and Procedural Benchmarks 53
3.2.4 Evaluation Metrics for Structured Text Generation 54
3.2.5 Limitations of Classical N-gram 54
3.2.6 Domain-Specific Recipe Metrics 55
3.3 Novel Recipe Detection Based on Topological Data Analysis 55
3.3.1 Mapping the "Shape" of Culinary Space 56
3.3.2 Experimental Verification and Sensory Testing 56
3.3.3 A Comparison among the Topological Methods 57
3.3.3.1 Contrasting Goals and Methodologies 57
3.3.3.2 The Critical Difference: Representation and Differentiability 57
3.4 Why custom loss matters? 57
3.4.1 Standard SFT vs. Custom Loss Functions 57
3.4.2 Metric-Guided Training and RLHF 58
3.5 Training Objectives vs. Inference Constraints: Two Views Compared 58
3.5.1 Constrained Decoding Techniques 59
3.5.2 Trade-offs and Complementarity 59
3.6 Gap in Literature and Contribution 60

4 Methodology 62
4.1 Problem statement 62
4.2 Dataset preparation 63
4.2.1 Recipes 63
4.2.2 General Questions 64
4.2.3 Missing ingredient identification 64
4.2.4 Substitution validation 65
4.2.5 Recipe scaling 65
4.2.6 Quantity Ingredient 65
4.2.7 Time 66
4.2.8 Temperature 66
4.3 Dataset preprocessing 66
4.3.1 Preprocessing of Recipe dataset 66

4.3.2 Preprocessing of Cooking questions 67
4.3.3 Conversion to Huggingface dataset format 67
4.4 Losses 68
4.4.1 Loss functions 68
4.4.1.1 Cross-Entropy 68
4.4.1.2 Focal 69
4.4.1.3 Dice 69
4.4.1.4 Topological 70
4.4.1.5 Step 1: Generate Soft Probabilistic embeddings . . 71
4.4.1.6 Step 2: Point Cloud Creation 71
4.4.1.7 Step 3: Measuring Geometric Distance with Sinkhorn Divergence 72
4.5 Model selection 73
4.6 Experiments 74
4.7 Evaluation and metrics 76
4.7.1 Traditional metrics 77
4.7.1.1 BLEU (Bilingual Evaluation Understudy) 77
4.7.1.2 ROUGE (Recall-Oriented Understudy for Gisting Evaluation) 77
4.7.1.3 BERTScore 77
4.7.2 Ad-Hoc Metrics for Recipe Generation 77
4.7.2.1 Ingredient Recall 78
4.7.2.2 Quantity Precision 78
4.7.2.3 Action Precision 78
4.7.2.4 Time and Temperature Precision 78
4.7.2.5 Action Edit Distance 78
4.7.2.6 Step Edit Distance 78

5 Results 80
5.1 Traditional metrics 80
5.1.1 BLEU 80
5.1.2 ROUGE1 81
5.1.3 BERTSCORE 81
5.2 Ad-Hoc metrics 82
5.2.1 Quantity precision 83
5.2.2 Ingredient recall 83
5.2.3 Action and step edit distance 83
5.2.4 Temperature precision 85
5.2.5 Time precision 86
5.2.6 Combining Topological and Dice Losses 87
5.2.7 Statistical Significance of Results 88
5.2.7.1 Methodology: Welch's t-test 88
5.2.7.1.1 Topological Loss vs. Cross-Entropy Baseline 89

5.2.7.1.2 Topological Loss vs. Dice Loss 89
5.2.7.1.3 Mixed Loss vs. Dice Loss vs. Topological Loss 90

6 Conclusions **92**

Bibliography **94**

List of Figures

2.1 This figure illustrates a recipe for Carbonara as example of main information needed to process a recipe. 5
2.2 This figure illustrates the Empirical Risk Minimization Principle, where we select an optimal function $\hat{g}$ from a hypothesis class $\mathcal{H}$ that minimizes the empirical error on the training data, thereby approximating the true target function f. 7
2.3 Illustration of Stochastic Gradient Descent (SGD). The elliptical contours represent the loss function landscape, with the minimum at $\hat{\theta}$. The orange arrows depict the iterative path of SGD, demonstrating how it converges towards the optimal parameters by taking noisy steps in the direction of the negative gradient, computed using mini-batches or individual samples. 9
2.4 This figure illustrates the classic bias-variance trade-off in machine learning. As model complexity increases, the model's bias (its tendency to miss the true relationship) decreases while its variance (its sensitivity to fluctuations in the training data) increases. The goal is to find the optimum model complexity that minimizes the total error, balancing these two competing sources of error. 10
2.5 A comparison of generative and discriminative models. Generative models, shown on the left, learn the underlying distribution of each class to model how the data was generated. Discriminative models, on the right, learn a decision boundary that directly separates the classes, without modeling the data distribution itself. 14
2.6 A comparative illustration of Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs). The VAE (top) consists of an encoder that maps input data x to a latent space representation, and a decoder that reconstructs the input ($x' \sim x$) from this latent representation, trained with reconstruction and KL divergence losses. The GAN (bottom) comprises a generator that creates fake data $\tilde{x}$ from random noise, and a discriminator that learns to distinguish between this fake data and real data $\hat{x}$ from a dataset using an adversarial loss, leading to a generative model capable of producing realistic outputs. 15

2.7 Architectural overview of the Transformer model. The model consists of an encoder stack (left, orange) and a decoder stack (right, purple). Both stacks utilize multi-head attention mechanisms, feed-forward layers, residual connections, and layer normalization. The decoder incorporates an additional masked multi-head attention layer to prevent attending to future tokens and a multi-head cross-attention layer to attend to the encoder's output. Positional encodings are added to input and output embeddings to retain sequential information. The final decoder output passes through a linear layer and a softmax function to produce output probabilities, typically trained with Cross-Entropy loss for generative tasks. 16
2.8 The architecture of the GPT decoder-only model. The left side shows the overall structure, including input embeddings, positional encodings, a stack of L Transformer Blocks, and a final linear layer with softmax for probability output. The right side details a single Transformer Block, highlighting the masked multi-head attention and feed-forward network sub-layers, each with residual connections and pre-layer normalization. 17
2.9 The training framework for a Latent Diffusion Model (LDM). A pre-trained autoencoder (comprising encoder E and decoder D) maps an image X_0 into a compressed latent space, Z_0. The forward process adds noise to create Z_t. A U-Net is trained to predict the added noise, conditioned on Z_t and an encoded prompt (e.g., text processed by τ). The denoised latent Z_0^* is estimated by subtracting the predicted noise from Z_t, and the decoder D reconstructs the image X_0^*. 18
2.10 Overview of the Large Language Model (LLM) training pipeline, encompassing both pre-training and post-training stages. The process begins with a dataset that undergoes preprocessing, including filtering, synthetic data generation, and mixing. This preprocessed data then feeds into the pre-training phase, which involves techniques such as Q&A format training, long-context stages, continued pre-training, high-quality data stages, and knowledge distillation. Following pre-training, the model undergoes post-training, utilizing methods like supervised finetuning (SFT), reinforcement learning with human feedback (RLHF), direct preference optimization (DPO), and further knowledge distillation. The final stage involves additional optimization techniques. 25

2.11 A diagram illustrating the process of knowledge distillation. A larger, more complex "Teacher Model" is first trained on the data. The "Knowledge" from this teacher is then distilled and transferred to a smaller, more efficient "Student Model," which learns to mimic the teacher's behavior. This process allows the student model to achieve a performance level close to the teacher's, while being more lightweight and computationally less expensive. 28
2.12 A visual explanation of the process of using prompts to fine-tune a large language model. A pre-trained LLM is shown being adapted into a fine-tuned LLM using a dataset of prompt-completion pairs. Examples of these pairs for different tasks, such as summarization and translation, are provided to illustrate the instruction-following nature of the fine-tuning process. 35
2.13 An overview of various Parameter-Efficient Fine-Tuning (PEFT) methods for pre-trained language models. The diagram categorizes these methods into five main groups: Additive Fine-tuning (including adapter-based and soft prompt-based methods), Partial Fine-tuning (such as bias updating and delta weight masking), Re-parameterized Fine-tuning (like low-rank decomposition methods), Hybrid Fine-tuning (using manual combinations), and Unified Fine-tuning. 38
2.14 A comparison between prompt tuning and prompt design. In prompt tuning, the pre-trained model is frozen, and a tunable soft prompt is trained to generate the desired output. In contrast, prompt design also uses a frozen pre-trained model but relies on an engineered, fixed text prompt to guide the model's behavior. 39
2.15 Performance comparison between LoRA (with different ranks $r = 16, 64, 256$) and Full Fine-tuning for the `Llama-2-7B` model across various coding and math tasks. 41
2.16 An illustration of the Low-Rank Adaptation (LoRA) method for parameter-efficient fine-tuning. The process involves keeping the original pre-trained weights, represented by the matrix W, frozen. A low-rank matrix, decomposed into two smaller matrices A and B, is trained to adapt to a new task. The output is a combination of the original forward pass with the pre-trained weights and the low-rank update, represented as $h = Wx + B(Ax)$. This approach significantly reduces the number of trainable parameters. 43

2.17 Workflow comparison between Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT). PTQ involves quantizing a pre-trained FP32 model using a calibration dataset to produce a Quantized INT8 Model, offering a fast process but with potential accuracy loss. QAT, conversely, integrates fake quantization nodes during the training process with a full training dataset, leading to better accuracy preservation and finer control over trade-offs, albeit being more time-consuming and complex. 47
2.18 Comparison of finetuning methods: Full Finetuning, LoRA, and QLoRA. QLoRA leverages 4-bit quantization and paged optimizers for improved memory efficiency. 49

3.1 Evolution timeline of AI models in NLP domain 51
3.2 Samples of multimodal recipes from Recipe 1M 53
3.3 Feedback loop of RLHF finetuning loop. 58

4.1 The process of generating a Soft Probabilistic Embedding (emb_{soft}). The model's raw output logits are passed through a softmax function to create a probability distribution (P) over the vocabulary. This vector is then multiplied by the full embedding matrix (E). The resulting emb_{soft} is a weighted average of all token embeddings, yielding a continuous, fully differentiable representation of the model's prediction. 72
4.2 The creation of two comparative point clouds in the embedding space for ingredient tokens. The Ground-Truth Point Cloud ($\mathbf{PC_{gt}}$) is formed by hard embedding lookups, while the Predicted Point Cloud ($\mathbf{PC_{pred}}$) uses the Soft Probabilistic Embeddings. The topological loss minimizes the distance between these two clouds by forcing the soft prediction points (light gray) to align with the ground truth points (black). 72
4.3 Measuring the geometric distance between the Predicted ($\mathbf{PC_{pred}}$) and Ground-Truth ($\mathbf{PC_{gt}}$) point clouds using **Sinkhorn Divergence**. This metric calculates the regularized optimal transport cost (Earth Mover's Distance) required to move the predicted distribution (black points) onto the ground truth distribution (gray points), providing a differentiable loss signal for the model. 74
4.4 Illustration of the **Named Entity Recognition (NER)** process applied to extract structured attributes from a natural language instruction. After generating recipes in a simplified JSON format, a dedicated Small Language Model (Qwen3-4B) performs NER to extract granular details such as the main action, temperature, and time. This approach maintains a simple JSON schema for the primary output while still allowing for detailed attribute extraction, addressing challenges with model stability when embedding too many attributes directly into the initial JSON generation. 76

5.1 Performance comparison of the Mixed Model (Topological + Dice) against the individual Topological Loss and Dice Loss models across all evaluation metrics. 87

List of Tables

2.1 Comparison of RNN, LSTM, and Transformer Architectures 19
2.2 Comparison of features between a Base Large Language Model (LLM) and an Instruction-Tuned Large Language Model. This table highlights differences in their purpose, task specialization, response style, consistency, ability to handle complex tasks, adaptability to tone and format, alignment with user intent, and example applications. 26
2.3 Comparison of Small Language Models (SLMs) and Large Language Models (LLMs) 30

3.1 Key characteristics most common recipe datasets. 54
3.2 Key characteristics and trade-offs of three optimization paradigms for structured text generation. 60

5.1 Table of results for BLEU score 81
5.2 Table of results for ROUGE1 score 82
5.3 Table of results for BERTScore F1 score 82
5.4 Table of results for action precision 83
5.5 Table of results for quantity precision 84
5.6 Table of results for ingredient recall 84
5.7 Table of results for action edit distance (the smaller, the better) . . . 85
5.8 Table of results for step edit distance (the smaller, the better) 85
5.9 Table of results for temperature precision 86
5.10 Table of results for time precision 86
5.11 Comparison of the results of Mixed loss compared to Dice and Topological losses 88
5.12 Welch's t-test results comparing Qwen3-4B finetuned with Cross-Entropy vs. Topological Loss 89
5.13 Welch's t-test results comparing Qwen3-4B finetuned with Dice Loss vs. Topological Loss 90
5.14 Welch's t-test results comparing Mixed Loss (Dice + Topological) vs. pure Topological Loss and pure Dice Loss. 91

Acronyms

Adam	Adaptive Moment Estimation.
AI	Artificial Intelligence.
BERT	Bidirectional Encoder Representations from Transformers.
BLEU	Bilingual Evaluation Understudy.
CE	Cross-Entropy.
CFG	Context-Free Grammar.
CoT	Chain-of-Thought.
DPO	Direct Preference Optimization.
DSC	Sørensen-Dice Coefficient.
ERM	Empirical Risk Minimization.
FL	Focal Loss.
FP32	32-bit floating-point.
GAN	Generative Adversarial Network.
GCD	Grammar-Constrained Decoding.
GPT	Generative Pre-trained Transformer.
i.i.d.	independent and identically distributed.
INT4	4-bit integer.
INT8	8-bit integer.
IRL	Inverse Reinforcement Learning.
LLM	Large Language Model.
LoRA	Low-Rank Adaptation.
LSTM	Long Short-Term Memory.
MDP	Markov Decision Process.

MSE	Mean Squared Error.
NER	Named Entity Recognition.
NLG	Natural Language Generation.
NLL	Negative Log-Likelihood.
NLP	Natural Language Processing.
PEFT	Parameter-Efficient Finetuning.
PPO	Proximal Policy Optimization.
PRM	Process-Supervised Reward Model.
PTQ	Post-Training Quantization.
QAT	Quantization-Aware Training.
QLoRA	Quantized Low-Rank Adaptation.
RLHF	Reinforcement Learning from Human Feedback.
RM	Reward Model.
RNN	Recurrent Neural Network.
ROUGE	Recall-Oriented Understudy for Gisting Evaluation.
SFT	Supervised Fine-Tuning.
SGD	Stochastic Gradient Descent.
SLM	Small Language Model.
STG	Structured Text Generation.
TDA	Topological Data Analysis.
VAE	Variational Autoencoder.

Chapter 1

Introduction

In recent years, Large Language Models (LLMs) have proven to be a truly powerful technology, able to produce human-like text for a near-universal range of topics. Conventionally, the adaptation of these generalist models to a particular task depends upon a simple finetuning regime led by Cross-Entropy loss. This loss-oriented technique, where the goal is simply to predict the next term in a sequence, has proved very successful for tasks where linguistic fluency is the overriding criterion of success. But most real-world applications need something more than fluently written text; they need the text to be factually accurate, procedurally correct, and bound by a rigid underlying structure. A particular area that constitutes a novel and intriguing set of challenges is the generation of cooking recipes.

A recipe is not just creative writing. It is a procedural document in which certain entities—ingredients, their exact counts, cooking times, and temperatures—are functionally important. The typical Cross-Entropy model, however, makes the unrealistic assumption that all words are equally important. This causes it to fail to properly penalize a model for replacing a crucial ingredient or making a gibbering quantity because these typically statistically rare tokens have little impact on the total loss relative to typical grammatical words. This limitation points to a major gap: the typical training regime for LLMs isn't specially optimized for those domains where fact and numerical accuracy are crucial.

LLM application to recipe generation is especially useful because it acts as a strong test for a model's capabilities for performing structured, procedural, and fact-intensive work. A successful recipe generation involves a careful balance between a creative author and a accurate database. A model needs to create a cohesive tale of instructions but also needs the factuality of its ingredient list and the numeric correctness of all its related values. Typical failure modes like poor ingredient recall or impracticable procedural phases are not merely trivial defects; they cause the overall output to become unusable.

Given the enormous potential for LLMs to break the generation of content in this field, these challenges are significant. The most important challenge is the token skew in the training dataset combined with the inability of the standard loss function to bias towards the most critical ingredients of a recipe. Most existing recipe datasets are also structured for standard language tasks and lack the ad-hoc, reason-based queries (e.g., recipe scaling or ingredient substitutes) that are needed to train a model to reason about ingredients of a recipe explicitly. Sidelining the objective of merely designing a digital chef, the scope of this thesis lies in overcoming these inherent weaknesses in the LLM finetuning pipeline. More precisely, the objective lies in introducing and testing a new training paradigm that breaks the ceiling of the uniform-importance token assumption. We present a sequence of bespoke loss functions—Focal, Dice, and a new Topological loss—that are crafted to complement the regular Cross-Entropy loss. The auxiliary losses are designed to emphasize the learning signal for the most important and oft-neglected aspects of a recipe: its ingredients and numerical values. Finetuning occurs on a specially curated set of recipes and expert culinary questions using a family of lightweight Small Language Models (SLMs) and the Parameter-Efficient Finetuning (PEFT) paradigm of Low-Rank Adaptation (LoRA). The thesis also comprises four additional chapters. Chapter 2 gives a detailed background, delving into the basic principles of Natural Language Processing, the Transformer model architecture, and the life cycle of Large and Small Language Models, together with the theory underlying finetuning methods such as PEFT, LoRA, and quantization. Chapter 3 surveys the related work in the area of recipe generation, putting forth the history of models and datasets, and pinpointing the lacuna in the literature that the current work shall address. Chapter 4 covers the methodology, such as the problem statement, the preparation of the dataset, the mathematical expressions for the loss functions that we define ourselves, and the complete experiment setup. Chapter 5 gives the analysis and tabulation of the results of the experiments that we conduct by comparing the performance of each loss component by using a set of novel and standard evaluation metrics and concludes by discussing the insights and the direction for future work.

Chapter 2

Background

2.1 Recipe

On casual observation, a recipe is a modest document: a list of ingredients followed by some instructions. Again, appearances here deceive. A recipe is an interesting and intricate item of human knowledge, a confluence of procedural text, factual information, and cultural tradition. It is simultaneously a tale and scientific formula, creative guide and technical manual. Because it is so complicated, generating recipes is a surprisingly tricky and insightful test of artificial intelligence. Before we can create a model that can compose a good recipe, we need to understand and value the complex structure and subtlety of the domain of recipes as such. This chapter delves into the realm of recipes, dissecting their parts and explaining why training an AI to handle them is an important benchmark of contemporary generative models.

2.1.1 Recipes as Structured Knowledge

For centuries, recipes were the chief manner in which we encode and pass along culinary information. They remain manuals that help us reproduce a dish, instruments in teaching basic cooking methods, and cultural products that embody the history of a family or region. They are a type of communication that is meant for action, to be used in the material world in order to create some tangible, and we hope delicious, outcome.

Prior to processing or generating a recipe, such a human-oriented concept must be converted into a form of structured object. A model must also acknowledge a recipe as being more than a chunk of text, but instead as a collection of separate data fields with specific roles. In modern applications, a recipe is most typically stored in the form of a JSON such that it purposefully separates its key components. As such, the task in this paper instructs the model to generate a recipe with distinct keys of "ingredients" and "instructions". It is through such a structured form that the output becomes consistent and easy for a human as well as another computer program to handle.

2.1.2 The Anatomy of a Recipe

To make a successful recipe, you must learn to manage its fundamental building components. Each ingredient presents a new type of problem.

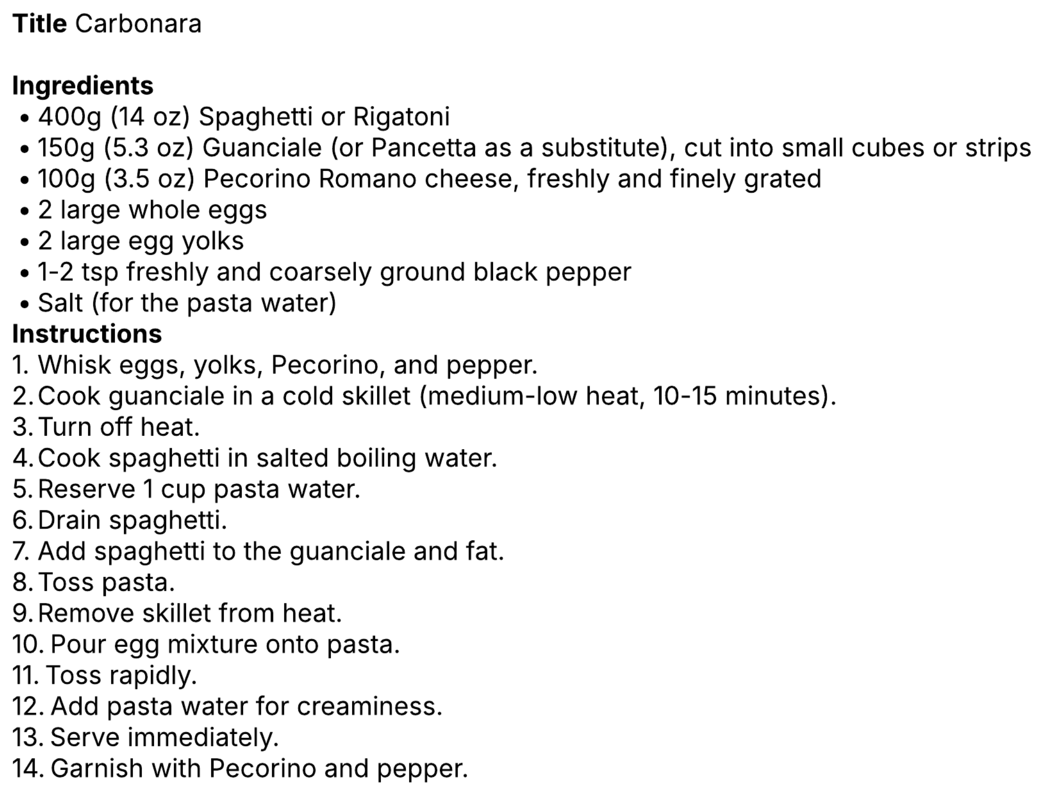
- **Title/Metadata**: It is also the primary key for the identification of the recipe (e.g., "Pasta Carbonara"). It may also contain metadata such as serving size, time of preparation, or cuisine type, all of them providing relevant information for the user.
- **Ingredients**: The list of ingredients is the factual nucleus of a recipe. It is not a suggestion but a precise bill of materials. An item in such a list is a well-formed piece of information, normally consisting of a quantity, a unit, and the ingredient's name, such as "200g Guanciale, cubed". The list is a list of unalterable entities that must be correct for the recipe to work. Its overriding importance is reflected in assessment criteria that examine the accuracy of quantities and recall of ingredients.
- **Instructions**: The instructions make up the procedural discourse of the recipe. The instructions comprise a chronologically sequenced set of steps, most commonly framed in imperative speech (e.g., "Boil salted water.", "Fry the guanciale."). The steps are more than words, being instituted in combination with other such important entities, such as precise times and temperatures for cooking, of whose precision a successful outcome depends.

An example of recipe can be found in Figure 2.1

2.1.3 Challenges in Recipe Generation

The precise nature of recipes makes them a difficult topic for Language models to learn.

- **Procedural and Sequential Dependency**: Operation order in a recipe is crucial. A model must learn that some steps precede some of the others. It must learn temporal reasoning and causality that is greater than is in typical language modeling.
- **High Density of Named Entities**: High density of named entities is present in the recipes (unit, ingredient, tool), which is statistically uncommon in ordinary text. It is then hard for a model to learn and recall with precision. As it turns out, low recall of ingredients is one of the typical failure modes of models.
- **The Grounding Problem**: The natural-language description of a recipe is "grounded" in the real world. The imperative "fry until crisp" is related to a chemical real-world process. In a model, these implicit physical constraints need to be learned but the model is never presented with a real-world kitchen.



Title Carbonara

Ingredients

- 400g (14 oz) Spaghetti or Rigatoni
- 150g (5.3 oz) Guanciale (or Pancetta as a substitute), cut into small cubes or strips
- 100g (3.5 oz) Pecorino Romano cheese, freshly and finely grated
- 2 large whole eggs
- 2 large egg yolks
- 1-2 tsp freshly and coarsely ground black pepper
- Salt (for the pasta water)

Instructions

1. Whisk eggs, yolks, Pecorino, and pepper.
2. Cook guanciale in a cold skillet (medium-low heat, 10-15 minutes).
3. Turn off heat.
4. Cook spaghetti in salted boiling water.
5. Reserve 1 cup pasta water.
6. Drain spaghetti.
7. Add spaghetti to the guanciale and fat.
8. Toss pasta.
9. Remove skillet from heat.
10. Pour egg mixture onto pasta.
11. Toss rapidly.
12. Add pasta water for creaminess.
13. Serve immediately.
14. Garnish with Pecorino and pepper.

Figure 2.1: This figure illustrates a recipe for Carbonara as example of main information needed to process a recipe.

- **Structural Enforcement**: Consistently producing a well-formed and correct structure, like constraining a JSON output, is a big technical obstacle.

2.1.4 Evolution of NLP Tasks in the Recipe Domain

The specification of problems in recipes has turned them into a highly popular topic for NLP/AI research, with the task progressing from basic analysis to advanced generation.

- **Recipe Analysis**: The initial applications of AI operated on the problem of processing known recipes. This was the problem of labeling a cuisine type of a recipe from ingredients or of using information extraction techniques to recover a structured list of ingredients from an unstructured blog posting.
- **Reasoning and Factual Manipulation**: Such more advanced tasks require a deeper, almost human-like comprehension of the internal workings of a specific recipe. These introduce a model's reasoning about relations between ingredients. The fine-tuning process in this task, for example, consisted of a dataset specifically crafted to train such abilities, including such issues as:
  - Missing ingredient identification
  - Commutativity verification
  - Recipe scaling

  - Specifying the amount, time, or temperature for an ingredient or procedure

- **Generation of Full Recipes: The Ultimate Challenge**: Full recipe composition is the holy grail because it involves generating all of the above components simultaneously. It needs to create a correct structure, reproduce factual entities accurately, form a coherent sequence of instructions, and express itself in a fluent, natural tone. It's a complete test of a model's capability as both a creative author and as a database of facts.

2.1.5 Conclusion: Why Recipe Generation is a Key Benchmark for Modern AI

As such, a recipe is much more than a mere text file. It is a highly complex object that simultaneously tries a generative model on multiple fronts. To successfully generate a recipe, a delicate balance of linguistic naturalness, factual correctness, procedural knowledge, and strictness in output format is required. As such, the domain is an ideal, but difficult benchmark for expanding the limits of contemporary AI, by finding the strengths and weaknesses of the models we build, as well as training methods we employ to control them.

2.2 Supervised Learning

Supervised learning is the base paradigm for a vast range of tasks in machine learning [1]. This section establishes the theoretical and practical framework of supervised learning, providing the necessary mathematical tools and conceptual understanding required to comprehend the subsequent chapters on customized optimization and loss functions.

2.2.1 Definition

Supervised learning is defined as the task of inferring a mapping function [2], or hypothesis, from labeled examples. The term "supervised" specifies that ground-truth labels are available to serve as the guide, or "supervisor," for the learning algorithm.

2.2.2 Dataset and Goal

We begin with a training dataset composed of N independent and identically distributed (i.i.d.) examples [3] extracted from an underlying, unknown data distribution $P(\mathcal{X}, \mathcal{Y})$:

$$\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$$

Here, $x^{(i)} \in \mathcal{X}$ is the input vector (e.g., a tokenized text sequence or prompt), and $y^{(i)} \in \mathcal{Y}$ is the corresponding desired output (e.g., a target sequence of tokens or a classification label). The goal is to find a function $h : \mathcal{X} \to \mathcal{Y}$ that minimizes the expected generalization error, or true risk [4], $R(h)$, which is the average loss over the entire data distribution P:

$$R(h) = \mathbb{E}_{(x,y)\sim P}[L(y, h(x))]$$

where $L(\cdot,\cdot)$ is the loss function measuring the penalty for the deviation of the prediction $h(x)$ from the true label y.

2.2.3 The Empirical Risk Minimization (ERM) Principle

Since the true data distribution P is unknown, we cannot directly minimize $R(h)$. Instead, supervised learning relies on the Empirical Risk Minimization (ERM) principle. ERM substitutes the true risk with the Empirical Risk $\hat{R}(h)$, which is the average loss calculated over the finite training set $\mathcal{D}$:

$$\hat{R}(h) = \frac{1}{N}\sum_{i=1}^{N} L(y^{(i)}, h(x^{(i)}))$$

The learning problem is thus transformed into finding the hypothesis h^* within a hypothesis space $\mathcal{H}$ that minimizes the empirical risk:

$$h^* = \arg\min_{h\in\mathcal{H}} \hat{R}(h)$$

The success of supervised learning hinges on the ability of the model that minimizes the empirical risk $\hat{R}(h)$ to also perform well on unseen data, meaning $\hat{R}(h) \approx R(h)$. This principle is visualized in Figure 2.2

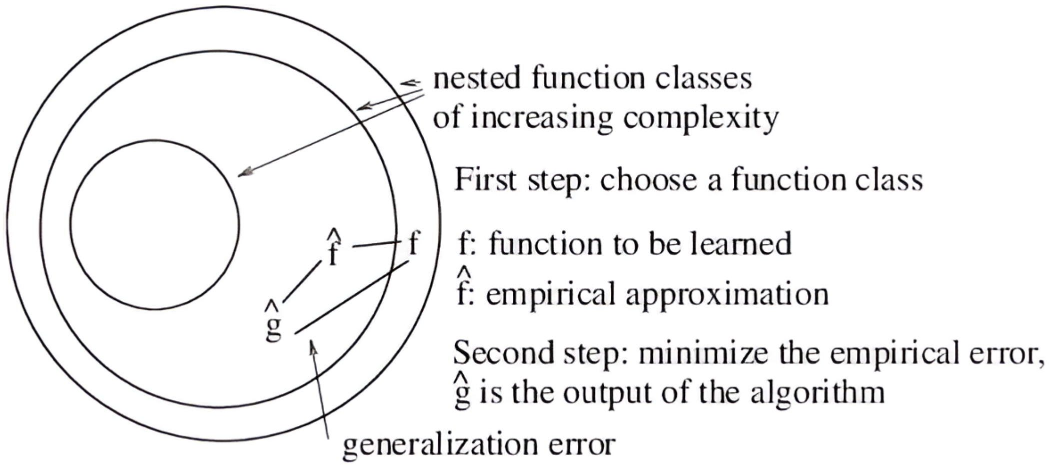

nested function classes
of increasing complexity
First step: choose a function class
$\hat{f}$
f
$\hat{g}$
f: function to be learned
$\hat{f}$: empirical approximation
Second step: minimize the empirical error,
$\hat{g}$ is the output of the algorithm
generalization error

Figure 2.2: This figure illustrates the Empirical Risk Minimization Principle, where we select an optimal function $\hat{g}$ from a hypothesis class $\mathcal{H}$ that minimizes the empirical error on the training data, thereby approximating the true target function f.

2.2.4 The Learning Objective Function and Regularization

In practice, the model h is a neural network parameterized by a vector of weights θ. Thus, we seek the optimal parameter set θ^*. The full Objective Function $J(\theta)$ that is minimized during training typically includes a Regularization term, designed to prevent overfitting and improve generalization:

$$J(\theta) = \hat{R}(\theta) + \lambda \cdot \Omega(\theta)$$

where $\hat{R}(\theta)$ is the empirical risk, $\Omega(\theta)$ is the regularization term (e.g., L_2 norm of the weights), and $\lambda \geq 0$ is the regularization coefficient, a hyperparameter controlling the strength of the penalty.

2.2.5 Core Components of the Supervised Learning Process

The process of training involves selecting a model architecture, defining the loss function, and utilizing an optimization algorithm to minimize the objective function $J(\theta)$.

2.2.6 The Parametric Model: Large Language Models

In the context of modern LLMs, the parametric model $h_\theta(x)$ is almost exclusively based on the Transformer architecture. This architecture, particularly its decoder-only variants, is designed for sequence-to-sequence or sequence completion tasks.

2.2.7 The Loss Function: Sequence-Level Negative Log-Likelihood (NLL)

While traditional supervised learning often uses Mean Squared Error (MSE) for regression or standard Cross-Entropy for single-label classification, LLM fine-tuning requires a loss function capable of handling sequences of discrete tokens. The standard choice is the Sequence-level Negative Log-Likelihood (NLL) [5], which is equivalent to the average Cross-Entropy loss over the target sequence. For a target sequence $y = (y_1, y_2, \ldots, y_T)$ of length T, the loss function $L(\cdot, \cdot)$ is defined as:

$$L_{NLL}(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \log P_\theta(y_t | x, y_{<t})$$

Where $P_\theta(y_t|x, y_{<t})$ is the probability assigned by the model h_θ to the correct target token y_t, conditioned on the input x and all previously generated tokens $y_{<t}$. This loss function promotes the maximum likelihood estimation of the target sequence [6]. By minimizing this loss, the model's parameters θ are adjusted to make the true sequence of tokens significantly more probable. The success of this standard approach has led to the development of custom loss functions, which seek to incorporate additional criteria (such as linguistic quality [7] , safety [8], or adherence to specific external metrics) beyond simple token match, forming the basis for the advanced work presented in this thesis.

2.2.8 Optimization: Stochastic Gradient Descent and its Variants

Optimization is the process of iteratively finding the parameter vector θ^* that minimizes the objective function $J(\theta)$. The foundation of nearly all deep learning optimization is Gradient Descent, which dictates that parameters are updated in the direction opposite to the gradient of the objective function:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta J(\theta^{(t)})$$

Here, η is the learning rate. For large datasets like those used in LLM fine-tuning, Stochastic Gradient Descent (SGD) is employed [9]. Instead of calculating the gradient over the entire dataset ($\nabla_\theta J(\theta)$), the gradient is approximated using a small subset of the data called a mini-batch $\mathcal{B} \subset \mathcal{D}$:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta \left(\frac{1}{|\mathcal{B}|} \sum_{(x^{(i)}, y^{(i)}) \in \mathcal{B}} L(y^{(i)}, h_\theta(x^{(i)})) \right)$$

This mini-batch approach introduces stochasticity, which helps models escape poor local minima and significantly speeds up training. Modern supervised fine-tuning heavily utilizes adaptive optimizers like Adam (Adaptive Moment Estimation) [10] or RMSprop [10], which dynamically adjust the learning rate η for each parameter based on the history of gradients, leading to faster and more stable convergence. The iterative path of SGD is depicted in Figure 2.3.

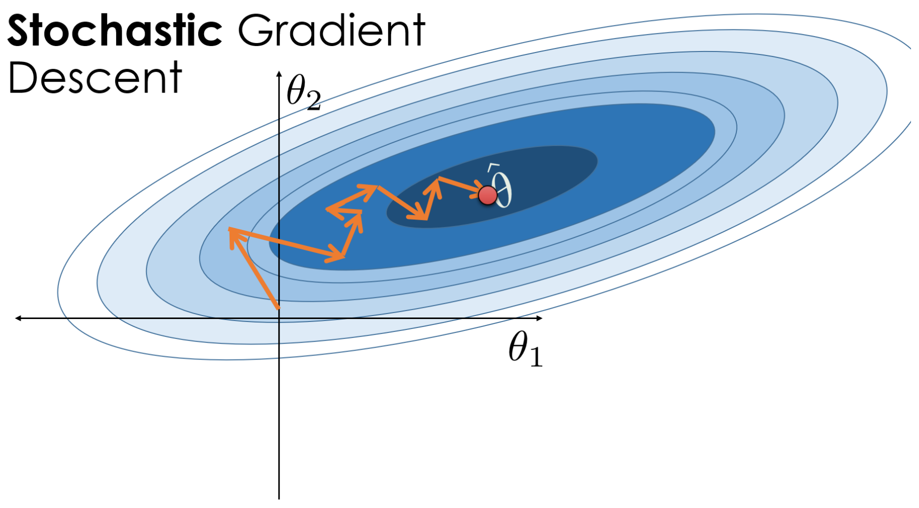


Figure 2.3: Illustration of Stochastic Gradient Descent (SGD). The elliptical contours represent the loss function landscape, with the minimum at $\hat{\theta}$. The orange arrows depict the iterative path of SGD, demonstrating how it converges towards the optimal parameters by taking noisy steps in the direction of the negative gradient, computed using mini-batches or individual samples.

2.2.9 Generalization and the Bias-Variance Tradeoff

A core challenge in supervised learning is ensuring generalization—the ability of the model to perform accurately on new, unseen data. This challenge is conceptualized through the Bias-Variance Tradeoff [11].

- **Bias**: Represents the error introduced by approximating a real-world problem with a simplified model. A high-bias model (e.g., an under-trained model or a model with insufficient capacity) often leads to underfitting, failing to capture the underlying patterns in the training data.
- **Variance**: Represents the error due to the model's excessive sensitivity to small fluctuations in the training set. A high-variance model (e.g., an over-trained model) often leads to overfitting, performing exceptionally well on the training data but poorly on the test data.

The objective in SFT is to strike a balance: achieving low empirical risk $\hat{R}(\theta)$ while maintaining the capacity for generalization, ensuring that the true risk $R(\theta)$ remains low. This balance is often managed through hyperparameters such as regularization strength λ, the choice of optimizer, and early stopping criteria [12]. This relationship is illustrated in Figure 2.4.

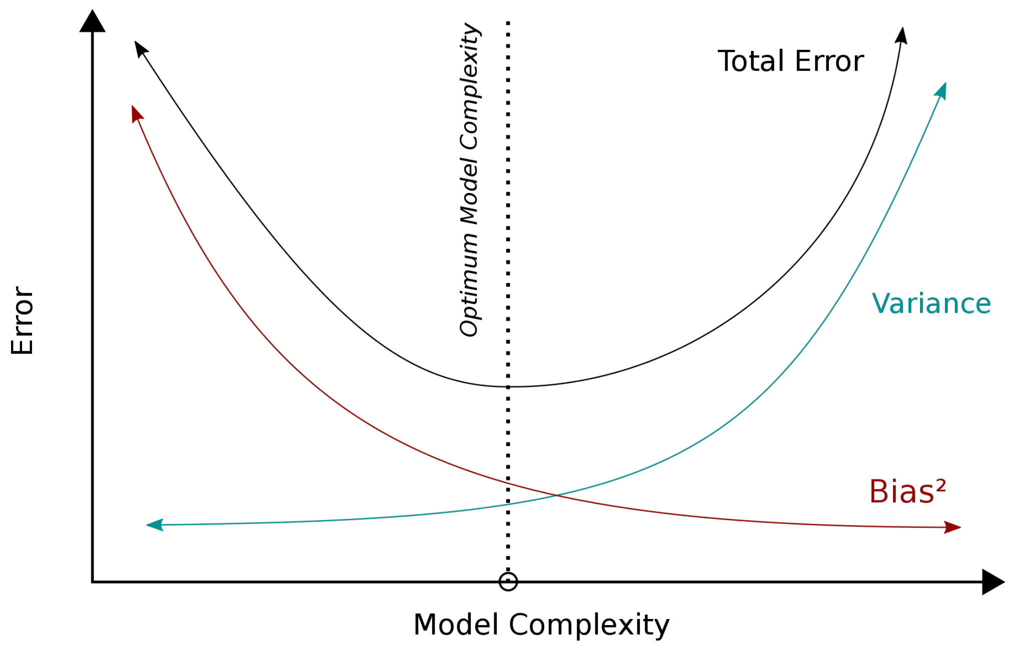


Figure 2.4: This figure illustrates the classic bias-variance trade-off in machine learning. As model complexity increases, the model's bias (its tendency to miss the true relationship) decreases while its variance (its sensitivity to fluctuations in the training data) increases. The goal is to find the optimum model complexity that minimizes the total error, balancing these two competing sources of error.

2.2.10 Supervised Fine-Tuning (SFT) as the Foundation

Supervised Fine-Tuning (which will be better introduced in the next chapters) involves leveraging the pre-trained knowledge θ_0 and further minimizing the empirical risk $\hat{R}(\theta)$ calculated via the sequence NLL loss over a specific task dataset. NLL/Cross-Entropy is mathematically convenient for likelihood maximization; However, it does not necessarily align with complex human preference metrics (like helpfulness, truthfulness, coherence) which are inherently non-differentiable. The limitations of the standard NLL loss in capturing nuanced objectives motivate the exploration of custom loss functions and advanced techniques introduced in the subsequent chapters to better optimize LLMs in the context of recipe generation.

2.3 NLP - Natural Language Processing

For decades, teaching a computer to truly understand human language felt like a distant goal. This is the challenge at the heart of Natural Language Processing (NLP), a field of artificial intelligence focused on bridging the between our fluid, contextual language and the rigid, numerical world of computers. Early attempts tried to solve this problem by hand-crafting vast encyclopedias of grammatical rules [13]. However, these systems were fragile; they would break when faced with slang, a simple typo, or the endless creativity of human expression.

The real breakthrough came with a change of perspective: instead of explicitly teaching computers the rules of language, we let them learn the patterns themselves from massive amounts of text data [14]. This data-driven approach is the foundation of modern NLP and the deep learning models that reached human-level performances, from translating languages in real-time to writing poetry. This chapter will walk through the key innovations that made this possible, building up the concepts needed to understand the core work of this thesis.

The first problem we encounter is the incapability for NLP models to understand human language since they can only process numerical values. To get a model to process a recipe, we first need a clever way to translate our words into its native language of numbers. This translation happens in two main steps: tokenization and embedding.

Before we can do any complex math, we have to divide our input text into manageable pieces. This process, called tokenization, breaks down a sentence into a list of "tokens." While a token can be a whole word, modern models often use a smarter approach called sub-word tokenization [15]. This allows the model to break down unfamiliar words into smaller, known pieces. For example, the word "undercooked" might become "under" and "cooked." This is an elegant solution that helps the model handle typos and rare words without needing an infinitely large dictionary.

With our text now a sequence of tokens, we need to convert each one into a vector of numbers. An early approach was one-hot encoding, which created a huge vector for each token with a single '1' and zeros everywhere else. The problem was that these vectors were enormous and, even worse, not conveying any semantic meaning. The vector for "cheese" was no more related to "milk" than it was to "rocket," making it impossible for a model to learn semantic relationships.

What really revolutionized NLP was the creation of token embeddings [16]. Instead of a sparse, giant vector, an embedding represents a token as a much shorter, dense vector of meaningful numbers. The real magic is that these vectors are not fixed; the model learns them during training. By analyzing how words are used across billions of sentences, the model adjusts these vectors.

This process organizes words into a sort of high-dimensional map, where tokens with similar meanings become close neighbors. In this space, "pecorino" and "parmesan" would cluster together, while "boil" and "fry" would also be nearby. This direct link between a word's meaning and its location in a geometric space is the fundamental concept that powers our topological loss (which will be better introduced in the next chapters). It allows us to mathematically measure the "distance" between two lists of ingredients and have that distance mean something real.

For years, NLP models processed text sequentially, like reading a sentence one word at a time. This struggled with long-range context. The game-changing architecture that solved this was the Transformer. While we will dedicate an entire chapter to its design, its potential lies in a mechanism called self-attention. In essence, self-attention gives the model the ability to look at all the words in a sentence at once and decide which ones are most important for understanding any given word. The incredible power and scalability of this design directly paved the way for the Large Language Models that are central to this thesis.

How does a model actually learn? It needs a teacher, or at least a critic, to tell it when it's wrong. This critic is the loss function. After the model makes a prediction, the loss function calculates a score that represents how big the error was. The entire goal of training is for the model to slowly adjust its internal parameters to make this error score as low as possible.

The most common loss function for training models to generate text is Cross-Entropy [17]. Its job is to look at the probability the model assigned to the correct next word. If the correct word was "cheese" and the model thought there was a 90% chance it was "cheese," the loss is very low. If the model only gave it a 1% chance, the loss is very high. Cross-Entropy assumes that all tokens have similar importance and frequency in texts and this is a significant issue because natural language famously follows a Zipfian distribution, or Zipf's Law [18]. This law states that a few tokens (like 'the', 'is', 'in') appear with extremely high frequency, while the vast majority of tokens (like 'pecorino' or 'guanciale') are exceptionally rare.

The catch is that Cross-Entropy, when trained on such an imbalanced distribution, it is incentivized to get the "easy" high-frequency words right, while treating a mistake on the common word "the" with the same severity as a mistake on the rare, but crucial, ingredient "pecorino." This limitation is the primary motivation for exploring the custom loss functions in this work.

Once the model is trained, we need a "judge" to evaluate its performance. To do this, we use a set of standard metrics. **BLEU** [19] acts like a strict grammarian, checking if the phrases in the generated text match a reference. **ROUGE** [20]is more of a fact-checker, seeing if the key individual words are present. Finally, **BERTScore** [21] is the semantic evaluator, using its own deep understanding of language to determine if the generated text means the same thing as the reference, even if the wording is different.

2.4 Generative AI

For most of its history, artificial intelligence has been a tool for analysis. We have trained models to find patterns, classify data, and make predictions—basically to act as a brilliant critic or a tireless judge. But in recent years, AI has made a remarkable progress in creative capabilities. It has learned not only to analyze the world but also to contribute to it with novel creations. This new frontier is the domain of Generative AI, a branch of artificial intelligence focused on creating new, original content that is indistinguishable from, and sometimes even surpasses, human-made creations.

From composing music in the style of Bach to designing novel proteins that have never existed in nature, Generative AI represents a shift in mentality from AI as an analyst to AI as a creator. This chapter will explore the fundamental principles that define this exciting field and it will guide through the key architectures that brought it to life, and survey its transformative impact across industries. Understanding this broader landscape is essential for the more focused explanation of Large Language Models that will follow.

2.4.1 Generative vs. Discriminative Models

In order to understand what makes Generative AI so special, it is useful to compare it with its more traditional counterpart: discriminative AI. This distinction lies at the very heart of what these models are trained to do.

Think of a discriminative model as a judge. Its primary goal is to learn the boundary that separates different categories of data. Given an input, it makes a decision or a prediction. A classic example is an email spam filter; its job is to look at an email and decide which of two boxes it belongs in: "spam" or "not spam." It learns the characteristics that differentiate the two but has no idea how to write a spam email itself. In more formal terms, it learns the probability of a label given an input, or $P(y|x)$.

A generative model, on the other hand, is an artist. Instead of learning to separate data, it learns the underlying patterns and structure of the data itself. Its goal is to understand the data so deeply that it can generate entirely new samples that look like they could have come from the original dataset. It doesn't just learn to recognize a cat; it learns the essence of "cat-ness" so well that it can create a picture of a cat that has never existed. Formally, it learns the distribution of the data itself, $P(x)$, allowing it to create new data points from scratch. A comparison of these two model types is illustrated in Figure 2.5.

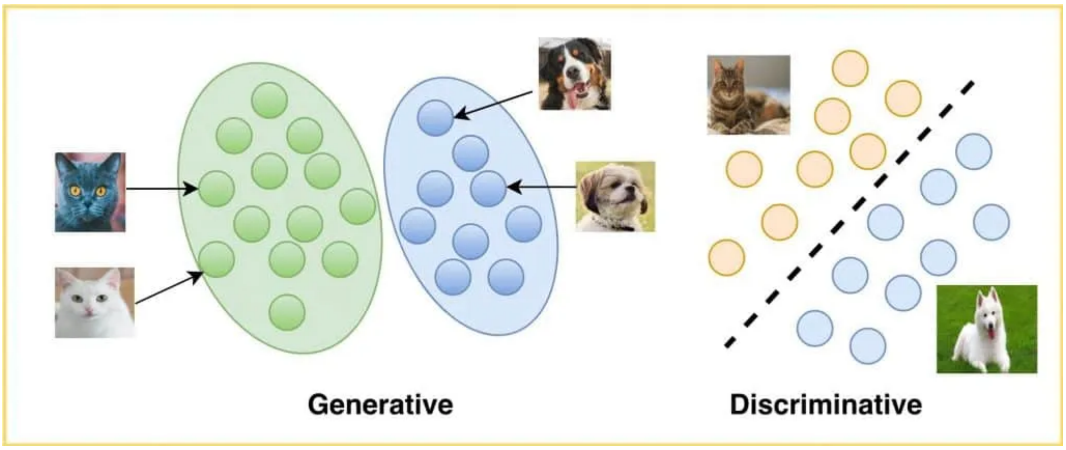


Figure 2.5: A comparison of generative and discriminative models. Generative models, shown on the left, learn the underlying distribution of each class to model how the data was generated. Discriminative models, on the right, learn a decision boundary that directly separates the classes, without modeling the data distribution itself.

2.4.2 Architectural Landmarks

Reaching nowadays state-of-the-art performance has been only possible thanks to a series of brilliant architectural innovations, each solving a crucial aspect of Generative AI tasks.

2.4.2.1 Early Deep Generative Models: VAEs and GANs

In the early days of the deep learning development, two architectures stood out. Variational Autoencoders (VAEs) [22] took an elegant approach: they learned to compress data into a simplified, latent representation and then learned to reconstruct the original data from this compressed form. By sampling from this latent space, they could generate new data. VAEs are typically trained by minimizing a combination of a *reconstruction loss* (ensuring the output resembles the input) and a *KL divergence loss* (ensuring the latent space is well-structured). VAEs are known for producing diverse and creative outputs, though they sometimes suffer from a tendency to generate slightly blurry or fuzzy results.

A few years later, a groundbreaking idea emerged: Generative Adversarial Networks (GANs) [23]. GANs introduced a clever cat-and-mouse game between two neural networks. The first, the Generator, acts like an art forger, trying to create fake images that look completely real. The second, the Discriminator, acts as an art critic, trying to tell the difference between the Generator's fakes and real images from the training data. These two networks train together using an *adversarial loss*, where the Generator tries to minimize the Discriminator's ability to spot fakes, while the

Discriminator tries to maximize it. This adversarial process proved to be incredibly effective, leading to the creation of hyper-realistic images and pushing the boundaries of what people thought AI could do. Figure 2.6 provides a comparative illustration of these two architectures.

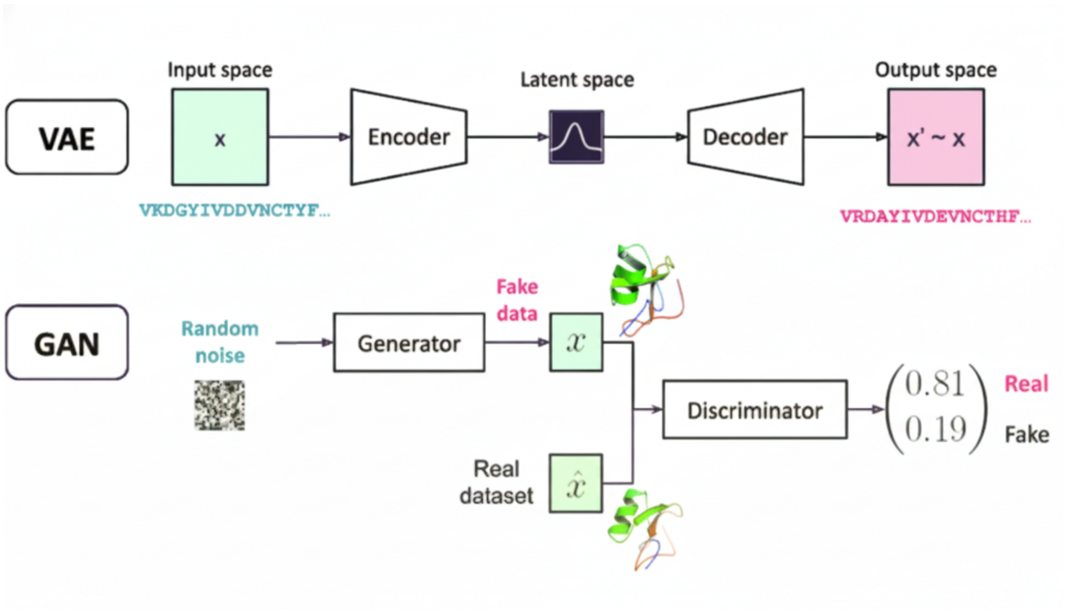


Figure 2.6: A comparative illustration of Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs). The VAE (top) consists of an encoder that maps input data x to a latent space representation, and a decoder that reconstructs the input ($x' \sim x$) from this latent representation, trained with reconstruction and KL divergence losses. The GAN (bottom) comprises a generator that creates fake data $\tilde{x}$ from random noise, and a discriminator that learns to distinguish between this fake data and real data $\hat{x}$ from a dataset using an adversarial loss, leading to a generative model capable of producing realistic outputs.

2.4.2.2 The Transformer's Role as a Universal Architecture

While VAEs and GANs were powerful, the Transformer architecture, first developed for language translation, completely reshaped the generative landscape. Its core innovation, the self-attention mechanism, turned out to be a remarkably versatile tool for finding patterns in almost any kind of data, not just text. Researchers discovered that this ability to address the importance of different data points in a sequence could be applied to pixels in an image, notes in a piece of music, or lines in a computer program. When applied to generative tasks, especially in the autoregressive setting (predicting the next element in a sequence), Transformers are typically trained using a *Cross-Entropy loss* objective. This discovery began to unify the field, establishing the Transformer as a universal blueprint for building powerful generative models across nearly every domain. The complete Transformer architecture is detailed in Figure 2.7.

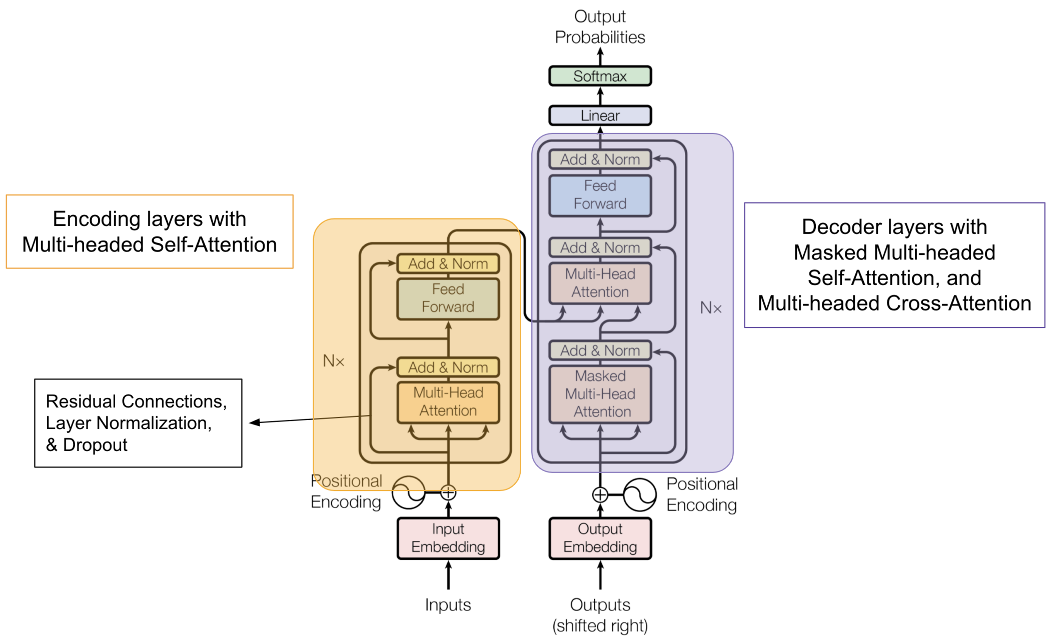


Figure 2.7: Architectural overview of the Transformer model. The model consists of an encoder stack (left, orange) and a decoder stack (right, purple). Both stacks utilize multi-head attention mechanisms, feed-forward layers, residual connections, and layer normalization. The decoder incorporates an additional masked multi-head attention layer to prevent attending to future tokens and a multi-head cross-attention layer to attend to the encoder's output. Positional encodings are added to input and output embeddings to retain sequential information. The final decoder output passes through a linear layer and a softmax function to produce output probabilities, typically trained with Cross-Entropy loss for generative tasks.

2.4.2.3 The GPT Family and Autoregressive Language Modeling

Perhaps the most prominent examples of generative Transformer models are those in the GPT (Generative Pre-trained Transformer) family, developed by OpenAI [**gpt**]. These models, including GPT-2, GPT-3, and subsequent iterations (often referred to generically as GPT-X), are typically decoder-only Transformers. They are pre-trained on vast amounts of text data with the simple objective of predicting the next word (or token) in a sequence. This autoregressive process, optimized using a standard *Cross-Entropy loss* (or Negative Log-Likelihood), allows the models to generate remarkably fluent and coherent text across a wide range of styles and topics. Their success demonstrated the power of scale and established the foundation for modern Large Language Models. Figure 2.8 shows the architecture of the GPT family.

2.4.2.4 Diffusion Models and Stable Diffusion

A more recent, yet incredibly powerful, class of generative architectures is Diffusion Models. These models have shown state-of-the-art results, particularly in image generation. The core idea is inspired by thermodynamics: a forward process gradually adds noise to an image until it becomes pure static, and a reverse process learns to

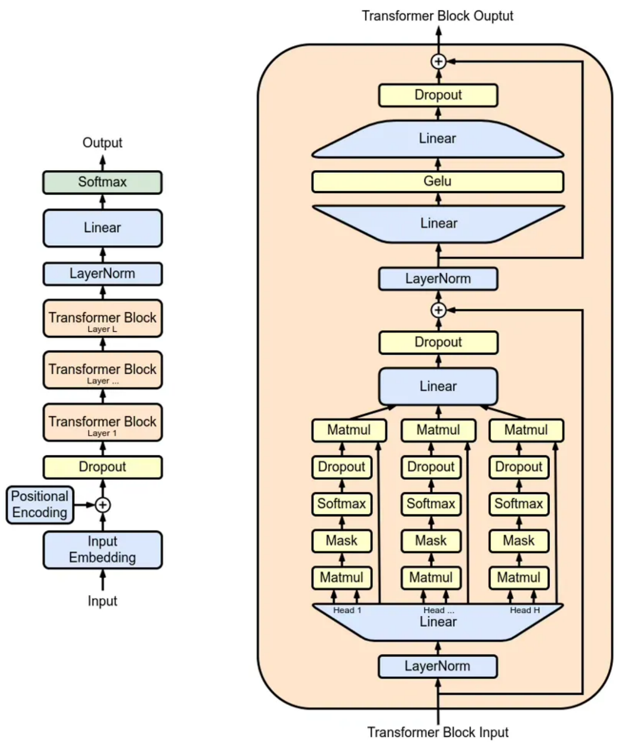


Figure 2.8: The architecture of the GPT decoder-only model. The left side shows the overall structure, including input embeddings, positional encodings, a stack of L Transformer Blocks, and a final linear layer with softmax for probability output. The right side details a single Transformer Block, highlighting the masked multi-head attention and feed-forward network sub-layers, each with residual connections and pre-layer normalization.

iteratively remove the noise, starting from static, to generate a clean image. Models like Stable Diffusion are prominent examples of Latent Diffusion Models, which perform the diffusion process in a compressed latent space for greater efficiency. The training objective for these models typically involves predicting the noise that was added at each step, often using a *Mean Squared Error (MSE) loss* between the predicted noise and the actual noise. To guide the generation process towards specific concepts (like generating an image based on a text prompt), these models are often conditioned using embeddings from other models like CLIP, which connects text and images. Figure 2.9 visually represent the training procedure for Stable diffusion models.

2.4.3 Modalities of Generative AI

The versatility of modern architectures means that Generative AI is not limited to a single type of content. Its capabilities span a wide range of human creativity and communication:

- **Text Generation**: This domain is represented by use cases involving chatbots, automated summarizers, creative writing partners, and, as explored in this thesis, the generation of recipes.

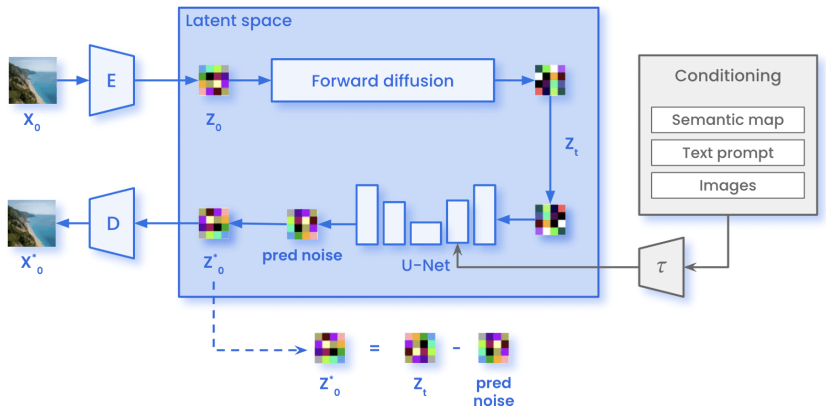


Figure 2.9: The training framework for a Latent Diffusion Model (LDM). A pre-trained autoencoder (comprising encoder E and decoder D) maps an image X_0 into a compressed latent space, Z_0. The forward process adds noise to create Z_t. A U-Net is trained to predict the added noise, conditioned on Z_t and an encoded prompt (e.g., text processed by τ). The denoised latent Z_0^* is estimated by subtracting the predicted noise from Z_t, and the decoder D reconstructs the image X_0^*.

- **Image and Video Generation**: This includes text-to-image models that can turn a simple prompt like "an astronaut riding a horse on Mars" into a stunning picture, as well as emerging models that can generate short video clips from text.
- **Audio and Music Synthesis**: This technology can clone a person's voice from a short sample, compose original music in any genre, or generate realistic sound effects for films and games.
- **Code Generation**: This domain consists of writing, suggesting and editing snippets of code in order to accomplish software engineering tasks, improving software engineers' productivity.

2.4.4 Applications and Impact

The ability to generate novel content has unlocked applications that were once unthinkable. In scientific research, Generative AI is being used to design new drugs and discover stable proteins which accelerated the pace of discovery. In the creative industries, it is used as a powerful co-pilot for artists, designers, and writers, helping them brainstorm ideas and produce content more efficiently. In software engineering, it is boosting developer productivity and lowering the barrier to entry for new programmers. The impact of this technology is already profound and continues to grow at an incredible rate.

Generative AI represents therefore a milestone in the capabilities of artificial intelligence, moving from analysis to creation. It is defined by its ability to learn the underlying patterns of data and generate new samples, a task made possible by a series of architectural breakthroughs from VAEs and GANs to the now-dominant Transformer. The latter is known as one of the key components of models known as Large Language Models (LLMs). The next chapter will dive deep into the world of LLMs, exploring what makes them so powerful and how they can be adapted for specialized tasks like the one at the core of this research. A comprehensive comparison of RNN, LSTM and Transformers can be found in table 2.1

Feature	**RNN**	**LSTM**	**Transformer**
Sequence Processing	Sequential, one element at a time via hidden state.	Sequential, like RNNs, but with internal gates.	Parallel, processes entire sequence via self-attention.
Handling Long-Range Dependencies	Poor due to vanishing/exploding gradients.	Improved over RNNs via gating mechanism controlling information flow.	Excellent due to self-attention enabling direct connections between any elements.
Parallelization	Difficult due to sequential computation dependency.	Difficult due to sequential computation dependency.	Highly Parallelizable due to independent computations within self-attention.
Key Innovation	Recurrence and hidden states for sequential data.	Gating mechanisms (input, forget, output) for improved memory.	Self-attention mechanism, eliminating recurrence for parallel processing.
Primary Limitation Addressed	N/A (Baseline sequential model).	Addresses RNN's vanishing gradients and poor long-range memory.	Addresses RNN/LSTM's sequential bottleneck (limits parallelization) and long-dependency issues.

Table 2.1: Comparison of RNN, LSTM, and Transformer Architectures

2.5 Transformer Architecture: principles and implementation

For years, the most popular style for processing language took inspiration from the way we human beings do it: sequentially. Models like Recurrent Neural Networks (RNNs) [24] and their later-to-be-more-popular LSTMs [25], would process a sentence word by word while maintaining a hidden "memory" of the prior things they'd processed. Intuitive as this approach is, it had a major bottleneck. The model's understanding of the first word's meaning had to be propagated all the way down a long chain before it could influence its understanding of the last word, often getting diluted along the way. This troubled them when they tried to process long-range dependencies in text. Also being models that were sequential in nature, they were difficult to parallelize, so training them using enormous datasets was a long and cumbersome process. The community required a new idea, and in 2017 an innovative paper titled "Attention Is All You Need" [26] provided it to them: the Transformer.

2.5.1 The Central Concept: Self-Attention

The Transformer's revolutionary concept is self-attention. Rather than processing a sentence word-by-word, the self-attention mechanism lets the model consider all the words of the input sequence as a whole. For each of the words it's processing, it's able to calculate a "attention" or "relevance" score for every one of the other words in the sequence in real time. This lets it borrow context from anywhere in the text without penalty for remoteness. For the sentence "The robot picked up the ball because it was heavy," self-attention lets the model know that "it" is the "ball" and not the "robot" by learning to attend disproportionately to the correct nouns.

2.5.2 Formalizing the Attention Mechanism: Query, Key, and Value

To bring this idea into practice, the attention mechanism follows a powerful analogy from the information retrieval literature that provides three vectors for every input token:

- **Question (Q)**: This vector is the current word that is querying for information. It is akin to a query asking the question, "What are the most relevant words in this sentence to my meaning?"
- **Key (K)**: This vector may be treated like a tag for every word in the sequence. It previews the type of information a word represents by answering a question by saying, "This is the type of context I am able to offer."
- **Value (V)**: This vector holds the real content or substance of a word. If the Key of a word goes well with a Query, its Value is actually what is propagated to offer context.

2.5.3 Mathematics Formulation for Scaled Dot-Product Attention

Computation of attention from these three vectors is mathematically valid and very elegant. It involves four primary steps:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

1. **Calculate Scores**: The model computes the similarity between the Key vector of the current word and the Query vector of all the words in the sequence. It is accomplished by performing the dot-product operation between the matrices (KQ^T). A high score means a strong relevance.
2. **Scale**: For stabilizing the gradients while training the model, the scores are scaled by normalizing the scores by the square root of the key vector's dimension $(\sqrt{d_k})$.
3. **Normalize**: We take the scaled scores and apply a softmax function to them. This normalizes them to a set of positive numbers that add to 1, which we

can view as the "attention weights." These are exactly the weights that denote exactly how much attention the current word must pay to each other word.

4. **Weighted Sum**: We weigh these attention weights by the Value vectors of individual words. Words that have very high attention weights contribute a lot of their meaning to the final representation of the current word and hence infuse it with rich relevant context.

2.5.4 Multi-Head Attention: Parallel Attention

One attention mechanism will potentially learn to attend to a single relationship type (e.g., subject-verb relationships). To create a stronger model, the Transformer uses Multi-Head Attention. This is done by running the scaled dot-product attention mechanism several times in parallel. Each of these runs in parallel is a "head." Before the process gets underway, the original Query, Key, and Value vectors get divided into smaller pieces and each head runs on a distinct piece. That way, each of the heads will potentially learn distinct relationship types of context all simultaneously. It's like having a panel of experts read the same sentence where each expert studies a distinct angle—one the grammar, another the semantics, a third the long-range dependencies. The output of all the experts is then aggregated together to produce a stronger and subtler final result.

2.5.5 Positional Encoding: Reinserting Sequence Order

A great side benefit of the self-attention mechanism is that it is order-agnostic by itself. It views a sentence as a "bag of words" and has no inherent sequence. To cure this problem, the Transformer incorporates information about the particular position of each token into the input. This is accomplished through the use of positional encodings, vectors that represent a token's particular position in the sequence. The positional vectors are added to the token embeddings very early in the process so that the model will notice the difference between "the dog chased the cat" and "the cat chased the dog."

2.5.6 The Complete Transformer Block: Putting the Parts Together

A Transformer model consists of lining up a number of identical blocks. Each of these blocks has two major sub-layers:

- **The Feed-Forward Network**: Following the Multi-Head Attention sub-layer having computed the inputs and accumulated context, the result is fed through a plain, fully connected feed-forward network. This network affords further computational depth and non-linear transformation for the individual token's representation.
- **Residual Connections and Layer Normalization**: Both of the sub-layers (attention and feed-forward) are encapsulated by two important parts: a

residual connection and layer normalization. The residual connection (or skip connection) combines the sub-layer's input with its output to avoid the vanishing of gradients in very deep networks. Layer normalization in turn re-scales the output to a standard distribution. Both of these components together are important for the stable training of very deep Transformer models.

2.5.7 The Original Architecture: Encoder and Decoder Stacks

The simplest Transformer model was designed for the task of machine translation and consisted of two sets of blocks:

- **The Encoder's Role**: The encoder's job is to read and understand the input sentence (e.g., in English). It's a stack of the encoder blocks that operate over the entire input sequence in parallel and construct a rich contextual knowledge of it.
- **The Decoder's Role and Masked Self-Attention**: The decoder's task is to produce the output sentence (e.g., the French sentence) token by token. It has self-attention but it incorporates a very crucial concept named Masked Self-Attention. This mask prevents the decoder from "cheating" by peeking ahead of the sequence it is trying to predict. For instance, when it's trying to predict the fourth word of the sentence in the output, the mask will see that the model will be able to attend to nothing but the first, second, and third words. This is a necessity for autoregressive generation tasks.

2.5.8 Architectural Variants and their Use Cases

Since the initial paper, the community has further extended the Transformer architecture to a few families:

- **Encoder-only Models** (e.g., BERT [27]): Encoder-only models employ the encoder stack and are the champions of understanding language. They are suitable for sentiment analysis, text classification, and question answering.
- **Decoder-only Models** (e.g., GPT [28], Qwen, SmolLM): These models are composed solely of the decoder stack and are text generation masters. Repeatedly anticipating the next token, they will write essays, generate code, and create recipes. Models utilized by this thesis fall under this category.

2.5.9 Conclusion: Why the Transformer Dominates Modern NLP

The Transformer's architecture overcame the major challenges that restrained its forerunners. Its design is very parallelizable such that it could be trained upon enormous datasets using current hardware. Its self-attention mechanism is surprisingly adept at discovering long-range complicated relationships in a piece of text. This efficiency and potency are why the Transformer has become the undisputed base for almost all the state-of-the-art language models that have come into being until the time of writing this thesis.

2.6 LLM - Large Language Models

The concepts of Generative AI and the Transformer architecture, described in the previous chapters, come together to create today's most impactful technology: Large Language Models (LLM). In recent years, LLMs have captured the public's attention and accelerated the pace of AI research like anything before. These models represent a clear advancement and application of the principles we have discussed, resulting in systems with an unprecedented ability to understand, generate, and interact with human language. They have impacted nearly every industry, abandoning pure academical interests and becoming a globally significant technology. This chapter will explore what LLMs are, how they are built, their remarkable capabilities, and their limitations.

2.6.1 What is an LLM?

Why Large Language models are defined as "large"? The term refers to more than just the simple number of parameters, which can vary from billions to trillions, but it also takes into account the huge amount of data they are trained on. The increase in computational resources and data not only results in quantitative improvements, but it also expands the ability of the model to adapt to new tasks never seen before. When models grow beyond a certain size and amount data, they begin to show skills and capabilities that they were never explicitly trained to perform. A model trained simply to predict the next word in a sentence might learn to translate languages, write poetry, summarize complex documents, or even perform rudimentary reasoning. LLMs distinguish themselves from their smaller predecessors due to this capability of generalization obtained from a simple training objective.

2.6.2 From Statistical Models to Transformers

Modern LLM are the result of decades of research. Early statistical models could predict the next word based on the previous few, but they missed any real understanding of grammar or meaning. The adoption of neural networks, particularly LSTMs, lead to a better understanding of linguistic nuances, however they suffered of short-memory issues. The true revolution, however, began with the Transformer architecture. Models like BERT and the GPT series were the first to combine this architecture with the "large-scale" philosophy. They demonstrated that by massively increasing the number of parameters and the amount of training data, a Transformer model could achieve a remarkably deep and flexible understanding of language. This discovery started a race to scale that has shaped the last several years of AI research, leading directly to the powerful models we have today.

2.6.3 The Two-Stage Training Paradigm

Modern LLMs are obtained by applying two training procedure, each of them with different objectives in mind: pre-training and post-training, or finetuning.

1. **Pre-training**: The first stage is pre-training, an incredibly resource-intensive process where the model is trained on a vast and diverse corpus of text. The goal here is not to teach the model any specific task, but to force it to learn the fundamental patterns of language, grammar, reasoning, and factual knowledge embedded in the data. For months, the model simply learns to predict the next word in a sentence, and through this simple objective, it builds a comprehensive internal representation of the world as described in text. The result of this phase is a foundational or "vanilla" model, which can serve as a performance lower bound.

2. **Fine-tuning**: A pre-trained LLM is like a university graduate with a vast general education but no specific job training. The second stage, finetuning, is the process of specializing this general model for a particular task. By continuing the training on a much smaller, curated dataset, we can adapt the model to become an expert in a specific domain. In the context of this thesis, finetuning is used to adapt a base model to the specific task of generating a complete recipe from a given title. To make this process more accessible, efficient methods like LoRA finetuning are often employed. Figure 2.10 provides an overview of this entire training pipeline.

2.6.4 Capabilities and Limitations

The capabilities of modern LLMs are vast, but it is just as important to understand their weaknesses. Beyond their emergent abilities, LLMs have become powerful tools due to their capacity for **in-context learning**. They can often perform a task with just a few examples provided in the prompt, without needing any changes to their internal parameters, allowing them to be adapted on the fly for a wide range of linguistic applications. Despite their power, LLMs suffer from several well-documented challenges. They are prone to **hallucinations**, where they generate confident and plausible-sounding information that is completely false. They can also inherit and amplify the biases present in their training data. For the scope of this research, a particularly critical weakness is that LLMs often struggle in text generation tasks where numbers are involved. This difficulty with numerical precision in a procedural context like recipe generation is a primary motivation for exploring alternative loss functions.

2.6.5 Taxonomy of LLMs

- **Base Models . Instruction-Tuned Models**: A base model is the direct output of the pre-training phase. It is excellent at completing text but does

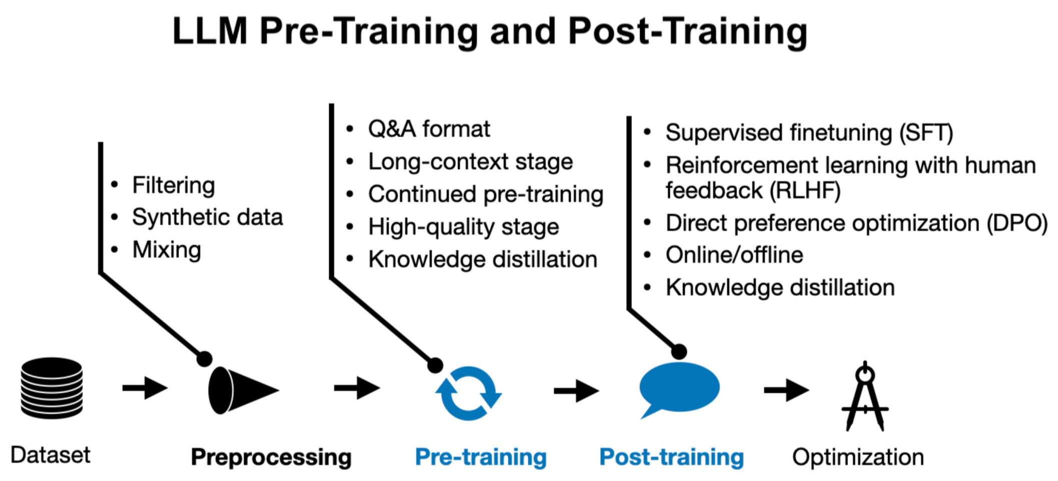


Figure 2.10: Overview of the Large Language Model (LLM) training pipeline, encompassing both pre-training and post-training stages. The process begins with a dataset that undergoes preprocessing, including filtering, synthetic data generation, and mixing. This preprocessed data then feeds into the pre-training phase, which involves techniques such as Q&A format training, long-context stages, continued pre-training, high-quality data stages, and knowledge distillation. Following pre-training, the model undergoes post-training, utilizing methods like supervised finetuning (SFT), reinforcement learning with human feedback (RLHF), direct preference optimization (DPO), and further knowledge distillation. The final stage involves additional optimization techniques.

not necessarily know how to follow user instructions or hold a conversation. An instruction-tuned model (often called a chat model) is a base model that has undergone an additional finetuning step on a dataset of instructions and responses, making it much better at being a helpful and interactive assistant. For instance, models like the base Llama 3 or Mistral 7B serve as foundations, while their instruction-tuned counterparts, such as Llama 3 Instruct or Mistral 7B Instruct (and proprietary models like OpenAI's GPT series which are often accessed in their instruction-tuned form), are specifically designed for interactive use.€

- **Closed-Source vs. Open-Source**: Closed-source models are proprietary and typically accessed only through an API provided by the company that developed them. In contrast, open-source models release their parameters publicly, allowing researchers and developers to run, modify, and finetune them on their own hardware. This research, for instance, utilizes several open-source models, including Qwen2.5 - 1.5B, SmolLM - 3B, and Qwen3 - 4B. The differences between base and instruction-tuned models are summarized in Table **??**.

Table 2.2: Comparison of features between a Base Large Language Model (LLM) and an Instruction-Tuned Large Language Model. This table highlights differences in their purpose, task specialization, response style, consistency, ability to handle complex tasks, adaptability to tone and format, alignment with user intent, and example applications.

Feature	**Base LLM**	**Instruction-Tuned LLM**
Purpose and Training Focus	Trained on broad datasets to learn general language and patterns, but without focus on specific instructions	Fine-tuned with instruction-specific data, allowing it to understand better and respond to user commands
Task Specialization	Performs well on broad language tasks	Specially optimized for following instructions and performing tasks on demand
Response Style	Outputs general language patterns, less sensitive to variations in phrasing or context	Focused responses that directly address user queries, highly sensitive to specific prompts, can adapt better to user instructions
Consistency and Reliability	Response quality and format can vary across similar prompts and are less consistent for standardized tasks	Provides consistent responses across similar prompts, reliable for applications requiring uniformity
Handling Complex Tasks	Can struggle with multi-step or layered instructions	Better at managing complex, multi-step tasks due to instruction training
Adaptability to Tone and Format	Can respond to tone requests but may need additional prompting to adjust style or structure precisely	More responsive to tone and formatting changes; can switch between casual, formal, bulleted, or numbered formats
Alignment with User Intent	May provide indirect answers	Aims for responses aligned with user intent and preferences
Example Applications	Broad NLP applications like language translation, text generation	Chatbots, virtual assistants, task-specific agents

2.6.6 Conclusion

Large Language Models represent the culmination of decades of research, combining the architectural prowess of the Transformer with the brute force of massive-scale data. Their two-stage lifecycle of pre-training and finetuning makes them both powerful generalists and adaptable specialists. However, their immense size and the standard methods used to train them present practical challenges, especially when aiming for high factual and numerical fidelity. This reality necessitates the development of more efficient and intelligent adaptation techniques, a topic we will explore in the next chapter about Finetuning and Parameter-Efficient Finetuning.

2.7 Small Language Models

2.7.1 Introduction: A New Paradigm of Efficiency and Specialization

For several years, the story of language models was a simple one: bigger was always better. Research labs were in a race to the top, scaling models to astronomical sizes with the belief that pure scale was the primary driver of intelligence. While this "scaling law" philosophy produced incredibly powerful and generalist models, it also created a new set of challenges related to cost, accessibility, and practicality. In response to these challenges, a new and exciting counter-trend has emerged: the rise of Small Language Models (SLMs). This chapter explores this shift towards efficiency and specialization, detailing what SLMs are, why they are becoming so important, and how they represent a more sustainable and accessible future for applied AI.

2.7.2 What is a "Small" Language Model?

The adjective "small" is, naturally, relative. What was huge a few years ago may count as small these days. An SLM is thus best characterised not by a concrete count of parameters, but in comparison with the gigantic "frontier" models with hundreds of billions or trillions of parameters that abound. SLMs fall generally in the range of several hundred million to some billions of parameters—large enough to be potent, but small enough to handle. This study, for example, involves a few models that fall neatly into this range, such as Qwen2.5 - 1.5B, SmolLM-3B, and Qwen3 - 4B.

2.7.3 SLM Philosophy: Data Quality more than Data Volume

Among the key insights behind successful contemporary SLMs is a change in philosophy from "big data" to "good data." The big-data-style strategy of training a model with a crude, unfiltered scrape of the whole web is being replaced by a finer-grained strategy. The "less is more" maxim implies that training a small model with an exhaustively curated, high-quality, diverse set of data can match or exceed the performance of a larger model trained with lower-quality data. Quality data is the key to creating successful and reliable SLMs, and that is an important foundation.

2.7.4 Key Techniques for Creating High-Performing SLMs

Other than utilizing improved data, certain significant techniques were devised for bettering small models' performance.

- **Knowledge Distillation**: This is an ingenious technique in which a small "student" model is trained to reproduce the outputs of a much larger, more proficient "teacher" model. The student is not only trained from the correct solutions in a dataset, but from the subtle probabilities and "reasoning" profiles of the teacher. This, in turn, imprints the advanced knowledge of the large model in a much more economical and lighter form.

- **Curated Pre-training Corpora**: The step of selecting and cleaning the data with great care before pre-training starts is no longer optional. By eliminating redundant or low-quality documents and making the data diverse and well-balanced, data scientists can give the model a better foundation from the beginning, enabling it to do more from the onset. The process of knowledge distillation is illustrated in Figure 2.11.

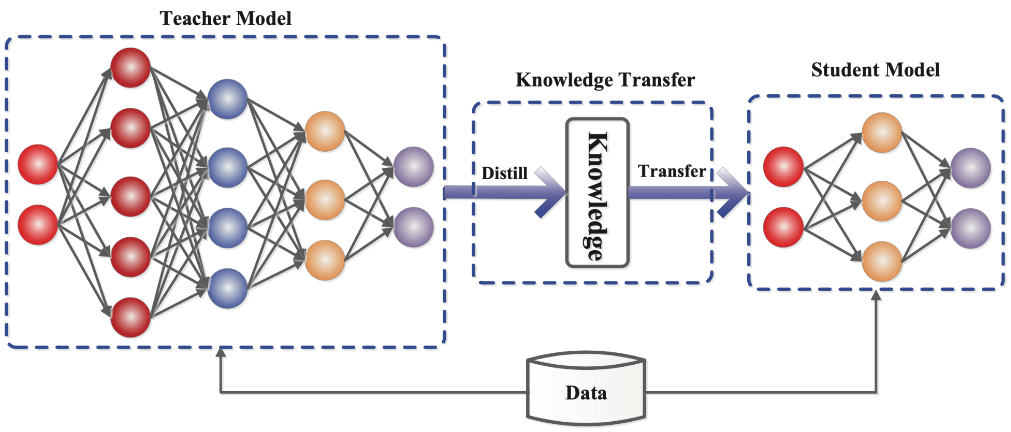


Figure 2.11: A diagram illustrating the process of knowledge distillation. A larger, more complex "Teacher Model" is first trained on the data. The "Knowledge" from this teacher is then distilled and transferred to a smaller, more efficient "Student Model," which learns to mimic the teacher's behavior. This process allows the student model to achieve a performance level close to the teacher's, while being more lightweight and computationally less expensive.

2.7.5 The Central Trade-off: Generalist Giants or Specialist Experts?

The contrast between a large LLM and a well-tuned SLM can be framed as that between a specialist and a generalist. The huge LLM has "read" much of the internet; it possesses a common sense, big-picture grasp of the world and is conversant about almost anything. Yet, in some cases, knowledge of it is shallower and more error-prone. An SLM, when it is fine-tuned for a particular domain, becomes a specialist expert. It knows nothing about history or astrophysics, but it can attain utterly stunning accuracy and reliability in its own narrow sphere of competence.

2.7.6 The Benefits of Small Language Models

The real-world advantages that spurring SLM adoption are impressive:

- **Computing Power and Cost**: SLMs need far less computing power for fine-tuning as well as for running inference. It reduces the financial cost by an enormous factor and brings them into the realm of far more researchers and organizations.

- **Lower Latency**: Because they're smaller, SLMs process data, build responses, and accomplish all that far faster. Lower latency is more important for real-time interactive applications like chatbots or coding assistants.
- **Flexible Deployment**: Because SLMs take low footprints, they can deploy in diverse settings. On-premise, they can run on a firm's inhouse server, local desktop computer, or edge devices such as in-car systems or smartphones.
- **Privacy and Security**: It is only by running a model locally or on a personal server that sensitive data need never travel through a third-party API, a critical requirement for most companies.

2.7.7 Limitation and Difficulty

There is, of course, a catch for such efficiency. SLMs cannot replace their large siblings in all applications.

- **Lower General World Knowledge**: SLMs, by their design, can store much less of the great corpus of facts that large models absorb in pre-training.
- **Fewer Emergent Skills**: They also generally lack the large, surprising skills such as multi-step reasoning or advanced creative writing that emerge in the largest models.
- **Catastrophic Forgetting**: There is more chance that in finetuning, an SLM can "forget" some of the initial general competences as it overspecializes for the new task. It is a problem that needs to be handled with care.

However, such models also tend to specialize because even the largest LLMs also have their vulnerabilities; for example, it is no secret that LLMs would fail in applications of text generation with numbers.

2.7.8 SLMs' Ideal Use Cases

SLMs are suited best for applications in which deep, consistent knowledge of a narrow domain is more in preference than the shallower, more broad knowledge. It is suited for applications such as domain-specialized chatbots, medical records summarization, and expert content creation. An application such as creating recipes, with issues such as low recall of ingredients, is a best-suited application for a domain-specialized SLM that can be highly fine-tuned to learn the specific structure and factual requirements of a recipe.

2.7.9 The Contribution of SLMs in This Research

The choice of using models in the sub-5-billion parameter count for the purpose of this thesis was a deliberate one. The models that were selected for these experiments—Qwen2.5 - 1.5B, SmolLM-3B, and Qwen3 - 4B —were specifically chosen because they fall into that class of fast, flexible, and specialist models. Their small sizes make them best suited for fast experimentation, and they allow for rigorous testing of a range of custom loss functions. Furthermore, they fall into that class of models most likely to appear in real-world, application-special applications such as the one analyzed here.

2.7.10 Conclusion: The Future is Both Large and Small

We summarize the key benefits and limits of both Small and Large Language models in Table 2.3 Ultimately, the scaling race is not concluded by Small Language Models but rather is their maturation. The AI of the future is not a monolithic system but an eclectic ecosystem. Into such a world, large, all-purpose "utility" models will exist side by side with a constantly shifting collection of highly-optimized, specialist SLMs. These easy-to-use and economical models will fuel the scaling of the next generation of AI, power innumerable real-world applications, and bring the promise of the language models into practical reality for all.

Table 2.3: Comparison of Small Language Models (SLMs) and Large Language Models (LLMs)

Feature	**Small Language Models (SLMs)**	**Large Language Models (LLMs)**
Knowledge Scope	Specialist: Deep knowledge in a narrow domain after finetuning.	Generalist: Broad knowledge across many topics; a "big-picture" grasp.
Accuracy / Reliability	Can attain high accuracy and reliability within their specialized domain.	Knowledge can be shallower and potentially more error-prone, especially in niche areas.
Computational Cost (Training & Inference)	Requires far less computing power, reducing financial costs significantly. More accessible.	Requires substantial computing power and resources, leading to higher costs.
Latency	Lower latency, faster responses, suitable for real-time applications.	Higher latency due to larger size and computational demands.
Deployment Flexibility	Highly flexible: can run on-premise, local machines, or edge devices (smartphones, cars).	Less flexible: typically requires significant server infrastructure or cloud APIs.
Privacy & Security	Enhanced privacy/security as they can run locally, keeping sensitive data in-house.	Often rely on third-party APIs, potentially exposing sensitive data.
Ideal Use Cases	Domain-specific applications requiring deep expertise (e.g., specialized chatbots, recipe generation, medical summarization, on-device AI).	Broad, general-purpose tasks requiring common sense reasoning, wide knowledge, and creative text generation across many domains.

2.8 Data Analysis Using Topological and Geometric Methods

2.8.1 Introduction: The Shape of Data, Beyond Traditional Statistics

In a subject as rich and complex as recipes, the structure among recipes can be more valuable than the recipes themselves. Traditional statistics can describe the mean number of ingredients or the cook verb most likely used, but too often it misses the big picture. To view the large-scale patterns we require a different set of tools. This is where Data Analysis (TDA) comes in. It is a modern field of mathematics that allows for a very powerful lens in viewing the "shape" in the data and revealing inherent structure that may otherwise go unseen [29].

2.8.2 From Data Points to Geometric Shapes: The Simplicial Complex

The first step in TDA is translating our cloud of abstract data points, in this case, recipes, into a shape we can see and handle geometrically. The concept's something like cosmic connect-the-dots. We get started with drawing a line segment joining any two recipes which are extremely close together. Then we finish any three recipes in which each one's close to the others with a triangle, and so on.

2.8.2.1 A Formal Definition

This process becomes formalized through the creation of a mathematical structure, a simplicial complex. The abstract simplicial complex K consists of a collection of finite sets, such that if the set σ belongs to K then any subset σ belongs also in K. Each σ contained in K we call a simplex: the 0-simplex is the vertex (a point), a 1-simplex the edge, the 2-simplex the triangle, etc.

Constructing this from data, we normally employ the Vietoris-Rips complex [30]. For a point set X and proximity threshold ϵ for distances, the Vietoris-Rips complex $V_\epsilon(X)$ consists in the set of all point subsets S with the property that the distance in X from each pair in the subset S does not exceed ϵ.

$$V_\epsilon(X) := \{\sigma \subseteq X \mid \forall x, y \in \sigma, d(x, y) \leq \epsilon\}$$

2.8.3 Homology: A Language for Describing Shape

Once we have shape, we need a language to describe it. Homology is the algebraic structure used by TDA in order to count and classify in a systematic fashion the various types of "holes" in our simplicial complex. [31]

- 0-Dimensional Holes (Connected Components): Counts the number of separate clusters or islands of data.

- 1-Dimensional Holes (Loops): Counts the number of independent loops or circular patterns.
- Higher-Dimensional Holes (Voids, Cavities): Counts hollow spheres or voids in higher dimensions.

2.8.3.1 The Algebra of Homology

Mathematically, homology is defined through a sequence of vector spaces and linear maps. For each dimension q, we define the chain group C_q as the vector space of all formal sums of q-simplices. The boundary operator $\partial_q : C_q \to C_{q-1}$ is a linear map that takes a simplex to the sum of its faces. From this, we define two important subspaces:

- The cycle group $Z_q = \ker(\partial_q)$, which contains chains with no boundary (i.e., loops).
- The boundary group $B_q = \mathrm{im}(\partial_{q+1})$, which contains chains that are themselves boundaries of higher-dimensional shapes.

Because every boundary is also a cycle, we have $B_q \subseteq Z_q$. The q-th homology group is then defined as the quotient vector space:

$$H_q(K) = Z_q(K)/B_q(K)$$

The dimension of $H_q(K)$ gives us the number of independent q-dimensional holes in our shape K.

2.8.4 The Scale Challenge and the Solution of Persistence

The first issue that directly comes up is: how far apart should we select ϵ? Too small an ϵ and we get lots and lots and lots of widely scattered points, too large and everything collapses into one large blob. The key solution to this problem is Persistent Homology.

2.8.5 Persistent Homology in Detail

Instead of choosing one ϵ, persistent homology analyzes the data across *all* possible scales simultaneously. This is done by creating a filtration, which is a nested sequence of simplicial complexes as ϵ increases: $V_{\epsilon_1} \subseteq V_{\epsilon_2} \subseteq \dots$ for $\epsilon_1 < \epsilon_2 < \dots$.

2.8.5.1 Birth, Death, and Persistence

persistent homology follows the lifespan of topological features along the course of filtration. When a new hole emerges at some specific distance, we refer to it with its "birth". Once ϵ keeps increasing, that hole shall finally get filled in, something we refer to with its "death". The "persistence" for a hole allows us to refer to the time from its birth till its death.

2.8.5.2 The Persistence Diagram: Visualizing Topological Features

The final result is a persistence diagram, a 2D plot whose points are the holes, with the x and y coordinates being its birth and death times. Those features well away from the diagonal are robust and the significant signals in the data.

2.8.6 From Analyzing a Single Shape to Comparing Two: Optimal Transport

While persistent homology is perfect for comparing a single point cloud, our thesis demands that we *compare* two highly disparate point clouds: the one from our model and the one from the ground truth. Here we refer back to the notion of Optimal Transport whose basic idea can be described with the analogy of the so-called "Earth Mover's Distance" [32] : how can we transport in the optimal way a heap of dirt from some initial state to some terminal state?

2.8.7 The Wasserstein Distance: Formalizing the Earth Mover's Distance

The mathematical formulation behind the idea is the Wasserstein distance. For two probability distributions μ and ν the p-Wasserstein distance can be defined as:

$$W_p(\mu,\nu) = \left(\inf_{\gamma \in \Pi(\mu,\nu)} \int d(x,y)^p \, d\gamma(x,y) \right)^{1/p}$$

Here, $\Pi(\mu,\nu)$ represents all admissible transport plans γ which describe how the mass from μ can be transported to ν, and $d(x,y)$ represents the cost transferring a single mass unit from point x to point y. The equation computes the plan with the lowest possible cost. Its key disadvantage resides in the computationally intensive cost of this optimal problem [33], and so the equation can't be utilized in training the model.

2.8.8 Sinkhorn Divergence: Approximating Optimal Transport

Sinkhorn Divergence is a computationally efficient and differentiable approximation of the Wasserstein distance, making it perfect for deep learning. [34]

2.8.8.1 The Role of Entropic Regularization

The key innovation is adding an entropic regularization term, $\epsilon H(P)$, to the optimization objective. The entropy of a transport plan P is defined as:

$$H(P) = -\sum_{i,j} P_{ij} \log(P_{ij})$$

This term "blurs" the transport plan, making the problem easier to solve. The full regularized objective is:

$$W_\epsilon(\alpha, \beta) = \min_{P \in U(\alpha,\beta)} (\langle P, C \rangle - \epsilon H(P))$$

2.8.8.2 The Final Formulation

The Sinkhorn divergence, S_ϵ, then debiases this value to ensure the distance of a distribution to itself is zero, making it a well-behaved loss function:

$$S_\epsilon(\alpha, \beta) = W_\epsilon(\alpha, \beta) - \frac{1}{2} W_\epsilon(\alpha, \alpha) - \frac{1}{2} W_\epsilon(\beta, \beta)$$

2.8.9 The Connection with Topological Loss

Our **Topological Loss**'s computational engine is the Sinkhorn divergence. It works off the principle of Optimal Transport, computing a "Wasserstein-like" distance that assesses the geometric dissimilarity between our model's predicted "soft" embeddings point cloud and the point cloud of the ground-truth "hard" embeddings.

2.8.10 Conclusion: A Bridge Between Topology and Deep Learning

This chapter has traced a path from the abstract idea of a data's "shape" to a practical algorithm for comparing point clouds. TDA provides a powerful language for geometric thinking. Optimal Transport offers a principled way to compare shapes. Finally, Sinkhorn divergence provides the computationally feasible algorithm needed to implement this comparison as a loss function inside a neural network. The work in this thesis builds upon this powerful bridge, connecting abstract geometric theory to the practical art of training better, more factually aware language models.

2.9 Finetuning

The advent of large, pre-trained models created an interesting new challenge for the AI community. We suddenly had these digital brains with a vast, generalist knowledge of the world, capable of discussing everything from history to programming. The question was no longer just about building bigger models, but about how we could take this powerful, general knowledge and make it useful for specific, expert tasks. To answer this question, we need to dive into the concept of finetuning, a process that transforms a model from generalist to specialized. In this chapter we will understand what is finetuning, what are the main components of it, which variants exist and the main challenges.

2.9.1 What is Fine-Tuning?

Finetuning is essentially a second phase of training that follows the initial, massive pre-training stage. You start with a powerful, general-purpose model and continue its training, but on a much smaller and highly specialized dataset [35]. The advantage of this approach is that we don't need to train a massive model from scratch for every new task, which would be computationally impossible but instead, we take the rich, internal representations the model has already learned and simply adapt them to the specific nuances of our target domain. For this thesis, it's the process that teaches a general model the specific language and structure of a good recipe.

2.9.2 Key Components of the Fine-Tuning Process

The source for this specialized training is the finetuning dataset and its quality is important, as the model's final performance depends on the quality of the data it learns from. A popular and effective format, and the one used in this work, is the instruction-following dataset. Here, the model is presented with thousands of input-output pairs. In our case this meant showing the model a prompt like "Generate a recipe for Pasta Carbonara" and the corresponding high-quality JSON output [36]. This direct approach provides a clear, unambiguous signal, teaching the model the precise format and style we expect as an outcome. This instruction-following finetuning process is visualized in Figure 2.12.

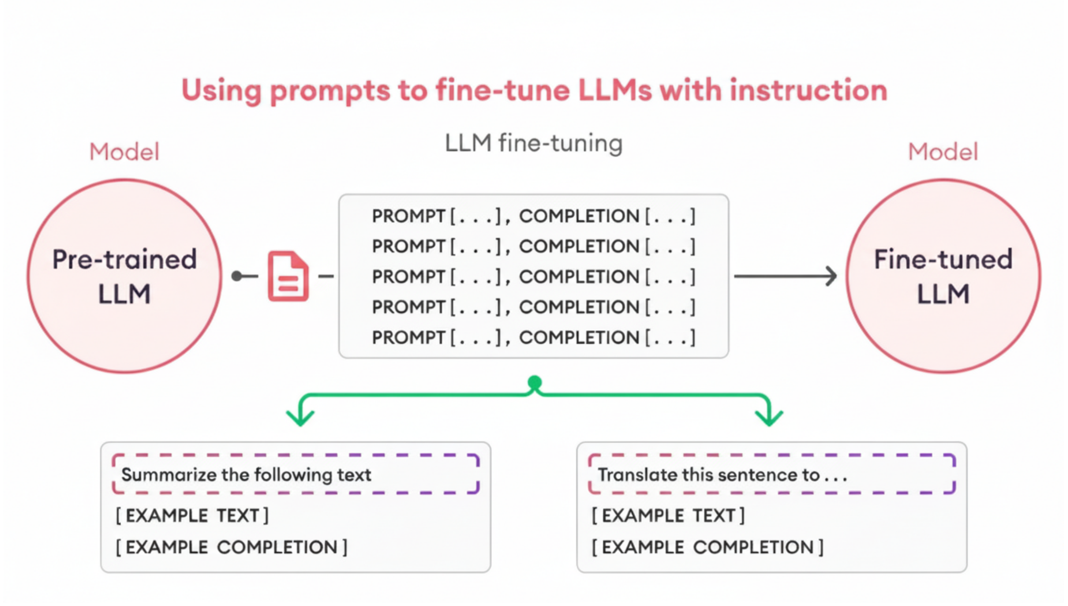

Using prompts to fine-tune LLMs with instruction
Model
LLM fine-tuning
Model
Pre-trained LLM
PROMPT [. . .], COMPLETION [. . .]
PROMPT [. . .], COMPLETION [. . .]
PROMPT [. . .], COMPLETION [. . .]
PROMPT [. . .], COMPLETION [. . .]
PROMPT [. . .], COMPLETION [. . .]
Fine-tuned LLM
Summarize the following text
[EXAMPLE TEXT]
[EXAMPLE COMPLETION]
Translate this sentence to . . .
[EXAMPLE TEXT]
[EXAMPLE COMPLETION]

Figure 2.12: A visual explanation of the process of using prompts to fine-tune a large language model. A pre-trained LLM is shown being adapted into a fine-tuned LLM using a dataset of prompt-completion pairs. Examples of these pairs for different tasks, such as summarization and translation, are provided to illustrate the instruction-following nature of the fine-tuning process.

2.9.3 Challenges and Strategies

The original approach to this process was direct and conceptually simple: full parameter finetuning. In this method, every single weight in the massive pre-trained model is "unfrozen" and allowed to be updated during the second training phase.

This brute-force technique allows the entire network to adapt to the new data, and for that reason, it often yields the best possible performance. However, this implies a huge computational cost. As models scaled into the billions of parameters, the hardware requirements for full finetuning exploded. The amount of GPU memory needed simply to load the model, let alone store the gradients and optimizer states for training, became immense. This created a significant limitation, making it financially and logistically impractical for most academic labs and even many companies to adapt these powerful new models. The rapid scaling had created a new problem: these incredible models were becoming too big to handle, threatening to slow down innovation.

This challenge started the development of more clever and nuanced finetuning strategies. One interesting strategy is multi-task fine-tuning, which consists of training the model on several tasks at once instead of narrowing the focus on a individual task. The idea is that learning complementary skills can lead to a more robust and generalized understanding of the core domain. In our own work, for instance, the models were trained on both recipe generation and a set of related question-answering tasks. This multi-task approach encourages the model to learn how to deal with the final application (in this case, recipe generation) by solving different problems that are important in order to obtain recipes of high quality strengthening its overall understanding of cooking recipes and its domain. The loss function is the component that determines the model's adjustments during training and its choice is critical because it defines what we consider to be a "good" or "bad" prediction, and thus what the model should prioritize learning. By augmenting the standard Cross-Entropy with a custom loss, we can direct the model's focus towards the elements we care about most—in our case, improving training accuracy on specific elements of a recipe, for example numerical values and ingredients. Overall finetuning represents the essential connection between a general model's potential and its real-world application. The brute-force method of updating every parameter is powerful but its cost made it prohibitive for the majority of innovation makers in the field. The necessity for efficient specialization has lead to the development of a new family of techniques knows as Parameter-Efficient Finetuning (PEFT), which offers an elegant solution to the problem of scaling.

2.10 PEFT - Parameter Efficient Tuning

2.10.1 Introduction: The Efficiency Revolution

The previous chapter left us at a critical juncture: Large Language Models are incredibly powerful, and finetuning is the key to specializing them, but the huge scale of these models created a massive computational barrier. The cost of full parameter finetuning threatened to lock this transformative technology away, accessible only to a handful of large corporations and research labs. This challenge, however, sparked an efficiency revolution in the AI community, leading to the development of a new family of techniques known collectively as Parameter-Efficient Finetuning, or PEFT [37]. This chapter explores these clever methods, which have made it possible for nearly anyone to customize and adapt even the largest models with a fraction of the resources.

2.10.2 The Core Principle: Freezing the Giant, Training the Specialist

The basic principle of all PEFT methods is elegance in simplicity. Rather than recalibrating all of the billions of parameters in a pre-trained model, we fix most of them. We recognize that a pre-trained model is already a master of language and reasoning, so we do not touch its basic knowledge. Then, we introduce or unfreeze a small subset of new, trainable parameters—the "specialist"—and train just these few parameters on our task-dependent data [38]. As a consequence, we can control the behavior of the model and fine-specialize knowledge by training far fewer than 1% of all of its parameters, cutting the required memory and computational power by a huge margin.

2.10.3 A Taxonomy of PEFT Methods

PEFT is not an algorithm but rather a family of various strategies, each of these strategies containing a philosophy of how it should, in an efficient manner, train the specialist parameters. These methods fall into one of the following three categories [39]:

- **Additive Techniques**: These introduce new learnable modules or parameters to the frozen model.
- **Selective Methods**: These choose a limited, strategic portion of the model's starting parameters to "unfreeze" and learn.
- **Reparameterization Techniques**: These subtle techniques adjust the form of the model's layers in order to describe the transformations in a more efficacious manner, as in LoRA. Figure 2.13 shows a taxonomy of these different PEFT strategies.

2.10.4 Additive Techniques: Injecting New, Trainable Modules

One of the most frequent techniques is to add small, new modules to the frozen LLM architecture. Think of it as inserting a small plugin into a large, complex software program. The base model isn't changed, but the plugin is adding new, specialized functionality. In the model, these "adapter" modules would typically be small neural networks inserted in between the pre-existing Transformer layers [38]. During finetuning, only the weights of these minuscule new adapters are trained, so that they can learn the specifics of the new task and propel the outputs of the larger, frozen layers.

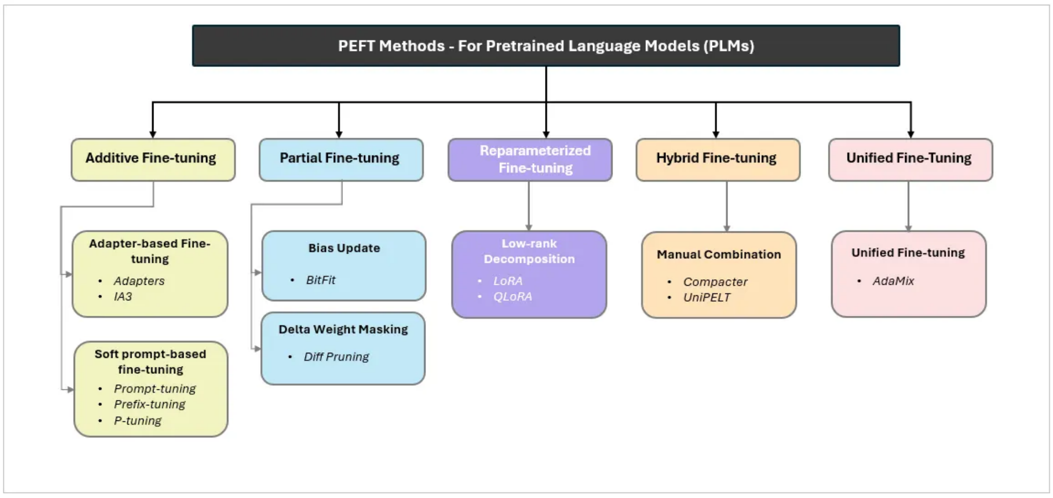


Figure 2.13: An overview of various Parameter-Efficient Fine-Tuning (PEFT) methods for pre-trained language models. The diagram categorizes these methods into five main groups: Additive Fine-tuning (including adapter-based and soft prompt-based methods), Partial Fine-tuning (such as bias updating and delta weight masking), Re-parameterized Fine-tuning (like low-rank decomposition methods), Hybrid Fine-tuning (using manual combinations), and Unified Fine-tuning.

2.10.5 A Deeper Look into Prompt-Based Tuning

One especially creative form of additive technique is prompt-based tuning. Rather than altering the internal architecture of the model, these methods aim at the input. The rationale is that rather than fine-tuning the model's "brain," we can learn the ideal "magic words" to achieve the desired output. These aren't words that humans would read but instead a series of trainable embedding vectors—a "soft prompt"—that is added to the front of the input [40]. During training, the model is trained on the best values for that soft prompt in order to best direct its frozen parameters to solve some particular task. Figure 2.14 compares this "prompt tuning" approach to traditional "prompt design".

2.10.6 Selective Methods: Finetuning a Small Subset of Existing Weights

The most obvious way of performing PEFT is possibly the selective method. The thought is straightforward: if we cannot possibly train all of the parameters, why not then choose a small subset of the most relevant parameters and train only them? Some methods, for instance, include fine-tuning the bias terms in the network but not the weights, or unfreezing only the last few layers of the model [41]. Simple, the problem here is then accurately selecting that particular small subset of the roughly billion parameters that is most relevant to the new task.

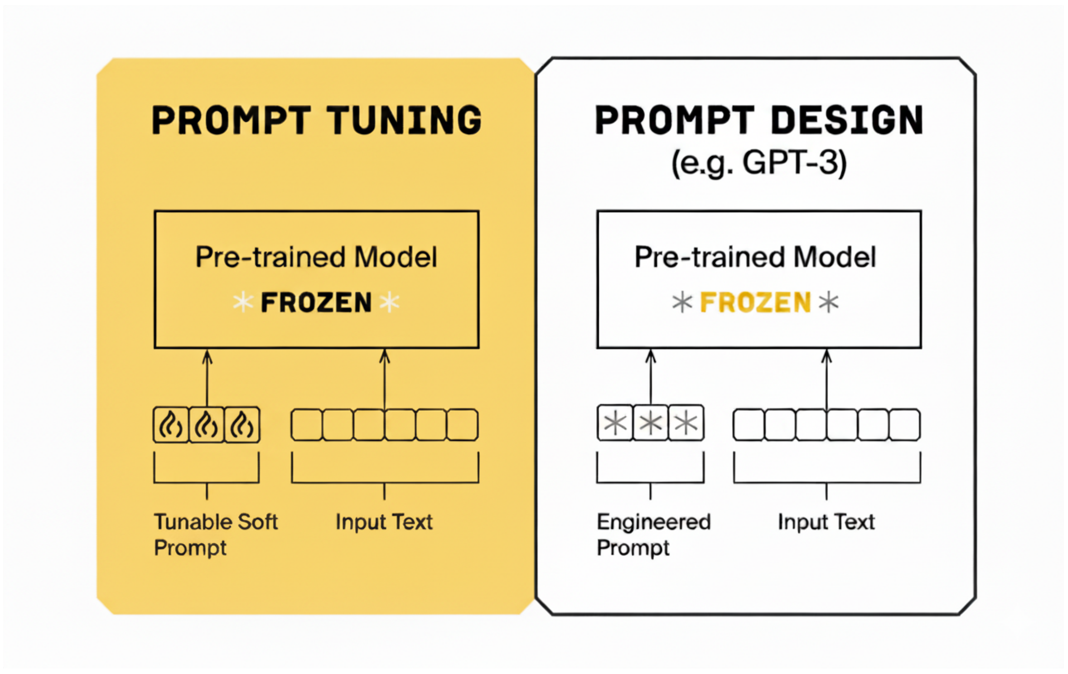


Figure 2.14: A comparison between prompt tuning and prompt design. In prompt tuning, the pre-trained model is frozen, and a tunable soft prompt is trained to generate the desired output. In contrast, prompt design also uses a frozen pre-trained model but relies on an engineered, fixed text prompt to guide the model's behavior.

2.10.7 Reparameterization Techniques: Another Change of Basis

A more mathematically elegant approach is reparameterization. The core idea here is a clever trick. A layer in a neural network is defined by a large weight matrix. Instead of trying to update this entire, massive matrix, we can assume that the change we need to make is actually very simple and can be represented in a more efficient way. This is the foundation of LoRA, which approximates this large change using two much smaller, "low-rank" matrices.

2.10.8 Low-Rank Adaptation (LoRA)

The most prominent and most used reparameterization technique nowadays is Low-Rank Adaptation [42] (LoRA), with it being the main PEFT technique utilized in this project. LoRA's idea is intuitive. Consider the gigantic, pre-trained weight matrix as a masterpiece artwork. Full fine-tuning would mean replicating parts of the original masterpiece. LoRA goes in a different direction: it lays a transparent overlay on top of the masterpiece and performs all of its tweaks on it. Technically, LoRA freezes the original weight matrix and injects two small randomly initialized matrices next to it. While training, only the two small matrices are updated. To obtain the final answer, outputs of the original frozen matrix and the two new trained matrices are added up. In such a way, we preserve the strong knowledge of the pre-learned model but efficiently learn a small "adjustment" that fine-tunes it for our new task.

2.10.9 Advantages of PEFT: Why It Is a Game-Changer

The development of PEFT was absolutely revolutionary due to the following main reasons:

- **Massively Lower Computational Cost**: PEFT methods decrease the GPU training memory by one order of magnitude, enabling full-size models to be trained from scratch on single, consumer-level GPUs.
- **Efficient Storage**: It is an enormous practical advantage. Rather than maintaining a full, independent, multi-gigabyte copy of the whole model for every new task, you simply retain the low, frequently just few-megabyte, specialist parameters. It is then feasible to store hundreds of specialist models.
- **Easy Switching of Tasks**: Since the specialist parameters are few and independent of the base, one could have a large base model and switch, on the fly, various PEFT adapters in and out in order to move from a task such as recipe generation, summarization, or coding.

2.10.10 Disadvantages and Real-world Considerations

Of course, there are no methods that come completely dependency-free. While PEFT methods come very close to achieving the performance of a full fine-tuning, they occasionally narrowly fall behind on highly complex tasks for which a more basic change in the knowledge of the model is necessary [43]. Furthermore, they bring with them new hyperparameters that must be adjusted, as happens with the rank (r) in LoRA, for example, that determines the dimension of the trainable matrices. Figure 2.15 depicts how Llama-2-7B performs on code and math evaluation dataset when full-finetuning and LORA are respectively applied.

2.10.11 The Current PEFT Environment: Libraries and Tooling

The rapid spread of PEFT was driven by the availability of highly capable and user-friendly open-source software. Libraries like Hugging Face's `peft` became the standard for the community, with production-ready, highly optimized, and correct implementations of most methods, including LoRA. Libraries in turn democratized the process of finetuning, making it almost accessible to everyone to fine-tune best-in-class models in a couple of lines of code.

2.10.12 Conclusion: PEFT as the New Standard for Model Adaptation

PEFT was successful in resolving the access crisis brought about by the sheer magnitude of modern LLMs. By being in a position to provide a suite of techniques for effectively specializing such models, PEFT is rapidly becoming the gold standard against which model adaptation is being assessed in research as well as in practice.

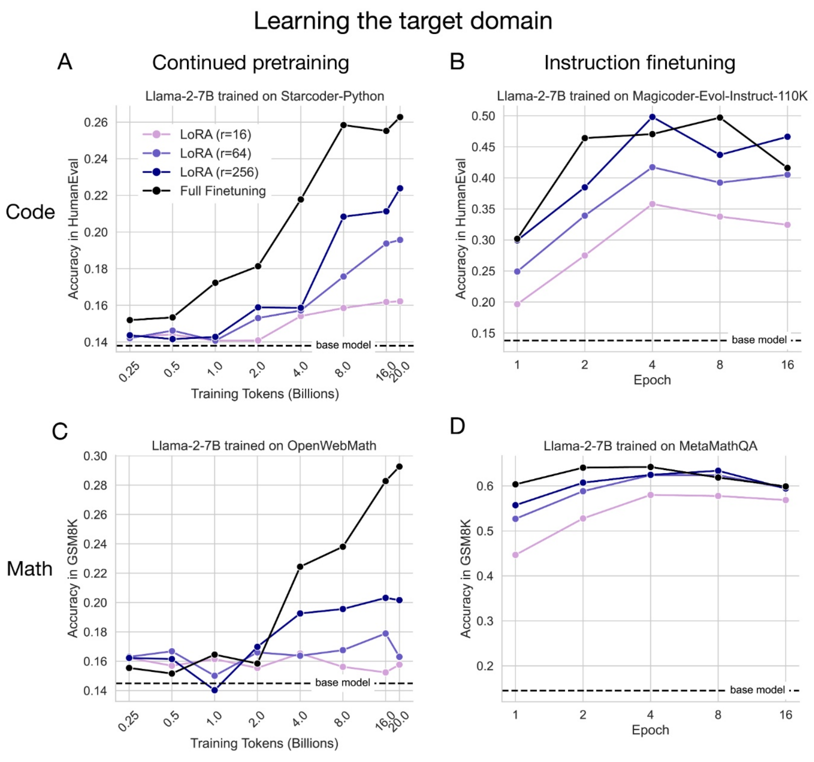


Figure 2.15: Performance comparison between LoRA (with different ranks $r = 16, 64, 256$) and Full Fine-tuning for the `Llama-2-7B` model across various coding and math tasks.

2.11 LORA - Low-rank adaptation

2.11.1 Introduction: A Principled Approach to Parameter Efficiency

The problem of scaling billion-parameter models efficiently has bred a whole range of new techniques. Of these, Low-Rank Adaptation (LoRA) [42] has come to prominence not as a mere practical hack, but as a principled and mathematically graceful method of parameter-efficient fine-tuning. It goes beyond adding or freezing parameters and rather reimagines the very representation of the learning process in a large model. This chapter is a technical deep dive into the theory behind LoRA, its mathematical derivation, as well as practical considerations that make it such a potent tool, warranting its choice as the base methodology for this thesis.

2.11.2 The Theoretical Foundation: The Intrinsic Rank Hypothesis

The whole LoRA methodology is constructed around an intriguing theory called the Intrinsic Rank Hypothesis. The theory proposes that while a pre-trained language model is a very complex, high-rank function, the transformation that needs to happen for it to adapt to a particular task is, unexpectedly, very easy and of low "intrinsic rank." That is, we wouldn't need some huge, complex transform from a generalist to a specialist model. The adaptation, rather, can be described by a much more modest, low-dimensional update. LoRA exploits such a hypothesis by assuming that we would not need to update the whole, complex weight matrix of a layer, but we need only figure out an optimal way of characterizing such a simple, low-rank adaptation.

2.11.3 The Mechanics of Low-Rank Adaptation: Decomposing the Weight Update

To understand the mechanics of LoRA, let's first consider a standard linear layer in a Transformer. Its behavior is governed by a pre-trained weight matrix, W_0. In traditional full finetuning, we would update this matrix by adding an update matrix, ΔW, which has the same large dimensions as W_0:

$$W_{finetuned} = W_0 + \Delta W$$

Training the entire ΔW matrix is the source of the high computational cost. LoRA's innovation is to constrain the structure of this update by decomposing it into two much smaller, low-rank matrices: a matrix B and a matrix A.

$$\Delta W = B \cdot A$$

Here, if the original weight matrix W_0 has dimensions $d \times k$, the LoRA matrices are defined with a small, shared dimension, r, called the rank. The matrix B will have dimensions $d \times r$, and the matrix A will have dimensions $r \times k$. The rank, r, is a hyperparameter that is chosen to be much smaller than d or k (i.e., $r \ll \min(d, k)$). This decomposition is the key to LoRA's efficiency.

2.11.4 Mathematical Formulation of the Forward Pass

With this decomposition, the forward pass of a LoRA-enabled layer is modified. The output, h, for an input, x, is the sum of the output from the original, frozen path and the new, trainable LoRA path. The pre-trained weights W_0 are kept frozen and do not receive gradient updates during training.

$$h = W_0 x + \Delta W x = W_0 x + BAx$$

In practice, the LoRA path is often scaled by a constant, α, which is typically set to be the same as the rank, r. This scaling helps to normalize the update's magnitude. The final forward pass is therefore:

$$h = W_0 x + \frac{\alpha}{r} BAx$$

During training, only the parameters of matrices A and B are optimized, while W_0 remains unchanged. A diagram illustrating this LoRA update mechanism is provided in Figure 2.16.

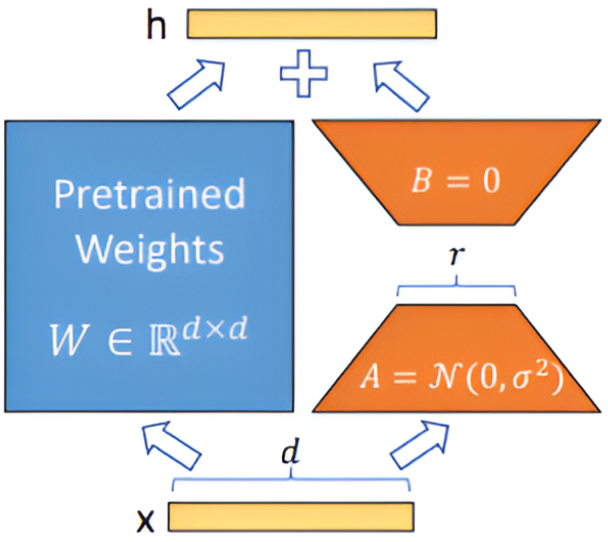


Figure 2.16: An illustration of the Low-Rank Adaptation (LoRA) method for parameter-efficient fine-tuning. The process involves keeping the original pre-trained weights, represented by the matrix W, frozen. A low-rank matrix, decomposed into two smaller matrices A and B, is trained to adapt to a new task. The output is a combination of the original forward pass with the pre-trained weights and the low-rank update, represented as $h = Wx + B(Ax)$. This approach significantly reduces the number of trainable parameters.

2.11.5 The Trainable Parameters: A Quantitative Analysis

The reduction in parameters by such a strategy is enormous. Take a typical weight matrix in a deep model with the input dimension d and the output dimension k being 4096.

- **Full Finetuning**: The number of trainable parameters in the ΔW matrix is $d \times k = 4096 \times 4096 = 16,777,216$.
- **LoRA Finetuning**: If we choose a small rank, for example $r = 8$, the number of trainable parameters is the sum of the parameters in A and B: $(d \times r) + (r \times k) = (4096 \times 8) + (8 \times 4096) = 32,768 + 32,768 = 65,536$.

It only needs to train 0.39% of the parameters that it would need for a complete finetune of that specific layer.

2.11.6 Key Hyperparameters: Rank (r) and Alpha (α)

The LoRA's action is mostly governed by two important hyperparameters:

- **Rank** (r): This is the most important hyperparameter. It specifies the rank of the update matrices and consequently the number of parameters that can be trained. The rank r determines the capacity and expressiveness of the adaptation. Increasing r enables more complex adaptations but raises the count of parameters and the danger of overfitting to the data of finetuning. Setting the rank low is more economical but may lack sufficient capacity for adapting the model fully.

- **Alpha (α)**: It is a scaling of the LoRA update. Modifying α allows you to scale the magnitude of adaptation compared to pre-trained weights, much as you would scale the magnitude of the gradient update by a learning rate. It is typically simply set equal to r.

2.11.7 Implementation in Transformer Models in Practice

In practice, LoRA is not typically applied to every weight matrix in a model. Research has shown that the most critical weights for task adaptation are often located within the self-attention mechanism. Therefore, a common and effective strategy is to apply LoRA only to the query (W_q), key (W_k), value (W_v), and output (W_o) projection matrices of the attention blocks, while leaving other layers, such as the feed-forward networks, completely frozen. This targeted approach further enhances efficiency while capturing most of the performance benefits. Given its effectiveness, LoRA was selected as the finetuning methodology for this research.

2.11.8 Merging LoRA Weights for Zero-Latency Inference

One of LoRA's most significant practical advantages is its behavior during inference. After training is complete, the learned matrices A and B can be multiplied to produce the full-rank update matrix $\Delta W = BA$. This update matrix can then be directly added to the original pre-trained weights:

$$W_{deploy} = W_0 + BA$$

The resulting W_{deploy} is a single weight matrix with the exact same dimensions as the original. This means that for deployment, the LoRA model can be "merged" back into the base architecture. The crucial takeaway is that LoRA introduces zero additional latency during inference, as there are no extra modules or calculations to perform. This sets it apart from other methods like adapters, which add a small but permanent computational overhead.

2.11.9 Technical Advantages

Technically, LoRA enjoys some of the following significant advantages: Massive Reduction in Trainable Parameters: As quantified above, this leads to lower VRAM requirements and faster training times. Zero Inference Latency: Since linearly combining of the weights is possible, the generated model is as inference-fast as the base model. Orthogonality with Other Methods: LoRA is a generic method that is extenable with other methods, for instance, with alternative schemes of optimization or with different methods of quantization.

2.11.10 Limitations and Theoretical Considerations

There is no technique that is not a compromise of sorts. The LoRA's key assumption, the low-rank hypothesis, must not generalize as well to all potential models or tasks. To achieve more dramatic model knowledge shifts, a low-rank update may potentially not match the performance of a full finetune. It is typically also an empirical choice of picking the optimal rank value of r, and experimentation and tuning is required for each specific task and data set.

2.11.11 Conclusion: LoRA as an Optimal Trade-off

Therefore, in conclusion, Low-Rank Adaptation offers a principled and highly successful solution to the problem of fine-tuning large models. Through reparameterization of the weight updates with the strong theoretical requirement of low intrinsic rank, it obtains an impressive decrease in computational cost with no injection of inference latency. It is the best trade-off among performance, efficiency, and ease of implementation, and as such, it justifiably occupies a prominent and potent role as an instrument for fine-tuning large language models for specialized domains.

2.12 Quantization

2.12.1 The Model Deployment Problem

After the process of pre-training and finetuning, a language model is at last complete. It has acquired the patterns of language and become expert at its assigned task but has one last, crucial hurdle to overcome: deployment. A model that exists solely in a research lab is a theoretical triumph but finds its true purpose when it can be applied to a real-world scenario. Here we hit a very real problem. The very size that gives these models their power also renders them slow, costly to run, and hardware-intensive. This is the "last-mile" problem of AI but the key technology that lets us surmount it is quantization. It's the crucial process that reduces these digital giants to a manageable size that is fast, efficient but practicable enough for day-to-day use.

2.12.2 What is Quantization? The Principle of Lower Precision

Quantization is actually a very basic notion at its core. It is the reduction in the numerical precision of the model's parameters. A standard language model keeps its weights in 32-bit floating-point numbers (FP32), a data type that has the capability of representing a very large number of values with very high precision. Quantization turns the higher precision values into a lower, more compact data type, the most typical form being an 8-bit integer (INT8).

The benefit of this new tack is a huge saving in efficiency. Consider it like a picture. An FP32 weight is a high-resolution professional photo file, packed with enormous detail. An INT8 weight is a high-quality JPEG of the same photo. It's a lot smaller and quicker to load, and for all intents and purposes, the miniscule details lost in compressing it are hardly perceivable. Quantization applies the same "good-enough" approximation principle to the numbers in a neural network.

2.12.3 The Mathematics of Quantization: Affine Mapping

The conversion of a high-precision floating-point number to a low-precision integer is typically done by a linear transformation that is quite a simple one called an affine mapping [44]. It is given by the equation below.

$$r = S(q - z)$$

Let's break down the components:

- r is the true original high-precision value (e.g., a 32-bit float).
- q is the new, quantized low-precision number (e.g., an 8-bit integer).
- S is the Scale factor. It is a positive floating-point number that decides the "step size" or resolution of our quantization. It gives us how much change in the real number corresponds to a change of 1 in the quantized number.
- z is the Zero-Point. This is an integer that ensures the true value of 0 can be represented exactly by some of the integer values from our quantized set. This is very important for operations like padding where a true zero is needed to represent the absence of information.

Whereas inferring the model performs its calculations using the fast, low-precision integers (q), and then converts the result back to a floating-point number using the scale (S) and the zero-point (z).

2.12.4 A Classification of Methods of Quantization

There are two major philosophies of when to employ this process of quantization [44]:

- **Post-Training Quantization (PTQ)**: This is the easiest method. A model is fully trained in its typical high-precision form. Once training is complete and final, a conversion process is run to quantize the weights. It's like having a complete item and refining it for efficiency.
- **Quantization-Aware Training (QAT)**: This is a more advanced technique where the model is made "aware" of the quantization during the training or finetuning process. The forward pass of the model simulates the effect of quantization, so the model learns to be robust to the small errors that will be introduced. This often results in a final quantized model with higher accuracy. Figure 2.17 provides a workflow comparison of PTQ and QAT.

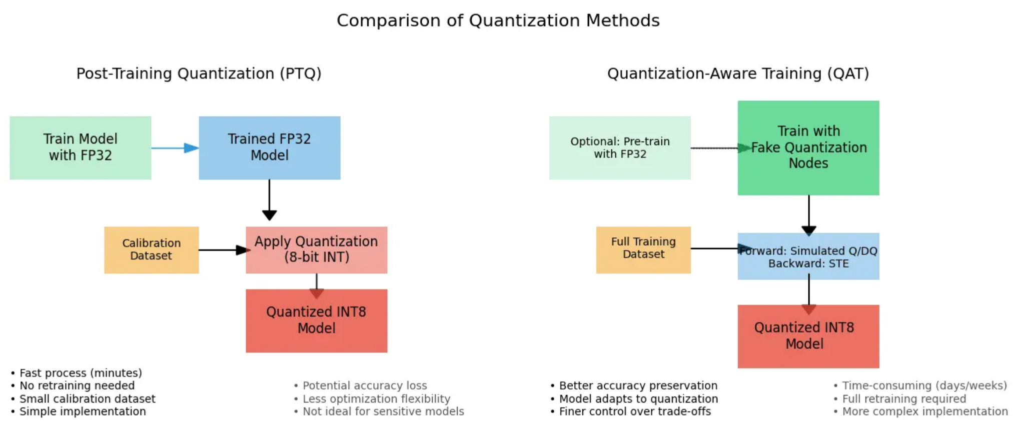

Comparison of Quantization Methods
Post-Training Quantization (PTQ)
Train Model with FP32
Trained FP32 Model
Calibration Dataset
Apply Quantization (8-bit INT)
Quantized INT8 Model
• Fast process (minutes)
• No retraining needed
• Small calibration dataset
• Simple implementation
• Potential accuracy loss
• Less optimization flexibility
• Not ideal for sensitive models
Quantization-Aware Training (QAT)
Optional: Pre-train with FP32
Train with Fake Quantization Nodes
Full Training Dataset
Forward: Simulated Q/DQ
Backward: STE
Quantized INT8 Model
• Better accuracy preservation
• Model adapts to quantization
• Finer control over trade-offs
• Time-consuming (days/weeks)
• Full retraining required
• More complex implementation

Figure 2.17: Workflow comparison between Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT). PTQ involves quantizing a pre-trained FP32 model using a calibration dataset to produce a Quantized INT8 Model, offering a fast process but with potential accuracy loss. QAT, conversely, integrates fake quantization nodes during the training process with a full training dataset, leading to better accuracy preservation and finer control over trade-offs, albeit being more time-consuming and complex.

2.12.5 A Further Look into Post-Training Quantization (PTQ)

Because of its effectiveness and simplicity, PTQ is the most commonly utilized technique today. It could also be categorized further into two subtypes:

- **Dynamic Quantization**: Under this method, the model's weights are pre-quantized while the activations (the neurons' outputs) are quantized "on-the-fly" when the model is executed for inference. Its benefit is that it's extremely easy to implement but it has the penalty of real-time conversion that runs a bit of computational cost.

- **Static Quantization**: This is the quicker of the two. Here, the weights and the activations are all quantized offline, prior to running. To accomplish this most effectively, the model must be "calibrated" by passing a small sample of typical data through it in order to establish the optimum scale and zero-point for the activations. The end result is a complete integer-based model that runs at top speed.

2.12.6 Exploring Bit Depths: 8-bit to 4-bit and Beyond

The compression level depends upon the bit depth of the object data type.

- **INT8 (8-bit)**: It is the most popular and well-supported level of quantization. It provides a 4x model size reduction and memory reduction and may offer considerable speedups for current hardware but often does so at a negligible loss in performance.

- **INT4 (4-bit)**: In the recent past, more aggressive 4-bit quantization has become very popular. It provides a 8x compression of model size such that very large models become executable using consumer-level GPUs [45]. But such a higher compression incurs a higher risk of performance loss because greater information gets lost while converting.

2.12.7 The Core Benefits: Why Quantize a Model?

The justification for accepting the quantization is pragmatic and overwhelming:

- **Smaller Footprint in Memory**: A model that has been quantized is a smaller file. This is less expensive to store, distribute, and load into limited CPU or GPU memory.
- **Faster Inference Speed**: For most modern hardware, mathematical operations by using integers are significantly faster than when using floating-point numbers. This translates directly to faster latency and a snappy response for the end-user.
- **Improved Energy Efficiency**: Computation is also faster when the model is compact. This means that less power is used for each inference run. This is important for battery-operated applications and reducing the running expenses of a high-capacity data center.

2.12.8 The Trade-off that is Unavoidable: Performance vs Efficiency

Note that quantization is a lossy compression scheme. We will inevitably need to incorporate small amounts of error by rounding the high-precision values to a lower number of possible integers. The key issues in the field of quantization are the building of schemes that will keep these accuracy errors to a minimum when it comes to the end performance of the model. A poorly managed quantization could decrease a model's accuracy, so a careful balancing has to be always achieved between efficiency enhancements and maintenance of performance.

2.12.9 PEFT and Quantization: A Close Relationship

Quantization and Parameter-Efficient Finetuning are complementary concepts but are otherwise potent technologies that are commonly employed together in order to have a very efficient end-to-end pipeline. A typical successful pipeline is to initially finetune a model using a PEFT technique such as LoRA. Once trained, the small specialist LoRA weights are combined back into the base model. Lastly, Post-Training Quantization is done to this combined, full-precision model so that it is ready for efficient deployment. This synergy has led to even more advanced techniques, such as QLoRA [45]. This innovative method cleverly integrates 4-bit quantization during the LoRA finetuning process itself, allowing researchers to finetune massive models with even less memory than standard LoRA. Figure 2.18 compares the memory workflow of Full Finetuning, LoRA, and QLoRA.

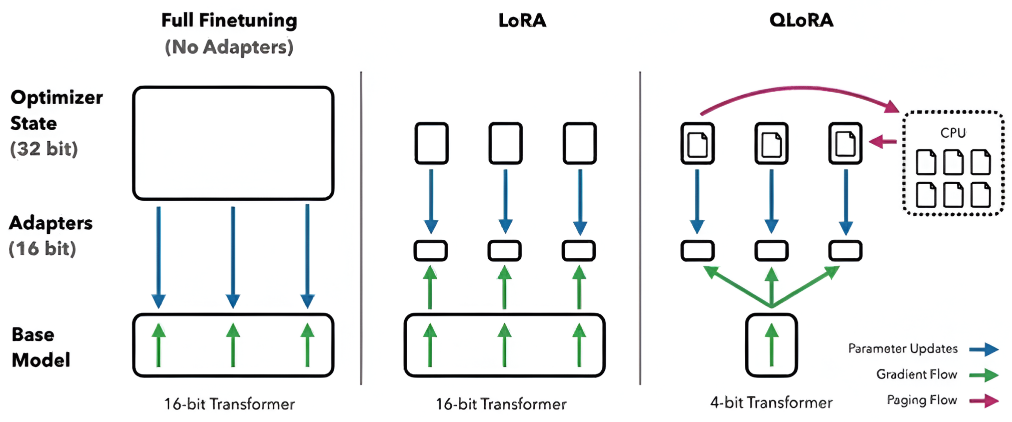


Figure 2.18: Comparison of finetuning methods: Full Finetuning, LoRA, and QLoRA. QLoRA leverages 4-bit quantization and paged optimizers for improved memory efficiency.

2.12.10 Conclusion: Quantization as a Standard for Deploying AI

In today's AI world, quantization has gone from a specialty optimization trick to a standard, indispensable step in the deployment pipeline. It's the bridge that turns the giant powerful models that are developed in research institutions into the efficient, usable, and accessible applications that are able to run in the real world.

Chapter 3

Related Work

3.1 The Evolution of Recipe Generation as an NLG Task

The computerized production of culinary recipes by large language models is a highly specialized and challenging subfield of Natural Language Generation. It involves more than just linguistic proficiency, as it needs to follow intricate, unforgiving structural and logical constraints that belong to the realm of procedural text. To achieve success, the recipe generator needs to operate, aside from being a proficient writer, as a very precise, commonsense-dense chef.

3.1.1 Classical Sequence-to-Sequence Approaches and Early Baseline

First studies in food computing and recipe generation used basic sequence-to-sequence models, specifically Long Short-Term Memory (LSTM) networks and Recurrent Neural Networks (RNNs).[46] Although these models set early performance standards for generating textual outputs, they revealed some inherent shortcomings in tackling long-range dependencies essential for multi-step culinary instructions.[46] The early transformer-based systems, like for the case of the Recipe1M+ dataset, all started with basic generation problems such as generating recipes from a list of ingredients.[47] It was typical for early models to simply be assessed with the typical translation metrics.[47] The obvious basic flaw, however, was that these neural models, no matter whether trained with large datasets, could not create text that was "structured, context- and commonsense-aware".[47] Recipes display procedural dependency, i.e., they behave much like a stream of actions where the successful execution of the present step is dependent solely on the prior steps' built-up state. It is much like a structured Markov Decision Process (MDP).[48] Early models' inability to retain that procedural context, i.e., forgetting that they've already added an ingredient, meant that the typical training objective was not sufficient to enforce long-distance procedural correctness rather than just-word accuracy in predicting. This difficulty in generating logically correct sequences necessitated optimization frameworks that could penalize executive errors in the procedural sequence and thus became a requirement.

3.1.2 Shifting Towards Transformer-Based LLMs and Adaptation

The transition to fine-tuning large, pre-trained transformer models, such as the GPT-2 family, was a huge stride in culinary NLG. Comparison works confirmed the architectural superiority of transformers compared to recurrent models, obtaining increases of well over in semantic relevance (BERTScore F1) and substantially reducing perplexity scores.[46] This established the transformer architecture as the basic requirement for addressing the complexity of generating recipes. Subsequent studies explored the impact of model scale, contrasting architectures ranging from smaller, resource-efficient models like T5-small and SmolLM-135M to larger foundational models like Phi-2.[49] These comparisons confirmed that while larger, more capable models provide superior linguistic capacity, domain adaptation remains paramount. A key technique for addressing domain specificity involved modifying the tokenizer. Generic subword tokenizers frequently break up crucial recipe elements, such as precise numerical quantities and fractions (e.g., $\frac{1}{2}$), leading to the loss of vital structural information.[46] To mitigate this, successful domain adaptation strategies include augmenting the tokenizer vocabulary with custom structural markers and dedicated fraction tokens, a critical step to prevent numerical and structural degeneration in the output.[46] While effective, this strategy primarily targets data representation during preprocessing; this thesis, however, concentrates on altering the learning dynamics during finetuning by introducing novel loss functions as the primary mechanism for improvement.

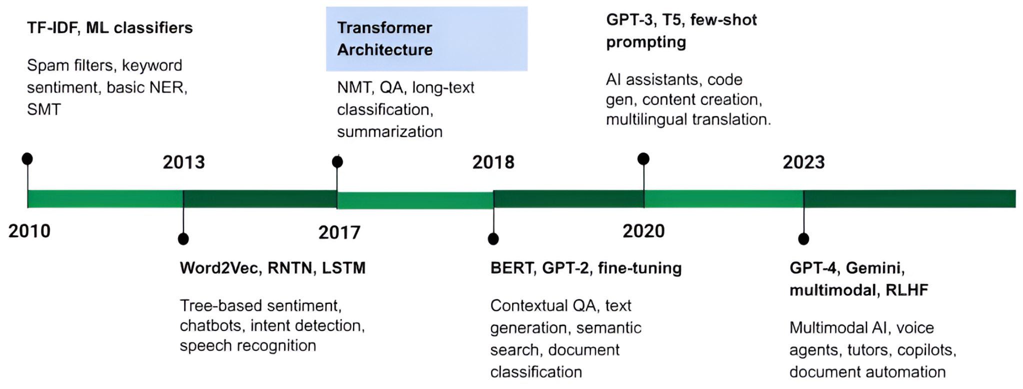

TF-IDF, ML classifiers
Spam filters, keyword sentiment, basic NER, SMT
Transformer Architecture
NMT, QA, long-text classification, summarization
GPT-3, T5, few-shot prompting
AI assistants, code gen, content creation, multilingual translation.
2010
2013
2017
2018
2020
2023
Word2Vec, RNTN, LSTM
Tree-based sentiment, chatbots, intent detection, speech recognition
BERT, GPT-2, fine-tuning
Contextual QA, text generation, semantic search, document classification
GPT-4, Gemini, multimodal, RLHF
Multimodal AI, voice agents, tutors, copilots, document automation

Figure 3.1: Evolution timeline of AI models in NLP domain

3.1.3 Recipe Generation as Constructed and Constrained NLG

Generation of recipes is technically a type of Structured Text Generation (STG), as the generated text must adhere to strict syntactic and semantic requirements.[50]

These in-born requirements are strict and imperative in order to achieve real-world usefulness. Cookbook texts must be clear and technically explicit, necessitating that best practices such as listing of ingredients in the order of usage, consistency of units and measures, and beginning instruction steps with imperative action verbs (e.g., "Chop," "Bake," "Mix") must be used.

The constraint enforcement problem is one that is well known in the literature. Even higher-order language models, when submitted for processing by their APIs, in their earlier days experienced breakdowns in maintaining format and context, their outputs reducing to repeated or meaningless combinations of ingredients.[51] This failure to ensure logical, functional format and factual integrity sets the task of generating recipes apart from that of the creative text task generally. Because the constraints, by their very nature, directly affect the safety and executability of the final product, structured output failures appear as pragmatic, oftentimes meaningless, instructions, thereby highlighting the importance of appropriate constraint enforcement in optimizing the model.[47]

3.2 Culinary NLG Datasets and Benchmarks

The type and nature of training data significantly affect an LLM's structuring of culinary text. The development of the recipe datasets is a trajectory from multimodal generic materials to text-based, high-fidelity structural norms, thus directly supporting the choice of the RecipeNLG corpus.

3.2.1 Predecessor Datasets (Recipe1M+ and Visual Focus)

First large culinary datasets, such as Recipe1M+, were for the most part authored with computer vision use cases in mind, i.e., with the objective of associating textual recipes with appropriate images.[47] This vision-centric prioritization very often created issues with integrity and validity of the textual data. Visually inspecting Recipe1M+ revealed pervasive structural issues, such as incorrectly created recipe structures, poor partitioning of instructions (with sentences utilized as steps in an improper manner), and erratic issues with the malformation or skipping of precise numerical fractions of ingredients.[47] These aspects made quantitative evaluation of textual quality unreliable and precluded generative models from being trained on logically coherent structures of ingredients and instructions.[47]. Some examples of recipes from Recipe1M can be found in Figure 3.2.

3.2.2 The RecipeNLG Dataset

The RecipeNLG dataset was created specifically as an answer to the textual shortage of the prior datasets. As a continuation of Recipe1M+, the design task was obvious: it was to aim more for the text of the recipes, their composition, and their reasoning, rather than for visual alignment.[47] Consequently, with well over a million newly preprocessed, also deduplicated, recipes, RecipeNLG was constructed as the

Meal Image	Ingredient List	Instruction List
	1. chicken 2. garlic 3. butter 4. lemon 5. rosemary 6. basil 7. thyme 8. salt and pepper.	1. Preheat oven to 370F. 2. Rub salt and pepper to the chicken and set aside for around 10 mins. 3. rub half of the butter to the chicken. 4. Mince 1 whole garlic and rub on the chicken. 5. Rub chicken with Herbs. 6. Stuff chicken with lemon, butter, 1 whole garlic, salt, pepper and herbs. 7. Put chicken in oven covered with foil for 45 mins. 8. Remove cover and cook for another 45 mins at 400F
	1. ladyfingers 2. cream cheese 3. lemon juice 4. lemon jelly powder 5. boiling water 6. ice cubes 7. cool whip topping 8. fresh raspberries 9. jelly powder	1. Grease 9 inch (23 cm) springform pan. 2. Place lady fingers around inside rim and set aside 3. Beat cream cheese in large bowl of electric mixer. 4. Add lemon juice and rind, beating on low speed until blended. 5. Dissolve lemon jelly powder in boiling water. 6. Add ice cubes, stirring until slightly thickened. 7. Add lemon jelly slowly to cream cheese mixture while beating. 8. Increase speed and beat just until well blended. 9. etc. ....
	1. strawberries 2. pineapple 3. non-dairy coffee creamer 4. orange juice	1. In blender add strawberries, pineapple, non-dairycreamer and orange juice and blend until smooth. 2. Poor into frosted glass and enjoy

Figure 3.2: Samples of multimodal recipes from Recipe 1M

largest available, open corpus for semi-structured text generation in the culinary domain.[47] Initial fine-tuning verified the effectiveness of this structurally improved corpus, demonstrating that models trained on RecipeNLG produced items with greater average cosine similarity to the gold standard than did models trained on Recipe1M+.[47] The value of this dataset is that it contains high-fidelity textual ground truth, enabling a controlled environment for experimentation with advanced training objectives, such as bespoke loss functions, crafted with the purpose of applying procedural integrity. By concentrating on RecipeNLG, research can safely attribute gains in performance to improved modeling of internal textual logic and structural constraint, separating optimization challenge from the noise of multi-modal data.[47]

3.2.3 Emerging Multimodal and Procedural Benchmarks

The most recent works were focused on developing multimodal datasets that emphasize procedure accuracy. RecipeGen, for instance, presents a step-aligned multimodal dataset that hosts over 21,000 recipes and near-140,000 corresponding images.[52] Its most important contribution is the enforcement of a hierarchical evaluation suite that directly assesses procedure quality by, notably, calculating three significant measures: Cross-Step Consistency, Ingredient Accuracy, and Interaction Faithfulness.[53] These new measures fortify the need to move optimization objectives beyond basic string similarity. They make it official that logical flow and factual fidelity are required, supporting the chief argument that basic cross-entropy loss is lacking. The literature also outlines a key design trade-off in procedural content generation: fidelity (compliance with necessitated constraints and accuracy), on the one hand, and novelty or personalization (culinary originality and customization), on the other.[49] Experiments have remarked that direct imposition of individual constraints, such as allergen safety, is liable to make overall coherence of the recipe decay.[49] As such, an effective custom loss function cannot simply impose an individual constraint but must instead be a judiciously weighted, multi-faceted objective that manages this inevitable dilemma by optimizing structural compliance while preserving creativity divergence. The key characteristics of these datasets are summarized in Table 3.1.

Dataset Name	**Primary Focus**	**Scale (Approx.)**	**Key Structural/-Content Distinction**
Recipe1M+	Multimodal (Image/Text Alignment)	>1 Million Recipes	High frequency of malformed instruction steps, missing fractions, and segmentation issues.
RecipeNLG	Textual Structure and Logic	>1 Million Recipes	Explicitly corrected structure, emphasis on text quality, logic, and consistent entity representation.
RecipeGen	Step-Aligned Multimodal Benchmark	26,435	Focus on Cross-Step Consistency, Ingredient Accuracy, and Semantic-Visual Alignment.

Table 3.1: Key characteristics most common recipe datasets.

3.2.4 Evaluation Metrics for Structured Text Generation

The optimization objective is specified by the evaluation framework. For generative LLMs, unsuitability of generic text metrics for structured, procedural outputs necessitates the design of domain-oriented metrics, and these metrics in turn warrant the employment of custom-designed loss functions during training.

3.2.5 Limitations of Classical N-gram

Traditionally, large models were evaluated with token-overlap scores including BLEU (Bilingual Evaluation Understudy) and ROUGE (Recall-Oriented Understudy for Gisting Evaluation).[49] Although effective for lexical correspondence and measures of fluency (e.g., Perplexity), these scores were widely criticized as being unsuitable for specialist NLG applications. Thorough reviews concluded that BLEU scores, in particular, do not accurately correspond to real-world usability or user satisfaction when used as evaluative criteria for individual generated samples beyond machine translation.[54] Importantly, for procedural generation, N-gram scores can't register the semantic or factual coherence of generated output. A syntactically natural but procedurally unacceptable generated step of a procedure (e.g., "strain the sauce" prior to combining the ingredients) would score highly in BLEU but would be of no practical use. This inappropriateness of high scores in the traditional measures and low practical utility is a common experience in culinary NLG research, and it raises the need for objective functions that directly register functional correctness.[49]

3.2.6 Domain-Specific Recipe Metrics

In order to overcome the shortcomings of traditional assessment, the literature created bespoke automated metrics for procedure properties. Some of them include Ingredient Coverage Tracking, which provides a quantitative score for the optimal use of all input ingredients by verifying that every item in the ingredient list is actually used in the instructions, Step Complexity, which assesses whether individual instructions are simple and atomic, penalizing steps that are overly long or combine too many distinct actions, and Recipe Coherence, which evaluates the high-level logical flow of the recipe to ensure steps are in a sensible order and no critical processes (like preheating an oven) are omitted.[49]

Moving beyond automated proxies, the hierarchical evaluation protocol introduced by RecipeGen demands verifiable procedural assessment based on three levels. The first is Cross-Step Consistency, which rigorously tracks the state of ingredients through the procedure (e.g., an ingredient ’chopped’ in step 2 cannot be ’diced’ in step 4). The second is Ingredient Accuracy, which measures whether the actions applied to ingredients are semantically and factually correct (e.g., the model should instruct to ’boil’ pasta, not ’fry’ it). The third, Interaction Faithfulness, ensures that the text faithfully describes the procedural action or the state of ingredients, which is especially critical in multimodal tasks for aligning text with images.[53] These metrics require the evaluation system to move beyond simple comparison with reference text, instead demanding structured comprehension of the logical and factual integrity of the generated sequence. The existence and adoption of such metrics provide the foundational criteria that the custom loss must internalize. The procedural metrics essentially define the specific elements of generative success that the training objective must maximize.

3.3 Novel Recipe Detection Based on Topological Data Analysis

While our work focuses on improving the generation of recipes with custom loss functions, we are not alone in thinking that the "space" of recipes has a meaningful structure. A highly relevant and innovative study by Escolar et al. [55], titled "A topological analysis of the space of recipes," explores this very idea from a different angle. Their primary goal was to apply a sophisticated mathematical framework called Topological Data Analysis (TDA) to the study of culinary recipes. The core idea was to map the geometric "shape" of all known recipes to find "holes" in this space—regions that are empty but surrounded by existing recipes, representing unexplored opportunities for creating entirely novel dishes.

3.3.1 Mapping the "Shape" of Culinary Space

To describe the food space, the researchers first had to phrase recipes in a language that geometry can understand. Their pipeline comprised the following key stages:

- **Data Representation and Distance**: What they started with was reducing the complexity of recipes all the way down to their very simplest component: their ingredient list. Then each recipe was translated into a vector in the 0-1 space (one-hot encoding), in which each dimension represents a different ingredient in the set. To determine how different any two such recipe vectors were, they employed cosine dissimilarity.

- **Detecting Holes with Persistent Homology**: With recipes now in the form of points in high-dimensional space, the researchers applied an algorithm referred to as persistent homology to examine the structure. The process involves creating a shape referred to as a Vietoris-Rips complex from the points in the data. In layman's terms, the process links recipes that are close enough to one another so that they create a complicated geometric web. Persistent homology then examines the web at varied scales to determine robust, significant "holes" and determine the "representative cycles" in existing recipes that surround them.

- **Developing Original Recipes with Combinatorial Optimization**: The discovery of these holes was more than a theoretical exercise. Escolar et al. utilized this topological data to come up with new recipes. They would take all the distinct ingredients from the recipes making up the cycle around the hole and put them in a candidate pool of ingredients. From this candidate pool, they would then attempt to find a new ingredient combination which was mathematically maximally distinct from all existing recipes in their original dataset. What they were trying to find was some point far in the midst of the "hole" they'd uncovered.

3.3.2 Experimental Verification and Sensory Testing

The interesting part about their research is that they verified the results in the real world. The exploration in the paper validated that the bulk of the ingredient pairs their algorithm generated were, in fact, new and did not exist in the original collection. To take it one step further, they selected some of the new ingredient pairings from their approach, which were among the more unusual ones, such as cream cheese, cranberry, gin, and whole grain wheat flour. From these lists, they came up with and baked multiple sets of biscuits. These fresh biscuits were then used in a sensory evaluation study and presented to a panel. The result from the study confirmed that the biscuits were, in fact, "acceptable enough," which was a good indication that their topologically-motivated ingredient pairings were more than just mathematically new, but culinarily possible.

3.3.3 A Comparison among the Topological Methods

The work of Escolar et al. could not be more timely since it reinforces the essential philosophy behind this thesis: that the application in the recipe space of topological concepts can yield powerful understanding. In any case, our work varies and deepens this idea in several key ways.

3.3.3.1 Contrasting Goals and Methodologies

The main objective of their work is exploratory analysis for the discovery of novelty. Theirs is an offline pipeline operating on a static dataset to suggest a novel recipe. In contrast, the objective of this thesis is to enhance the generative fidelity of a language model. Our method is an training process where a topological measure functions as a loss function in order to direct the learning process of an LLM in real-time.

3.3.3.2 The Critical Difference: Representation and Differentiability

The basic difference is in the way recipes are represented. Their method depends on sparse, non-semantic one-hot vectors for ingredient lists. This is the root cause: the representation can only be discrete and non-differentiable. Due to this fact, their topological analysis cannot be plugged into the gradient-based training loop in a neural net. Our method, described in the chapter on methodology, actually lifts this limitation specifically. By framing the model's predictions in the form of the "soft" probabilistic embeddings from the model's logits, our entire process can be made fully differentiable. It is this key innovation that allows us to frame the geometric distance between point clouds of ingredients in the form of a continuous loss function. The loss can then be optimized with backpropagation, and this permits an end-to-end kind of training that trains the model directly to respect the semantic and topological structure of the ingredient space.

3.4 Why custom loss matters?

The main reason for adopting a custom loss function is to bring about the LLM's generative policy to follow abstract and procedural constraints, which cannot be processed by the usual token-level prediction loss.

3.4.1 Standard SFT vs. Custom Loss Functions

Standard fine-tuning of LLMs utilizes Supervised Fine-Tuning (SFT), reducing the adverse log-likelihood by the Cross-Entropy (CE) loss. SFT is successful in training fluency and grammar by guessing the sequence's following token, but it optimizes automatically for statistical resemblance with training corpus. It is not concerned with properties of factual truthfulness, executability of procedure, or constraint satisfaction by abstraction. For such activities as generating recipes, in which the generated output needs to follow particular, complicated rules (e.g., appropriate ingredient use, appropriate procedure sequence), in-training methods, through losses adjusted during training, intervene by altering the model's parameters as well as its internal policy, so that the training objective specifically includes domain-aware performance measures in addition to typical language modeling objectives.[56]

3.4.2 Metric-Guided Training and Reinforcement Learning from Human Feedback

When standard Cross-Entropy fails for complex procedural tasks like recipe generation, metric-guided training using Reinforcement Learning (RL) offers an alternative [57]. The most successful approach is Reinforcement Learning from Human Feedback (RLHF), which trains a Reward Model (RM) on human preferences to guide LLM finetuning via RL algorithms like PPO, as illustrated in Figure 3.3 [48]. Effective RM design, particularly Process-Supervised RMs offering step-by-step feedback, is crucial for achieving procedural accuracy [57, 48]. While RLHF effectively addresses procedural errors common in standard Supervised Fine-Tuning (SFT) [47, 53, 57], its exploration falls outside the scope of this thesis, which instead focuses on directly augmenting SFT with custom, differentiable loss functions.

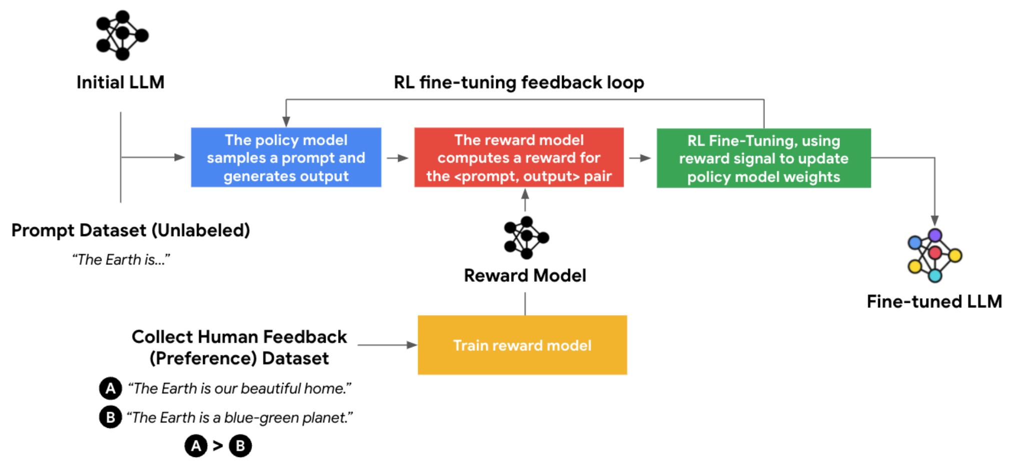

Initial LLM
RL fine-tuning feedback loop
The policy model samples a prompt and generates output
The reward model computes a reward for the <prompt, output> pair
RL Fine-Tuning, using reward signal to update policy model weights
Prompt Dataset (Unlabeled)
"The Earth is..."
Reward Model
Fine-tuned LLM
Collect Human Feedback (Preference) Dataset
Train reward model
A "The Earth is our beautiful home."
B "The Earth is a blue-green planet."
A > B

Figure 3.3: Feedback loop of RLHF finetuning loop.

3.5 Training Objectives vs. Inference Constraints: Two Views Compared

When implementing structural and semantic constraints in generative LLMs, there exist two dominant methodologies: adjusting the training objective (custom loss/RLHF) or constraining at decoding time at inference (constrained decoding). It is imperative to understand the trade-offs behind both strategies in defense of the thesis's focus.

3.5.1 Constrained Decoding Techniques

Constrained decoding (a technique used during the process) alters the token production process of the LLM by controlling the lower-level numerical probability values (logits) of each step [56]. The basic mechanism is "masking," or minimizing the probability of undesirable tokens, such that produced sequences adhere to pre-specified rules. The most constraining form, Grammar-Constrained Decoding (GCD), employs Context-Free Grammars (CFGs) to ensure that the LLM's decoding exactly aligns with particular syntactic rules, including well-formed JSON, XML, or fixed recipe templates [50]. While extremely successful at assuring structural correctness—an imperative for assuring the format of the generated recipes is proper—GCD is computationally not-so-lightweight, frequently necessitating offline precomputation for aligning the subword vocabulary of the LLM with CFG terminals. More advanced variations, for example, NeuroStructural Decoding, make efforts at incorporating syntactic constraint by applying dependency parsing on partial generations for applying richer lexico-syntactic constraints [58].

3.5.2 Trade-offs and Complementarity

The main point of divergence is the strength of constraints they can impose. Constrained decoding is typically limited to imposing syntactic correctness—the form and organization of the text [59]. On the other hand, training-based methods, such as specifically custom loss/RLHF, need to be used for optimizing semantic constraints (factual correctness, coherence, procedural rationality, and high-level correspondence) [60]. Constrained decoding is also computationally cheap at inference time for LLM development, with a straightforward means of ensuring the output format, but it cannot ensure the abstract properties of the content itself [61].

The bespoke loss (in-training) needs access to model weights and much higher training complexity, but it injects the LLM with the internal policy necessary for deep, meaningful alignment with procedure-oriented goals [61]. Crucially, custom loss functions, often drawn from domains like computer vision where structured outputs are common [62], can serve as effective differentiable proxies for complex, non-differentiable evaluation metrics or abstract procedural requirements, enabling direct optimization of these desired qualities during training [63]. For the problem of creating recipes, when fidelity of procedure is paramount (i.e., the generated recipe must work), a custom loss function is the choice that must be made. Constrained decoding can help to ensure that generated text is in the correct form for recipes (e.g., correct list ordering), but only a custom loss, informed by measures of procedure, such as Cross-Step Consistency, is able to penalise semantic errors, such as instructing a user to perform some procedure (such as baking) before completion of necessary prior actions (such as mixing ingredients). While complexity is high, literature suggests optimal performance is achieved with a combination of a hybrid strategy, such that the custom loss reduces abstract, semantic quality, and constrained decoding ensures rigid syntactic fidelity [61]. The trade-offs between these different optimization paradigms are summarized in Table 3.2.

Paradigm	**Mechanism**	**Intervention Stage**	**Strength**	**Limitation**
Supervised Fine-Tuning (SFT)	Cross-Entropy Loss Minimization	Training	Establishes foundational fluency and domain-specific grammar.	Fails to penalize factual errors, procedural inconsistencies, or address abstract quality metrics.
Custom Loss (RLHF or Metric-Guided)	Policy optimized via learned Reward Model (RM) proxy.	Training	Aligns generation with complex, abstract, task-specific metrics (Coherence, Faithfulness, Ingredient Accuracy).	High training complexity; requires annotated preference data; guarantees are probabilistic, not absolute.
Constrained Decoding (GCD)	Token Logit Masking (CFG, RegEx).	Inference (Intra-processing)	Guarantees syntactically valid and strictly structured output formats (e.g., valid instruction/ingredient list structure).	Cannot enforce semantic, procedural, or factual rules; limited to hard syntactic constraints.

Table 3.2: Key characteristics and trade-offs of three optimization paradigms for structured text generation.

3.6 Gap in Literature and Contribution

The comparison with related work illustrates that although Culinary NLG based on transformer architecture is the norm for Large Language Models[46], and high-accuracy text datasets such as RecipeNLG are available[47], it is still a challenge to match the generative policy with the stringent requirements of procedural fidelity. Classical SFT, as fine-tuned through cross-entropy loss, fails inherently to condemn procedural and factual mistakes and is thus shown in early attempts at research to introduce inconsistencies.[47] Additionally, cutting-edge evaluation requires procedure-verifiable measures such as Ingredient Accuracy and Cross-Step Consistency.[53] The literature also strongly suggests the use of high-level measurable quality attribute and learnable model's parameters through advanced training objectives, i.e., for example, the RLHF-based ones. In particular, for procedural tasks, it is confirmed as necessary the design of Process-Supervised Reward Model.[57] This thesis bridges the key literature gap by being the first to rigorously apply a specially-designed loss function, guided by the procedural evaluation metrics, to fine-tuning an LLM on structurally enhanced RecipeNLG corpus. This work directly addresses empirically-verifiable results such as Ingredient Accuracy and Cross-Step Consistency, and offers an foundational work in applying metric-driven policy optimization to highly-constrained, practical, procedural text generation. The methodology established here is likely to form the basis for further advanced adaptation strategies, such as multi-task learning models that simultaneously optimize for culinary finesse and intricate targets such as allergen safety.[49]

Chapter 4

Methodology

This chapter aims to describe how the dataset has been selected and preprocessed, the choice of the models, the choice of the custom loss paired to the CE, the choice of the evaluation metrics, and how the experiments have been performed.

4.1 Problem statement

Creating accurate cooking recipes is a notable challenge for Large Language Models (LLMs) because these tasks demand precise procedural knowledge and numerical reasoning. This study seeks to enhance the effectiveness of LLMs in tackling this specialized task. While traditional fine-tuning methodologies are effective for general-purpose text, they demonstrate considerable limitations when applied to procedural and niche domains like cooking recipes.

More formally, we define the task of structured recipe generation as a mapping $f : P_{in} \to R_{out}$, where P_{in} is a natural language prompt (e.g., "Generate a recipe for Pasta Carbonara") and R_{out} is a structured text output that must conform to a specific JSON schema. This output $R_{out} = \{\boldsymbol{I}, \boldsymbol{S}\}$ consists of two key components:

- A list of ingredients $\boldsymbol{I} = \{I_1, I_2, \ldots, I_n\}$, where each I_j is a string.
- A list of instructions $\boldsymbol{S} = \{S_1, S_2, \ldots, S_m\}$, where each S_k is a string.

The objective is to learn a function f that successfully satisfies multiple constraints simultaneously. These include not only structural correctness (i.e., generating valid JSON) but also factual, procedural, and numerical accuracy. For example, for the prompt "Pasta Carbonara," the model must satisfy:

- **Factual Constraints:** The ingredient list $\boldsymbol{I}$ must be factually correct and complete, containing items like 'guanciale', 'pecorino', 'eggs', and 'pepper'.
- **Numerical Constraints:** The entities within $\boldsymbol{I}$ (e.g., quantities like "200g") and $\boldsymbol{S}$ (e.g., times like "10 minutes") must be plausible and accurate.
- **Procedural Constraints:** The sequence of instructions $\boldsymbol{S}$ must be logical and executable (e.g., the guanciale must be cooked before being mixed with the pasta).

The fundamental challenge of this task lies in the inherent asymmetry of token importance. In a recipe, tokens are not of equal value; the generation is governed by key entities. Recipes are carefully structured guides where "key players"—such as ingredients, their amounts, and cooking steps—are far more crucial than the connective words. This deficiency manifests in common failure modes, such as poor ingredient recall, inaccurate numerical values, and procedurally incorrect instructions, which ultimately render a generated recipe unusable.

Recognizing that success in this domain is defined by a combination of fluency, structural integrity, and factual correctness, this study assumes that a standard training objective is insufficient. Our main objective is to create a finetuning framework that enhances the standard Cross-Entropy loss by incorporating additional, specialized loss functions. These secondary losses are designed to explicitly target and amplify the learning signal for the most essential elements of a recipe, forcing the model to pay closer attention to ingredients and numerical values. This work, therefore, focuses on the loss function as the most direct lever for influencing model behavior to solve the limitations of the training process.

4.2 Dataset preparation

The dataset consists of two main parts:

- Recipes
- Questions related to general cooking knowledge and skills

4.2.1 Recipes

The first part has been obtained as a selection of recipes from the dataset RECIPE-NLG [47], a large-scale corpus of over 2.2 million cooking recipes developed for natural language processing (NLP) and semi-structured text generation tasks. It was created to address the limitations of previous recipe datasets, such as Recipe1M+, which were often designed with computer vision applications in mind and suffered from structural inconsistencies. We have selected a subset of 4000 recipes related to pasta, rice, and sandwiches. This specific subset was chosen because these dishes are not only highly common but also tend to share similar preparation structures and ingredient types, providing a coherent and focused domain for specialization This selection is motivated by the fact that preliminary results based on fine-tuning the model on a subset representative of the entire corpus led to poor results in ingredient recall, caused by the limited training sample size and time for training. We believe this was caused not only by the limited training time but, more importantly, by a significant distributional mismatch between the broad training data and the test set. To draw a parallel, a model trained broadly might not perform well on specific categories, much like a chef who only knows how to cook rice-based dishes would struggle to adapt to fish-based dishes. Therefore, to ensure a more robust validation by aligning the train and test distributions, we decided to limit the dataset to these specific first-course recipes, specializing the model within this category. Besides the better consistency of RECIPE-NLG compared to Recipe1M+, we opted for this dataset since it provides both a list of ingredients and instructions which is well suited for structured output as JSON, which we have chosen to perform a smoother evaluation of the models after fine-tuning.

4.2.2 General Questions

In addition to the recipe collection, we manually curated and added a new dataset of 235 questions related to general knowledge in the cooking domain. We decided to augment the dataset with this set of questions to allow the model not only to learn how write recipes in general (which is a task already solved by means of most of vanilla LLM), but also to allow the model to understand how to better deal with quantities, ingredients similarities and differences, ingredients proportions, and other challenges related to recipe generation. The questions have been generated from a set of humanly written samples that have been augmented using Gemini 2.0 Flash [64]. The resulting augmented set has been then evaluated by humans for the quality of questions, diversity in recipes and ingredients involved, and potential data leaks in relation to the test dataset. The questions can be categorized into the following.

- Missing ingredient identification
- Substitution validation
- Recipe scaling
- Quantity ingredient
- Time
- Temperature

4.2.3 Missing ingredient identification

The following questions consist of the presentation of recipes along with an incomplete list of ingredients. The questions evaluate therefore the capacity of the model to identify the missing ingredients and learn the relationships between the ingredients.

Example

Question: You are an expert chef. You have been tasked with generating a recipe for Carbonara. The following ingredients have been identified: pasta, pecorino, eggs. Which ingredient(s) are missing?

Answer: pepper, guanciale

4.2.4 Substitution validation

This set of questions has been inserted to teach the model about similarity and differences about ingredients. Our aim with this set of questions is to teach the model how to deal with uncertainty about new recipes and, in case the recipes contains not the exact ingredients, that at least similar candidates are contained.

Example

> **Question**: You are an expert chef. You have been tasked with generating a recipe for Carbonara. Pecorino is not available and you have been asked to propose an alternative ingredient. Which ingredient can best substitute it?
>
> **Answer**: Parmesan

4.2.5 Recipe scaling

One of the common problems of SLM (Small Language Models) is their inability to predict reliable quantities for ingredients, especially when the portions required are different from the one in the training dataset. The goal of this subset of questions is to train the model in identifying the scaling rules that are used for different kinds of ingredients, in particular the differences between main ingredients (for example, carbohydrates, protein, fibers) and condiments (for example, oil,butter,salt, herbs).

Example

> **Question**: You are an expert chef. You have been tasked to modify the following list of ingredients, aimed to be used for two servings, so that it can be used for 4 servings. Modify the quantities of each ingredient. List of ingredients: 200g of pasta, 100g of pecorino, 3 yolks, 140g of guanciale.
>
> **Answer**: 400g of pasta, 100g of pecorino, 3 yolks, 140g of guanciale.

4.2.6 Quantity Ingredient

In relation to the previous set of questions, this set aims to teach the model how to properly dose ingredients based on the context.

Example

> **Question**: You are an expert chef. How many grams of spaghetti are needed to prepare a Carbonara for two people?
>
> **Answer**: 400g

4.2.7 Time

Recipes not only consist of ingredients, but also of a series of steps to follow in order to accomplish the final dish. One of the key attributes of these steps is the time needed to execute that specific step. This set of questions aims to teach the model how to properly define times in relation to the different steps involved.

Example

> **Question**: You are an expert chef. How much time do you need to boil pasta?
>
> **Answer**: 10 minutes.

4.2.8 Temperature

In relation to the previous point, it is also important that the model learns, whenever necessary, the temperature at which certain ingredients should be exposed in order to accomplish a specific step. This set of questions aims to do so.

Example

> **Question**: You are an expert chef. At which temperature should you boil pasta?
>
> **Answer**: 100 Celsius degrees.

4.3 Dataset preprocessing

Both sections of the dataset have been saved into a unique file in csv format. In order to be utilized for finetuning, a series of steps have been applied to both of them.

4.3.1 Preprocessing of Recipe dataset

Recipe-NLG contains recipes expressed in a convenient format, as they already contain two fields called "ingredients" and "instructions" containing a list of the respective entities. The main aspect that needed to be modified for our experiment is the conversion of the unit of measure of the ingredients and other entities to the metric systems. We decided to convert to the metric system to have reliable and consistent numerical values related to ingredient quantities, time and temperature. In order to do so, we opted for using an LLM (Gemini 2.0 Flash) to convert all quantities to the metric system. Finally, the recipes have been saved in JSON format. The final result can be seen in the following example:

Recipe Example

```
{
  "ingredients": [
    "200g Guanciale, cubed",
    "4 large egg yolks",
    "50g Pecorino Romano cheese, grated",
    "320g Spaghetti",
    "Coarsely ground black pepper",
    "Salt"
  ],
  "instructions": [
    "Boil salted water in a large pot.",
    "Fry the guanciale in a skillet until crispy.",
    "Remove the skillet from the heat.",
    "Combine egg yolks, Pecorino, and pepper in a bowl.",
    "Garnish with extra cheese and pepper.",
    "Serve immediately."
  ]
}
```

4.3.2 Preprocessing of Cooking questions

Since these questions have been generated through LLM augmentation, we had more flexibility in shaping the samples in the format desired. In this case, questions have been saved in JSON format containing one field for the question and one for the answer.

Cooking Question Example

```
{
  "question": "You are an expert chef. What is the temperature in
    Celsius to boil water?"
  "answer": "100"
}
```

4.3.3 Conversion to Huggingface dataset format

In order to be utilized, both sections of the dataset have been converted to the "message" template format of HuggingFace. Regardless of the type of task (recipe,question), the finetuning sample has been converted from their respective JSON format to a message template comprising of a user request and an assistant answer.

Example of row in HuggingFace dataset format

```
[
  {
    "role": "user",
    "content": "How many minutes should you boil an egg for a jammy
    yolk?"
  },
  {
    "role": "assistant",
    "content": "7"
  }
]
```

4.4 Losses

This section describes the loss functions utilized in the experiment section.

4.4.1 Loss functions

4.4.1.1 Cross-Entropy

Cross-Entropy (CE) is a widely used method for training large language models on generative tasks due to its effectiveness in optimizing model performance. Given its effectiveness in optimizing model performance, we have decided to use the results from fine-tuning with pure Cross-Entropy as our primary baseline, as it provides a standard method for comparison.

Cross-Entropy is used to measure the dissimilarity between the model's predicted probability distribution and the ground-truth distribution. The model produces a vector of raw scores, known as logits (z), which represent the unnormalized probabilities for every token in its vocabulary, V. These scores are then converted into a probability distribution, p, using the softmax function:

$$p_j = \frac{e^{z_j}}{\sum_{i=1}^{|V|} e^{z_i}}$$

where p_j is the predicted probability for the j-th token. The ground truth is represented as a one-hot encoded vector, y, where the element corresponding to the correct token is 1 and all others are 0. The Cross-Entropy loss, $H(y, p)$, is then formally calculated as:

$$H(y, p) = -\sum_{i=1}^{|V|} y_i \log(p_i)$$

This formula simplifies to calculating the negative log-probability of the single correct token, c, because in a one-hot encoded vector y, only the correct token has a value of 1, while all others are 0. Therefore, we can rewrite the formula as:

$$L_{CE} = -\log(p_c)$$

This resulting loss value is low if the model assigned a high probability to the correct token (i.e., p_c is close to 1), but it grows larger as the predicted probability for the correct token approaches zero. During fine-tuning, the objective of the model is to adjust its internal parameters to minimize this loss value across the training data. Building on its foundational role, in this research, Cross-Entropy is used in two key ways. First, it establishes our baseline performance level. Second, it acts as a core component in all of our custom loss experiments, where it is combined with the novel loss functions in a 60/40 weighted average. This highlights its significance as both a standard for comparison and a stabilizing element, which is important for the effectiveness of our more advanced loss formulations.

4.4.1.2 Focal

One of the primary challenge in the fine-tuning of our language model for recipe generation is the imbalance in token frequency within the training corpus. Standard training with a Cross-Entropy loss function results in learning extremely common words and phrases (e.g., "add a pinch of," "mix until combined," "in a large bowl"), as correctly predicting these high-frequency tokens produces a quick reduction in the overall loss. This result in a model that is proficient at generating generic cooking recipes but fails to master the ingredients that define specific dishes, such as distinguishing between "pancetta" and "guanciale" in a pasta recipe or suggesting "basmati" for a rice pilaf, as well as correctly generating the related quantities. In order to mitigate this, we used Focal Loss, originally proposed by Lin et al. (2017). While this loss is mostly implemented in computer vision tasks, its core principle is directly applicable to addressing token-level imbalance in language modeling. The Focal Loss is a dynamic extension of the standard Cross-Entropy (CE) loss. In the context of an LLM, the CE loss for predicting a target token is:

$$CE(p_t) = -\log(p_t)$$

where p_t is the model's predicted probability of the correct next token. The main issue is that the loss for correctly predicting the common token "the" is treated with the same importance as the loss for correctly predicting the crucial ingredient "saffron." Focal Loss addresses this by introducing a modulating factor $(1 - p_t)^\gamma$ to the CE loss:

$$FL(p_t) = -(1 - p_t)^\gamma \log(p_t)$$

This factor has two major consequences. When the model correctly predicts a very common token with high probability (an "easy" prediction, $p_t \to 1$), the $(1-p_t)^\gamma$ term approaches zero, effectively silencing the loss for this trivial prediction. In contrast, when the model is uncertain about a rare but critical token, such as a specific type of cheese for a sandwich or a particular herb for a rice dish (p_t is low), the modulating factor is close to 1. This makes sure that the model is significantly penalized for failing to learn the important, low-frequency vocabulary. The focusing parameter, $\gamma \geq 0$, controls the strength of this effect. By setting $\gamma = 2$ in our experiments, we encouraged the model to focus its learning capacity on the challenging and defining tokens of our recipe dataset, leading to the generation of more accurate, diverse, and high-quality recipes.

4.4.1.3 Dice

Standard loss functions such as Cross-Entropy concentrate on predicting the correct sequence of words one by one, we also integrated the Dice Loss to train the model to have a more holistic understanding of a recipe's success. This methodology draws inspiration from techniques used in medical image analysis, but we've adapted its core concept to our task. Rather than penalizing every minor wording deviation, Dice Loss encourages the model to produce the correct set of key ingredients and instructions, bringing it closer to how a human chef thinks.

The Dice Loss is derived from the Sørensen–Dice Coefficient (DSC), a statistic developed to measure the similarity between two sets. In essence, it measures the overlap between the set of keywords in the generated recipe (A) and the set in the reference recipe (B):

$$DSC = \frac{2|A \cap B|}{|A| + |B|}$$

The formula is applied in a differentiable form in order to be applied to a neural network, since they operate on continuous outputs instead of discrete sets.

Let the ground truth be represented by a one-hot vector $\mathbf{y}$ of vocabulary size V, where the single non-zero element corresponds to the correct token. Let the model's prediction be the probability vector $\hat{\mathbf{y}}$ produced by the final softmax layer, also of size V. The intersection term $|A \cap B|$ is approximated by the dot product of these two vectors, $\sum_{i=1}^{V} \hat{y}_i y_i$. This effectively isolates the model's predicted probability for the single correct token. The size of the sets, $|A|$ and $|B|$, are approximated by the sum of the squares of their vector elements, $\sum_{i=1}^{V} y_i^2$ and $\sum_{i=1}^{V} \hat{y}_i^2$ respectively. This leads to the differentiable formulation of the Dice Score for a single prediction:

$$DSC_{soft} = \frac{2\sum_{i=1}^{V} \hat{y}_i y_i + \epsilon}{\sum_{i=1}^{V} \hat{y}_i^2 + \sum_{i=1}^{V} y_i^2 + \epsilon}$$

Here, ϵ is a small smoothing constant that ensures numerical stability and prevents division by zero, especially in the early stages of training. The final Dice Loss is then defined as $L_{Dice} = 1 - DSC_{soft}$.

The minimization of this loss implies the direct optimization of the alignment between the model's predicted probability distribution and the ground truth, compelling it to accurately capture the essential semantic components of a recipe.

4.4.1.4 Topological

One of the main challenges in the generation of recipes consists of understanding the relationships between ingredients. While standard Cross-Entropy (CE) loss optimizes for token-level prediction accuracy, it does not inherently grasp the semantic or geometric relationships between them. For instance, substituting "pecorino" with "parmesan" is a semantically plausible error that CE loss would penalize just as heavily as substituting it with an unrelated ingredient, like "chocolate". To solve this, we introduce a topological-based loss function designed to operate in the embedding space which will complemented the CE loss.

The core principle of this loss is to represent the list of ingredients not as a sequence of tokens, but as a "point cloud" in a high-dimensional space, where each point is an ingredient's embedding. The loss then measures the geometric dissimilarity between the point cloud generated from the model's prediction and the point cloud from the ground truth. The intuition is that a semantically correct, although not perfectly identical, list of ingredients will form a shape geometrically similar to the ground truth in the embedding space. The loss is implemented through a three-step process:

1. generating differentiable representations of the model's predictions;
2. generating the point clouds;
3. calculating the geometric distance between the two point clouds.

4.4.1.5 Step 1: Generate Soft Probabilistic embeddings

Differentiability is a fundamental requirement for any loss function and a simple argmax over the model's output logits to select the predicted token is a non-differentiable operation therefore, to overcome this, we formulate a "soft" or "probabilistic" embedding for each predicted ingredient token. For a given token position, we apply a softmax function to the model's raw logit vector, which converts the scores into a probability distribution, P, over the entire vocabulary V.

$$P = \text{softmax}(\text{logits}), \quad \text{where } P \in \mathbb{R}^{|V|}$$

Next, we perform a matrix multiplication of this probability vector with the model's token embedding matrix, $E \in \mathbb{R}^{|V| \times d_{emb}}$, where d_{emb} is the dimension of the embeddings.

$$emb_{soft} = P \cdot E$$

The resulting emb_{soft} is a weighted average of all token embeddings in the vocabulary, with the weights being the model's predicted probabilities. For example, if the model is 80% confident the ingredient is "sugar" and 20% confident it is "flour," the resulting soft embedding will be a vector located geometrically much closer to the embedding for "sugar" but pulled slightly towards the one representing "flour". This process yields a fully differentiable representation of the model's prediction for that position. Figure **??** illustrates this process of generating a soft probabilistic embedding.

4.4.1.6 Step 2: Point Cloud Creation

With a method to derive differentiable embeddings, we are able to construct two distinct point clouds for comparison. The process is focused exclusively on the tokens identified as ingredients, using a pre-computed `ingredients_mask` to select the relevant positions. Follows a description of the two point clouds:

- **Predicted Point Cloud (PC_{pred})**: This cloud is formed by applying the soft embedding process described above to the logit outputs at each ingredient position, representing the model's probabilistic prediction of the entire ingredient list in the embedding space.
- **Ground-Truth Point Cloud (PC_{gt})**: For the ground truth, we simply perform a standard embedding lookup for the label tokens at the ingredient

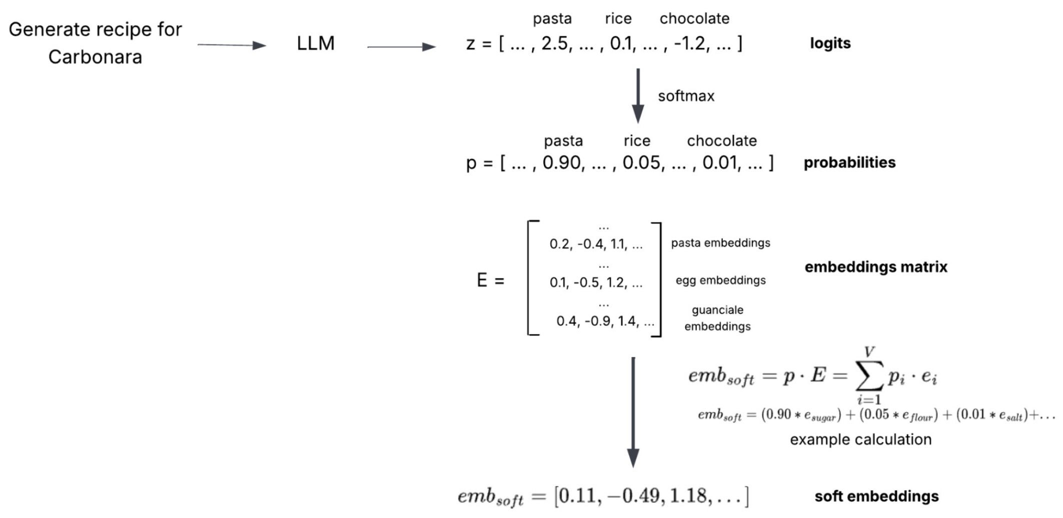

Generate recipe for Carbonara
LLM
pasta rice chocolate
z = [... , 2.5, ... , 0.1, ... , -1.2, ...]
logits
softmax
pasta rice chocolate
p = [... , 0.90, ... , 0.05, ... , 0.01, ...]
probabilities
E =
0.2, -0.4, 1.1, ...
0.1, -0.5, 1.2, ...
0.4, -0.9, 1.4, ...
pasta embeddings
egg embeddings
guanciale embeddings
embeddings matrix
$emb_{soft} = p \cdot E = \sum_{i=1}^{V} p_i \cdot e_i$
$emb_{soft} = (0.90 * e_{sugar}) + (0.05 * e_{flour}) + (0.01 * e_{salt}) + \ldots$
example calculation
$emb_{soft} = [0.11, -0.49, 1.18, \ldots]$
soft embeddings

Figure 4.1: The process of generating a Soft Probabilistic Embedding (emb_{soft}). The model's raw output logits are passed through a softmax function to create a probability distribution (P) over the vocabulary. This vector is then multiplied by the full embedding matrix (E). The resulting emb_{soft} is a weighted average of all token embeddings, yielding a continuous, fully differentiable representation of the model's prediction.

positions. This is the "hard" point cloud, representing the ground truth. The creation of these two comparative point clouds is visualized in Figure 4.2.

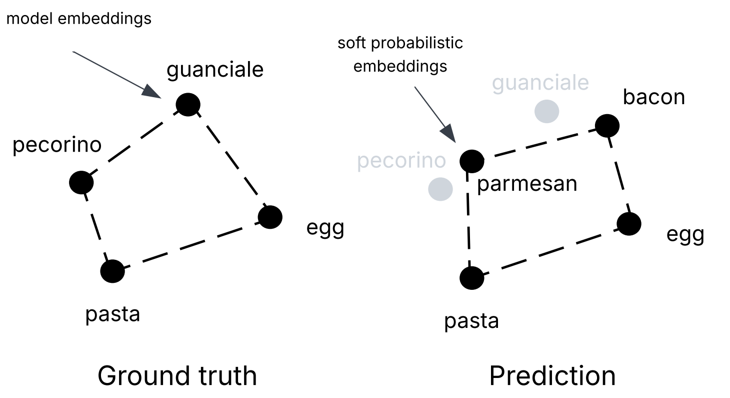


Figure 4.2: The creation of two comparative point clouds in the embedding space for ingredient tokens. The Ground-Truth Point Cloud ($\mathbf{PC_{gt}}$) is formed by hard embedding lookups, while the Predicted Point Cloud ($\mathbf{PC_{pred}}$) uses the Soft Probabilistic Embeddings. The topological loss minimizes the distance between these two clouds by forcing the soft prediction points (light gray) to align with the ground truth points (black).

4.4.1.7 Step 3: Measuring Geometric Distance with Sinkhorn Divergence

The final step consists of the quantification of the dissimilarity between PC_{pred} and PC_{gt}. We utilized the Sinkhorn divergence, a computationally efficient and differentiable approximation of the Wasserstein distance, also known as the Earth Mover's Distance. The choice of Sinkhorn divergence over the classical Wasserstein distance is motivated by several key factors essential for deep learning applications:

- **Computational Speed**: Standard Wasserstein distance involves solving a linear programming problem, which is computationally prohibitive for the iterative nature of model training. Sinkhorn divergence utilizes an entropic regularization term which allows a faster iterative matrix-scaling algorithm.
- **Differentiability**: The entropic regularization ensures the objective is strictly convex, leading to a unique solution and stable gradients essential for back-propagation.
- **Regularization**: The regularization strength, ϵ, acts as a tunable hyper-parameter that smooths the loss landscape which may lead to more stable training.

The Sinkhorn divergence stems from the regularized optimal transport cost, $W_\epsilon(\alpha, \beta)$, between two distributions α (our PC_{gt}) and β (our PC_{pred}):

$$W_\epsilon(\alpha, \beta) = \min_{P \in \mathcal{U}(\alpha,\beta)} (\langle P, C\rangle - \epsilon H(P))$$

Where C is the cost matrix (typically the squared Euclidean distance between embedding points), P is the transport plan, and $\epsilon H(P)$ is the entropic regularization term. The Sinkhorn divergence, S_ϵ, then debiases this value to ensure the divergence of a distribution with itself is zero:

$$S_\epsilon(\alpha, \beta) = W_\epsilon(\alpha, \beta) - \frac{1}{2} W_\epsilon(\alpha, \alpha) - \frac{1}{2} W_\epsilon(\beta, \beta).$$

Our topological loss consists of the above divergence formula. By minimizing this geometric distance, we encourage the model not only to predict the correct tokens, but to generate ingredient lists that are semantically and structurally coherent in the embedding space. Figure 4.3 illustrates this measurement of geometric distance using Sinkhorn divergence.

4.5 Model selection

We have selected three specific language models: Qwen2.5 - 1.5B, SmolLM-3B, and Qwen3 - 4B to perform our experiments. This decision has been taken by considering our necessity for efficient experimentation. To ensure that we could run and evaluate many tests quickly, we deliberately chose models with fewer than 5 billion parameters. Besides speed, this selection was designed to see how well our methods would work on different types of models. using models with different architectures (Qwen versus SmolLM) and of varying sizes (1.5, 3, and 4 billion parameters) allowed us to better understand if our findings are broadly applicable rather than being specific to just one model architecture or scale.

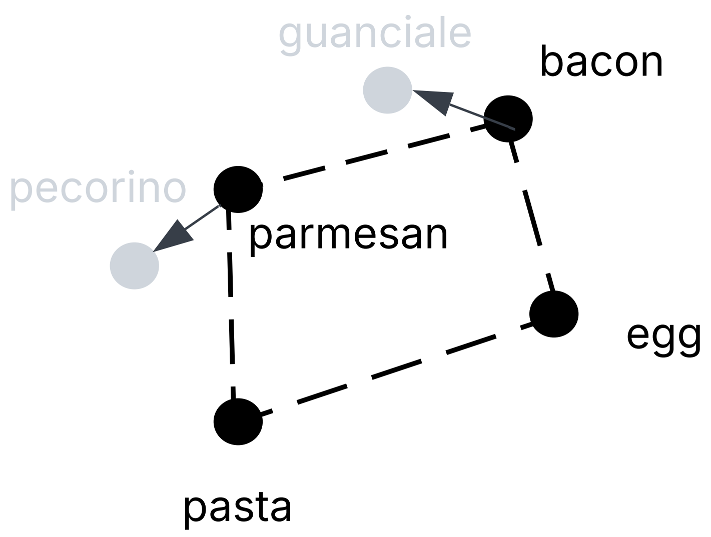


Figure 4.3: Measuring the geometric distance between the Predicted ($\mathbf{PC_{pred}}$) and Ground-Truth ($\mathbf{PC_{gt}}$) point clouds using **Sinkhorn Divergence**. This metric calculates the regularized optimal transport cost (Earth Mover's Distance) required to move the predicted distribution (black points) onto the ground truth distribution (gray points), providing a differentiable loss signal for the model.

4.6 Experiments

The goal of the following set of experiments has been the impact of various custom loss function on linguistic and numerical properties of recipes, as well as the correct usage of ingredients. All losses functions have been tested with the three language models specified in the model section:

- Qwen2.5 - 1.5B
- SmolLM-3B
- Qwen3 - 4B

All experiments were conducted using the Low-Rank Adaptation (LoRA) methodology, which significantly reduces the number of trainable parameters and memory requirements, to ensure that the finetuning process was computationally feasible. We established two key benchmarks to contextualize our results. The performance of the non-finetuned, "vanilla" models serves as a lower bound, while the primary baseline for comparison consists of models finetuned using a pure Cross-Entropy (CE) loss on the recipe generation task. We conducted a series of experiments where each model was finetuned with a composite loss function. These setups combined the standard CE loss with one or more of the custom losses introduced earlier. For the experiments with a single custom loss, the final loss value was a weighted average of 60% CE and 40% custom loss. The 60-40 proportion was decided empirically, as considering this factor as an additional hyperparameter would have notably increased the number of experiments. A final "Mixed Loss" experiment was also conducted to test a combination of the two most promising custom losses. The full list of experimental configurations for each model was as follows::

- Base model (not finetuned);
- Finetuning with 100% Cross-Entropy loss;
- Finetuning with 60% CE + 40% Focal loss
- Finetuning with 60% CE + 40% Dice loss
- Finetuning with 60% CE + 40% Topological loss
- Finetuning with 60% CE + 20% Dice loss + 20% Topological loss

Composite loss functions have been used on samples regarding:

- Recipe generation
- Missing ingredient identification
- Substitution validation
- Recipe scaling

For samples regarding pure numerical predictions (for example, temperature, time and ingredient quantity), we utilized MSE as loss function. All training procedure have been optimized keeping into account computational efficiency. While a comprehensive set of hyperparameters was used, several were particularly important for managing resource constraints:

- **Mixed-Precision Training (bf16: True)**: We utilized bfloat16 mixed-precision training to represents numbers with 16 bits instead of the standard 32, effectively halving the GPU memory required for model weights and gradients. This leads to a significant increase in training speed with minimal impact on final model performance.
- **Gradient Accumulation (gradient_accumulation_steps: 3)**: To achieve the stability of a larger batch size without the associated memory cost, we used gradient accumulation. While the batch size per device was small (per_device_train_batch_size: 2), gradients were accumulated over 3 steps before a model weight update was performed. This simulates a larger effective batch size of 6, leading to more stable convergence during training.
- **Optimizer (optim: adamw_torch)**: The adamw_torch optimizer was selected for its efficiency and robust performance in training large language models. The learning rate was set to 1e-4 across all experiments, and training was conducted for a total of 2 epochs, as we noticed that further epochs were not substantially increasing performances.

4.7 Evaluation and metrics

This section describes the evaluation framework that has been adopted, in particular the metrics that have been utilized and arrangements applied in order to mitigate potential issues.

The evaluation consisted of the generation of 1000 recipes in JSON format, following the schema introduced in the dataset section. While generating a recipes as a simple text could have been more straightforward, we decided to utilize a structured output to facilitate the extraction and evaluation of ingredients and the steps of the recipes. One of the challenges that we immediately faced was the inconsistency in the JSON format, often resulting in unparsable content. To address this limitation, we opted for the usage of "outlines", an open-source library utilized by organization such as NVIDIA,VLLM, Huggingface to enforce a specific schema at inference time. The library offers methods to pass different kind of schemas to a language model, as well as methods for validation and automatic retrial in case of failure. The adoption of this tool allowed us to completely remove the problem of corrupted and incorrect JSON structures. Following this, the output of the model has been passed to NER realized by means of a Small Language Model (Qwen3-4B) to extract attributes from the ingredients (quantity, unit of measure) and instructions (main action, temperature, time). We decided to opt for the application of NER after model inference instead of expanding the JSON schema because we noticed worse performance and instability of the output when involving a large number of attributes in the JSON schema. The usage of NER allowed us to keep simple the JSON structure of the output while allowing us to extract all necessary attributes from the model response. This NER process is illustrated in Figure **??**.

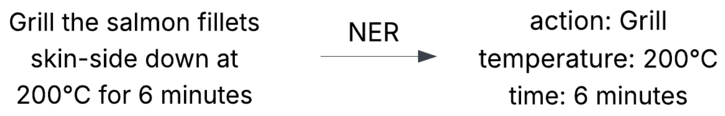

Grill the salmon fillets skin-side down at 200°C for 6 minutes
NER
action: Grill
temperature: 200°C
time: 6 minutes

Figure 4.4: Illustration of the **Named Entity Recognition (NER)** process applied to extract structured attributes from a natural language instruction. After generating recipes in a simplified JSON format, a dedicated Small Language Model (Qwen3-4B) performs NER to extract granular details such as the main action, temperature, and time. This approach maintains a simple JSON schema for the primary output while still allowing for detailed attribute extraction, addressing challenges with model stability when embedding too many attributes directly into the initial JSON generation.

In order to comprehensively assess the performance of our models, we designed an evaluation framework consisting of both metrics commonly found in literature as well as ad-hoc ones designed for these experiments. Standard language generation metrics are insufficient for a task like recipe generation, where factual accuracy and procedural correctness are as important as linguistic fluency. Consequently, our framework combines established NLP benchmarks with a suite of custom, ad-hoc metrics specifically tailored to the nuances of recipes. This dual approach allows for a comprehensive understanding of model capabilities, from semantic coherence to recall of ingredients and numerical precision. The following metrics are commonly used to evaluate the quality of machine-generated text by comparing it to a human-written text.

4.7.1 Traditional metrics

4.7.1.1 BLEU (Bilingual Evaluation Understudy)

BLEU measures the precision of n-grams (contiguous sequences of n words) in the generated text relative to the reference text and it was originally developed for machine translation. BLEU considers the proportion of matching n-grams (typically from 1 to 4) and applies a "brevity penalty", which penalizes generated pieces of texts that are much shorter than the reference one. BLEU serves therefore as a proxy to measure grammatical correctness and fluency. A high BLEU score means that the model's sentence structure and word choices are similar to the ground truth from a style perspective.

4.7.1.2 ROUGE (Recall-Oriented Understudy for Gisting Evaluation)

While BLEU is precision-focused, ROUGE is recall-focused. It measures how many of the n-grams in the reference text appear in the generated output. We specifically utilize ROUGE-1, which focuses on the overlap of individual words (unigrams). ROUGE-1 is particularly useful for evaluating if key terms and concepts from the original recipe are present in the model's output regardless of the exact sentence construction.

4.7.1.3 BERTScore

BERTScore uses contextual embeddings from a pre-trained BERT model to represent each token in the generated and reference texts and it computes the cosine similarity between token embeddings, matching words based on their meaning in context. Differently from n-gram based method, it evaluates semantic similarity. A high BERTScore implies that the generated recipe is semantically close to the ground truth, even if it uses a different phrasing (e.g., correctly identifying "finely chop" as similar to "mince").

4.7.2 Ad-Hoc Metrics for Recipe Generation

In order to mitigate the limitation of standard metrics in evaluating the quality of cooking recipes, we developed a set of custom metrics which target the most critical components of a recipe.

4.7.2.1 Ingredient Recall

It measures the fraction of ingredients from the ground-truth recipe that are correctly listed in the generated output. It is calculated as the size of the intersection of the two ingredient sets divided by the total number of ingredients in the ground-truth recipe. A low score indicates the incapacity of the model of understanding which ingredients are important for a specific recipe.

4.7.2.2 Quantity Precision

This metric measures whether a model is able to handle the numerical quantities associated with the correctly recalled ingredients. For instance, if the model correctly recalls "flour" but generates "100g" instead of the correct "200g," it is penalized. This metric is crucial as incorrect quantities can make a recipe inedible.

4.7.2.3 Action Precision

This metric evaluates the procedural correctness of the instructions. In other words, it measures the precision of key cooking verbs (e.g., "boil," "fry," "sauté," "bake") in the generated steps compared to the reference. It is fundamental to identify the set of actions in order to execute a recipe correctly.

4.7.2.4 Time and Temperature Precision

Similar to quantity precision, these metrics focus on the model's ability to handle numerical values within the instructions, in this case temperature and time. They specifically extract and compare all mentions of cooking durations (e.g., "15 minutes") and temperatures (e.g., "180°C") against the ground truth. The accuracy of these parameters is important to execute the recipe successfully.

4.7.2.5 Action Edit Distance

To quantitatively assess the procedural accuracy of the generated instructions, we have devised a metric based on the concept of edit distance. After extracting the sequence of primary cooking actions—such as 'boil,' 'fry,' or 'mix'—from both the predicted and ground-truth recipe, each unique action is then mapped to a distinct character; This process transforms the entire sequence of instructions of a recipe into a string. The Levenshtein distance is calculated between on top of the two strings to determine the minimum number of actions (insertions, deletions, or substitutions) required to align the two strings. The resulting score measures the differences in the core cooking steps, where a lower distance indicates a more accurate sequence of actions.

4.7.2.6 Step Edit Distance

Built on top of the previous concept, the Step Edit Distance provides a more comprehensive and granular evaluation of the accuracy of the entire sequence of instructions. This approach considers not only the cooking actions but also their critical parameters, such as specified cooking times and temperatures. In this method, the encoding for each step is more descriptive, meaning that a step like "bake for 30 minutes" is considered fundamentally different from "bake for 45 minutes." Therefore, this metric penalizes a wider range of procedural errors, keeping into account details that determine the success of the final dish and offering a more comprehensive measure of a recipe’s correctness.

Chapter 5

Results

This chapter presents the results obtained from the application of the different losses function previously defined to three Small Language Models. The results are presented in three sections: traditional evaluation metrics (BLEU, ROUGE, BERTScore), ad-hoc metrics for recipe generations (precision and recall of different recipe attributes defined in the evaluation section) and human evaluation. In particular it will be given a panoramic of which combination performs better in different circumstances and the reasons behind it. Concerning metrics, the results are reported as bar plots representing each experiment in the format “Model name - Experiment name”. Finally an analysis of the statistical significance of these results is presented.

5.1 Traditional metrics

5.1.1 BLEU

We bring to attention the following aspects of the performances on the BLEU metric:

- The application of LORA finetuning is beneficial for all models. The main reason behind this is the specificity of the task, which is difficult to handle from the base model as it requires extensive knowledge of the cooking domain as well as learning how to handle structured outputs.
- Some combination of custom losses always outperforms the baseline established by the Cross entropy loss. In particular we can notice that CE combined with either Topological Loss or Dice Loss outperforms pure CE across all models, while the application of the Focal loss is not beneficial. The increased performance might suggest that pure CE is not sufficient to learn appropriately the usage of less common yet critical tokens, such as ingredients and quantities.
- Topological loss performs better than Dice loss across all models. This result, if remains consistent across all metrics, might suggest that recipe generation does not only entails learning how to use less common tokens, but also the topological structure that they compose in a recipe.

- All models have a performance below 0.5, which could be explained by the missing of ordering rules in the generation of the ingredients and attributes in the recipe. Since the BLEU metric heavily depends on n-grams, the miss of ordering rules (for example, alphabetical) might lead to penalties in such scores. The results for the BLEU metric are presented in Table 5.1 .

Experiment Name	BLEU
Qwen3-4B-Topological	0.303985
Qwen3-4B-DiceLoss	0.283497
Qwen3-4B-FocalIngredients	0.275094
SmolLM3-3B-TopologicalLoss	0.270666
Qwen3-4B-CrossEntropyLoss	0.260293
SmolLM3-3B-DiceLoss	0.250838
SmolLM3-3B-CrossEntropyLoss	0.243894
SmolLM3-3B-FocalLoss	0.224938
Qwen2-1.5B-TopologicalLoss	0.217590
Qwen2-1.5B-DiceLoss	0.189898
Qwen2-1.5B-CrossEntropyLoss	0.170838
Qwen2-1.5B-FocalLoss	0.160374
Qwen3-4B-Baseline	0.054075
SmolLM3-3B-Baseline	0.049744
Qwen2-1.5B-Baseline	0.020875

Table 5.1: Table of results for BLEU score

5.1.2 ROUGE1

Similar to the evaluation of BLEU metric, the evaluation of ROUGE1 depicts the better performance of the combination CE+ Topological over the other experiments across all models. In this case, we can also observe that the base models do not accumulate anymore at the right tail but rather they get close to the cluster of experiments concerning their finetuned counterparts. These results are shown in Table 5.2.

5.1.3 BERTSCORE

The analysis of BERTScore has highlighted different aspects from what we have seen until now. As depicted in the figure, we can notice that the finetuning is not notably increasing the performances of the tested models. These results imply that base models are already capable of handling in a general way the creation of recipes and that finetuning should not aim to improve the semantic capabilities of the model but rather focus on the imbalance of tokens distributions in the responses and their relative contribution to the correctness of the answer. While we can still notice better performances when using the Topological Loss when associated to Cross

experiment$_{n}ame$	ROUGE1
Qwen3-4B-Topological	0.309586
Qwen3-4B-DiceLoss	0.298744
Qwen3-4B-CrossEntropyLoss	0.273048
Qwen3-4B-FocalIngredients	0.260949
SmolLM3-3B-TopologicalLoss	0.259079
SmolLM3-3B-DiceLoss	0.230849
Qwen3-4B-Baseline	0.224946
SmolLM3-3B-CrossEntropyLoss	0.218935
SmolLM3-3B-FocalLoss	0.203938
Qwen2-1.5B-TopologicalLoss	0.168577
SmolLM3-3B-Baseline	0.160909
Qwen2-1.5B-DiceLoss	0.143806
Qwen2-1.5B-CrossEntropyLoss	0.140978
Qwen2-1.5B-FocalLoss	0.139860
Qwen2-1.5B-Baseline	0.094880

Table 5.2: Table of results for ROUGE1 score

Entropy, we cannot anymore guarantee the statistical significance of these results. The BERTScore results are shown in Table 5.3.

Experiment Name	BERTSCORE F1
Qwen3-4B-Topological	0.909755
Qwen3-4B-DiceLoss	0.904908
Qwen3-4B-CrossEntropyLoss	0.902099
Qwen3-4B-FocalIngredients	0.899484
SmolLM3-3B-TopologicalLoss	0.897789
SmolLM3-3B-DiceLoss	0.895904
SmolLM3-3B-FocalLoss	0.890018
SmolLM3-3B-CrossEntropyLoss	0.887898
Qwen3-4B-Baseline	0.879339
Qwen2-1.5B-TopologicalLoss	0.877860
SmolLM3-3B-Baseline	0.864398
Qwen2-1.5B-DiceLoss	0.854987
Qwen2-1.5B-Baseline	0.854249
Qwen2-1.5B-FocalLoss	0.852535
Qwen2-1.5B-CrossEntropyLoss	0.850986

Table 5.3: Table of results for BERTScore F1 score

5.2 Ad-Hoc metrics

Action precision

In the context of the evaluation of the action precision, we can notice that the usage of Topological Loss has higher performance then baseline established by the usage of pure CE or combined with other loss functions. This is demonstrated in the plot in Table 5.4.

Experiment Name	Action Precision
Qwen3-4B-Topological	0.596869
Qwen3-4B-DiceLoss	0.505969
Qwen3-4B-CrossEntropyLoss	0.450969
Qwen3-4B-FocalIngredients	0.410958
SmolLM3-3B-TopologicalLoss	0.375597
SmolLM3-3B-DiceLoss	0.350967
SmolLM3-3B-CrossEntropyLoss	0.348390
SmolLM3-3B-FocalLoss	0.344904
Qwen3-4B-Baseline	0.324096
SmolLM3-3B-Baseline	0.310831
Qwen2-1.5B-TopologicalLoss	0.298187
Qwen2-1.5B-DiceLoss	0.286090
Qwen2-1.5B-CrossEntropyLoss	0.278700
Qwen2-1.5B-Baseline	0.273960
Qwen2-1.5B-FocalLoss	0.267894

Table 5.4: Table of results for action precision

5.2.1 Quantity precision

The figure confirms similar trends observed in other metrics also for what concerns the evaluation of quantity precision in recipe generation. We can observe in particular the major performances observed by using Topological and Dice Loss combined with Cross-Entropy. The full comparison is shown in Table 5.5.

5.2.2 Ingredient recall

The analysis of the recall of ingredients confirms better performance when using Topological loss. It is worth noting that the performance of the best models does not pass the 0.5 threshold. We believe this happens as the task consists of generating recipes based entirely on the title of the recipe. Since training and test dataset contain similar but yet different recipe, the model cannot always infer the entire list of ingredients simply based on a title of the recipe. Table 5.6 depicts the ingredient recall scores for all experiments.

5.2.3 Action and step edit distance

Action and Step edit distance are metrics similar to each other, therefore they share similar patterns and conclusions. We can indeed notice how the usage of Topological loss leads to a minimization of the differences between the target recipe and the

Experiment Name	Quantity Precision
Qwen3-4B-Topological	0.639350
Qwen3-4B-DiceLoss	0.574495
Qwen3-4B-FocalIngredients	0.549483
SmolLM3-3B-TopologicalLoss	0.518961
Qwen3-4B-CrossEntropyLoss	0.509490
SmolLM3-3B-DiceLoss	0.495758
SmolLM3-3B-CrossEntropyLoss	0.477397
SmolLM3-3B-FocalLoss	0.473939
Qwen2-1.5B-TopologicalLoss	0.434973
Qwen2-1.5B-DiceLoss	0.426840
Qwen2-1.5B-CrossEntropyLoss	0.417600
Qwen2-1.5B-FocalLoss	0.414877
Qwen3-4B-Baseline	0.250948
SmolLM3-3B-Baseline	0.222941
Qwen2-1.5B-Baseline	0.188897

Table 5.5: Table of results for quantity precision

Experiment Name	Ingredient Recall
Qwen3-4B-Topological	0.485976
Qwen3-4B-DiceLoss	0.449086
Qwen3-4B-FocalIngredients	0.430958
SmolLM3-3B-TopologicalLoss	0.369899
Qwen3-4B-CrossEntropyLoss	0.359849
SmolLM3-3B-DiceLoss	0.340976
SmolLM3-3B-CrossEntropyLoss	0.320394
SmolLM3-3B-FocalLoss	0.310928
Qwen2-1.5B-TopologicalLoss	0.292870
Qwen2-1.5B-DiceLoss	0.274907
Qwen2-1.5B-CrossEntropyLoss	0.264791
Qwen3-4B-Baseline	0.260958
Qwen2-1.5B-FocalLoss	0.253222
SmolLM3-3B-Baseline	0.220935
Qwen2-1.5B-Baseline	0.148790

Table 5.6: Table of results for ingredient recall

predicted, followed by the application of Dice Loss. The results for action edit distance are shown in Table 5.7. We can notice an higher error in step distance as it implies the observations of more attributes in order to consider two steps as identical, which results more often in dissimilar steps. The step edit distance results are shown in Table 5.8.

Experiment Name	Action Edit Distance
Qwen3-4B-Topological	0.304959
Qwen3-4B-DiceLoss	0.310949
Qwen3-4B-FocalIngredients	0.374093
Qwen3-4B-CrossEntropyLoss	0.378348
SmolLM3-3B-CrossEntropyLoss	0.410928
SmolLM3-3B-TopologicalLoss	0.415958
SmolLM3-3B-FocalLoss	0.429830
SmolLM3-3B-DiceLoss	0.435958
Qwen2-1.5B-TopologicalLoss	0.453903
Qwen2-1.5B-DiceLoss	0.464897
Qwen2-1.5B-CrossEntropyLoss	0.479685
SmolLM3-3B-Baseline	0.484039
Qwen3-4B-Baseline	0.485094
Qwen2-1.5B-FocalLoss	0.508560
Qwen2-1.5B-Baseline	0.579897

Table 5.7: Table of results for action edit distance (the smaller, the better)

Experiment Name	Step Edit Distance
Qwen3-4B-Topological	0.340959
Qwen3-4B-DiceLoss	0.350868
Qwen3-4B-FocalIngredients	0.380597
Qwen3-4B-CrossEntropyLoss	0.394859
SmolLM3-3B-CrossEntropyLoss	0.420928
SmolLM3-3B-TopologicalLoss	0.420973
SmolLM3-3B-FocalLoss	0.438929
SmolLM3-3B-DiceLoss	0.445958
Qwen2-1.5B-TopologicalLoss	0.519821
Qwen3-4B-Baseline	0.520928
Qwen2-1.5B-DiceLoss	0.526694
Qwen2-1.5B-CrossEntropyLoss	0.555576
SmolLM3-3B-Baseline	0.560939
Qwen2-1.5B-FocalLoss	0.589650
Qwen2-1.5B-Baseline	0.634479

Table 5.8: Table of results for step edit distance (the smaller, the better)

5.2.4 Temperature precision

The analysis of precision in regards to the temperature at which each step should be performed yields different results from the other steps. Table 5.9 depicts the dice loss experiment to outperform the topological loss across all models. The reason behind this might lie in the implementation of the topological loss, which focuses exclusively on the list of ingredients, while the dice loss is extended to the entire recipe. Therefore, topological loss does not outperform dice as temperature is not an attribute related to the ingredient list.

Experiment Name	Temperature Precision
Qwen3-4B-DiceLoss	0.745886
Qwen3-4B-Topological	0.655969
Qwen3-4B-CrossEntropyLoss	0.619384
Qwen3-4B-FocalIngredients	0.596080
SmolLM3-3B-DiceLoss	0.589575
SmolLM3-3B-TopologicalLoss	0.579797
SmolLM3-3B-CrossEntropyLoss	0.566698
SmolLM3-3B-FocalLoss	0.563903
Qwen2-1.5B-DiceLoss	0.486987
Qwen2-1.5B-CrossEntropyLoss	0.477866
Qwen2-1.5B-TopologicalLoss	0.470790
Qwen2-1.5B-FocalLoss	0.470004
Qwen3-4B-Baseline	0.398451
SmolLM3-3B-Baseline	0.375040
Qwen2-1.5B-Baseline	0.310980

Table 5.9: Table of results for temperature precision

5.2.5 Time precision

Similarly to the analysis of temperature precision, Table **??** shows that Dice loss outperforms topological loss and the other experiments across all models. We believe similar conclusions to the one obtained in the previous analysis can be drawn to motivate these results.

Experiment Name	Time Precision
Qwen3-4B-DiceLoss	0.596868
Qwen3-4B-Topological	0.555598
Qwen3-4B-CrossEntropyLoss	0.520938
Qwen3-4B-FocalIngredients	0.485298
SmolLM3-3B-DiceLoss	0.464484
SmolLM3-3B-TopologicalLoss	0.458889
SmolLM3-3B-CrossEntropyLoss	0.444984
SmolLM3-3B-FocalLoss	0.443098
Qwen2-1.5B-DiceLoss	0.429074
Qwen3-4B-Baseline	0.414509
Qwen2-1.5B-TopologicalLoss	0.409097
Qwen2-1.5B-CrossEntropyLoss	0.398967
Qwen2-1.5B-FocalLoss	0.393346
SmolLM3-3B-Baseline	0.373020
Qwen2-1.5B-Baseline	0.330888

Table 5.10: Table of results for time precision

5.2.6 Combining Topological and Dice Losses

Having noted the differing and also complementary qualities that have been seen in previous sections—the Topological Loss performs well regarding ingredient-level metrics, as well as action-related metrics, while the Dice Loss performs well in numerical qualities such as time and temperature—a final experiment was carried out in the hopes that a combination of the two aforementioned custom loss functions would yield a model that performs in a well-rounded fashion.

In order to verify this, a new composite loss was introduced that included both Dice Loss and Topological Loss, also alongside Cross Entropy. This "Mixed Model" setup was used to finetune the Qwen3-4B model, and its performance was compared to those models that were trained only on Dice and Topological loss. The findings from this are highlighted in Figure 5.1, while numerical values can be observed in 5.11

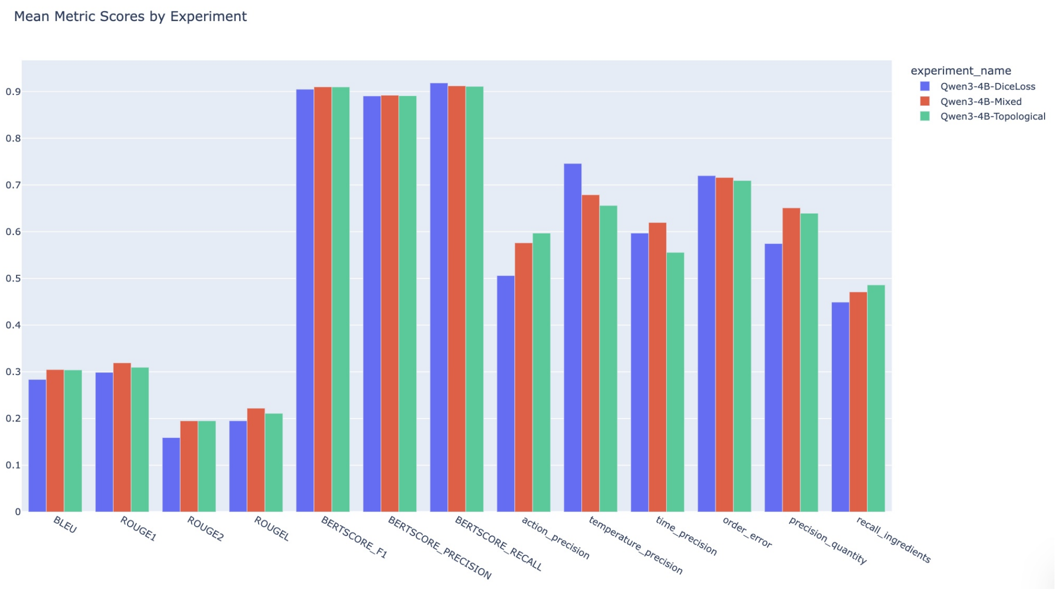


Figure 5.1: Performance comparison of the Mixed Model (Topological + Dice) against the individual Topological Loss and Dice Loss models across all evaluation metrics.

This analysis of results highlights a complex landscape of trade-offs:

- On the level of linguistic properties (BLEU, ROUGE metrics), it is possible to observe that Mixed Model performs at least as well as the pure Topological model and, in several instances, it is actually slightly better. In a similar fashion to previous experiments, no particular points of interest have emerged in terms of differences at BERTScore metrics.
- Some of the key metrics in an ad-hoc setting have a Mixed Model as a compromise. In terms of Action Precision, Temperature Precision, Ingredient Recall, and order/edit distance metrics, a score of the Mixed Model lies in between that of a Topological model and a Dice model.

Metric	Qwen3 4B Dice	Qwen3 4B Mixed	Qwen3 4B Topological
BERTSCORE F1	0.904908	**0.909940**	0.909755
BLEU	0.283497	**0.304599**	0.303985
ROUGE1	0.298744	0.319045	0.309586
Action Edit Distance	**0.310949**	0.304959	0.304959
Action Precision	0.505969	0.575984	**0.596869**
Quantity Precision	0.574495	**0.650957**	0.639350
Ingredients Recall	0.449086	0.470949	**0.485976**
Step Edit Distance	**0.350868**	0.340959	0.340959
Temperature Precision	**0.745886**	0.678958	0.655969
Time Precision	0.596868	**0.619585**	0.555598

Table 5.11: Comparison of the results of Mixed loss compared to Dice and Topological losses

- Nevertheless, in two critical areas, Quantity Precision and Time Precision, a synergistic effect is seen in the Mixed Model. In such scenarios, a joint loss function enabled a better score than those of either a Topological or a Dice model when trained individually.

Based upon this experiment, we have been able to draw one clear conclusion, which is that using a combination of custom loss functions will not make it possible for a model to do well at all aspects at the same time. This combination might work better than its "parents" in a few conditions, but in most conditions, it enables a balance between them.

5.2.7 Statistical Significance of Results

In order to be sure that performance differences evident through these finetuned models are not simply a result of random chance, we performed a rigorous statistical test. This is in consideration for guaranteeing that these gains evident through our custom loss functions, particularly our Topological Loss, are significant rather than simple luck. We compared the output from the top-scoring model, Qwen3-4B, under different loss settings used for this test.

5.2.7.1 Methodology: Welch's t-test

In order to compare two independent sample groups' means (e.g., 1000 recipes created with Topological Loss with 1000 recipes created with Cross-Entropy), we used Welch's t-test. It is a variant of the standard Student's t-test that is particularly developed for cases when both groups under comparison may possess different variances. Since different loss functions can modify consistency as well as distribution of a model's output, we cannot presume that the standard deviation of scores for each metric would be identical for different experimental setups. The Welch's t-test does not demand that assumption, along with being more reliable and robust for our analysis purposes.

For each comparison, we tested the null hypothesis that the mean scores of two groups are equal. We set our significance level to $\alpha = 0.05$. If a resulting p-value falls below this critical region, then we can reject the null hypothesis and state that we see a statistically significant difference in what was observed.

The following tables present the statistical comparisons between the Topological Loss and the two other key configurations: the baseline Cross-Entropy loss and the next-best custom loss, Dice Loss.

5.2.7.1.1 Topological Loss vs. Cross-Entropy Baseline Firstly, we compare our best-performing method with fine-tuning baselines of standard methods. The "Cross-Entropy" is compared with "Topological Loss" in Table 5.12.

Table 5.12: Welch's t-test results comparing Qwen3-4B finetuned with Cross-Entropy vs. Topological Loss .

Metric	Loss Function	Mean	Std. Dev.	t-statistic	p-value	Significant?
BLEU	Cross-Entropy	0.2339	0.0681	15.9620	< 0.0001	Yes
	Topological	0.3040	0.1209			
ROUGE-1	Cross-Entropy	0.2406	0.0799	15.9548	< 0.0001	Yes
	Topological	0.3096	0.1109			
BERTScore F1	Cross-Entropy	0.9028	0.0171	4.6318	< 0.0001	Yes
	Topological	0.9098	0.0440			
Action Precision	Cross-Entropy	0.3340	0.0760	65.9818	< 0.0001	Yes
	Topological	0.5969	0.1005			
Quantity Precision	Cross-Entropy	0.4820	0.0689	52.5476	< 0.0001	Yes
	Topological	0.6393	0.0649			
Ingredient Recall	Cross-Entropy	0.3109	0.1119	39.3956	< 0.0001	Yes
	Topological	0.4860	0.0849			
Temperature Precision	Cross-Entropy	0.5640	0.1111	19.3751	< 0.0001	Yes
	Topological	0.6560	0.1008			
Time Precision	Cross-Entropy	0.4505	0.1209	21.5386	< 0.0001	Yes
	Topological	0.5556	0.0958			

The results are unmistakable. The Topological Loss model is considerably better than the Cross-Entropy baseline on every single one of these measures. The p-values are consistently well below 0.05, thereby substantiating that these improvements are not due to chance. The extremely large t-statistics for the ad-hoc measures, particularly for Action Precision (t=65.98), Quantity Precision (t=52.55), and Ingredient Recall (t=39.40), indicate a very large and in practice significant performance gap. This is compelling evidence that the application of a custom loss was successful in directing the model's learning to these critical, domain-specific features.

5.2.7.1.2 Topological Loss vs. Dice Loss Then we compare the top-two best-performing custom losses to see whether the dominance of the Topological Loss is also statistically significant. In Table 5.13 we can observe the outcome of the comparison of the "Dice Loss" with the "Topological Loss".

Table 5.13: Welch's t-test results comparing Qwen3-4B finetuned with Dice Loss vs. Topological Loss .

Metric	Loss Function	Mean	Std. Dev.	t-statistic	p-value	Significant?
BLEU	Dice Loss	0.2835	0.1039	4.0638	< 0.0001	Yes
	Topological	0.3040	0.1209			
ROUGE-1	Dice Loss	0.2987	0.0929	2.3695	0.0179	Yes
	Topological	0.3096	0.1109			
BERTScore F1	Dice Loss	0.9049	0.1198	1.2011	0.2299	No
	Topological	0.9098	0.0440			
Action Precision	Dice Loss	0.5060	0.1041	19.8691	< 0.0001	Yes
	Topological	0.5969	0.1005			
Quantity Precision	Dice Loss	0.5745	0.0829	19.4766	< 0.0001	Yes
	Topological	0.6393	0.0649			
Ingredient Recall	Dice Loss	0.4491	0.0958	9.1093	< 0.0001	Yes
	Topological	0.4860	0.0849			
Temperature Precision	Dice Loss	0.7459	0.1050	-19.5323	< 0.0001	Yes
	Topological	0.6560	0.1008			
Time Precision	Dice Loss	0.5969	0.1028	-9.2831	< 0.0001	Yes
	Topological	0.5556	0.0958			

This analysis confirms the nuanced trade-offs observed in the main results. For the metrics where Topological Loss was superior—BLEU, ROUGE-1, Action Precision, Quantity Precision, and Ingredient Recall—the improvements are statistically significant. Conversely, for Temperature and Time Precision, the stronger performance of the Dice Loss is also highly significant. Finally, the test validates that the small difference in BERTScore is not statistically significant (p=0.2299), reinforcing the conclusion that semantic fluency is not the primary differentiator between these two advanced loss functions.

5.2.7.1.3 Mixed Loss vs. Dice Loss vs. Topological Loss In examining these results, we observe an interesting phenomenon. From Table 5.14 , our hypothesis that the mixed loss is not a universal solution that outperform in all the metrics is verified. On the contrary, it appears more as a compromise between those two pure losses. For instance, if we take a look at Àction Precision', it was actually better off with pure topological loss, whereas in T̀emperature Precision', pure Dice loss was a winner. In such scenarios, our mixed loss was often in the middle of everything. This is proof that it is handling it well. However, in a few other instances, it actually performed better than both. This is because in R̀OUGE-1', T̀ime Precision', and Q̀uantity Precision', we observe that it led to a definitive and significant improvement over our previous performance. This indicates that it is a good strategy to incorporate a mix of a loss in order to obtain a balance that might result in a s̀weet spot'.

Table 5.14: Welch's t-test results comparing Mixed Loss (Dice + Topological) vs. pure Topological Loss and pure Dice Loss.

Metric	Comparison	Mean	Std. Dev.	t-statistic	p-value	Significant?
BLEU	Mixed Topological	0.3046 0.3040	0.1109 0.1209	0.1183	0.9059	No
	Mixed Dice	0.3046 0.2835	0.1109 0.1039	4.3910	< 0.0001	Yes
ROUGE-1	Mixed Topological	0.3190 0.3096	0.1039 0.1109	1.9676	0.0492	Yes
	Mixed Dice	0.3190 0.2987	0.1039 0.0929	4.6047	< 0.0001	Yes
BERTScore F1	Mixed Topological	0.9099 0.9098	0.0491 0.0440	0.0888	0.9293	No
	Mixed Dice	0.9099 0.9049	0.0491 0.1198	1.2292	0.2192	No
Action Precision	Mixed Topological	0.5760 0.5969	0.0984 0.1005	-4.6961	< 0.0001	Yes
	Mixed Dice	0.5760 0.5060	0.0984 0.1041	15.4592	< 0.0001	Yes
Temperature Precision	Mixed Topological	0.6790 0.6560	0.1119 0.1008	4.8251	< 0.0001	Yes
	Mixed Dice	0.6790 0.7459	0.1119 0.1050	-13.7908	< 0.0001	Yes
Time Precision	Mixed Topological	0.6196 0.5556	0.0975 0.0958	14.7958	< 0.0001	Yes
	Mixed Dice	0.6196 0.5969	0.0975 0.1028	5.0677	< 0.0001	Yes
Quantity Precision	Mixed Topological	0.6510 0.6393	0.0551 0.0649	4.3101	< 0.0001	Yes
	Mixed Dice	0.6510 0.5745	0.0551 0.0829	24.2931	< 0.0001	Yes
Ingredient Recall	Mixed Topological	0.4709 0.4860	0.0948 0.0849	-3.7322	0.0002	Yes
	Mixed Dice	0.4709 0.4491	0.0948 0.0958	5.1275	< 0.0001	Yes

Chapter 6

Conclusions

In this thesis, we have explored the finetuning of Small Language Models on the niche topic of recipe generation and proposed a novel framework for achieving maximal factual and procedural correctness. The primary goal of this work was to move beyond the limitations of standard finetuning methodologies that rely upon a Cross-Entropy loss poorly suited for tasks where numerical correctness and recall of entities are paramount. Framing the exploration on a highly structured, fact-rich domain, this work provided a controlled environment for exploring a concrete alternative to popular training procedured and revealed the tremendous future potential of customized loss functions for constructing generative models that are more accurate and useful.

The major restriction of typical finetuning by the "equal importance" requirement of Cross-Entropy is why models inevitably fail to learn the important but statistically rare tokens that define the correctness of a recipe. This token skewness and the lack of numeric reasoning in the training objective mandate a differently constructed solution so that true domain expertise may be fostered. Through the combination of a multi-task dataset and a composite loss setup, the new model has been able to generate models that have a better understanding of the recipe space. The incorporation of Focal, Dice, and a new Topological loss that are all meant to compensate for a particular failing of the baseline has resulted in statistical improvements. Most prominently, the Topological loss, acting over the geometric relationship between ingredients in the embedding space, showed the highest performance by far across the vast majority of our bespoke evaluation measures, from ingredient recall to all the way through procedural accuracy. Further, the models have been able to achieve this higher performance while having been trained under the very efficient regime of the LoRA methodology, and so show that significant improvements are possible without the unrealistic cost of full parameter finetuning.

Furthermore, a final experiment combining the Dice and Topological losses revealed a nuanced landscape of trade-offs: while the mixed loss often acts as a compromise between the two, it can also create a synergistic effect, outperforming both parent losses in specific metrics like Quantity and Time Precision.

As language models become more deeply integrated into practical, real-world applications, the accuracy of facts, reliability, and structural correctness of the output of these models will become crucial. This work demonstrates that by carefully crafting the loss function to accommodate a specific domain's unique needs, we are able to significantly improve a model's quality along these crucial axes. The loss functions carefully constructed for the unique needs of this work, and the Topological loss in particular, have the potential to be transferred to a great number of domains where entities' relationships and factuality are relevant. Future work may entail the exploration of other topological and computer vision losses, as well as the application of custom losses on other procedural problems such as prescription of medicines.

Bibliography

[1] Sotiris Kotsiantis. “Supervised Machine Learning: A Review of Classification Techniques.” In: *Informatica (Slovenia)* 31 (Jan. 2007), pp. 249–268 (cit. on p. 6).

[2] L. G. Valiant. “A theory of the learnable”. In: *Commun. ACM* 27.11 (Nov. 1984), pp. 1134–1142. ISSN: 0001-0782. DOI: 10.1145/1968.1972. URL: https://doi.org/10.1145/1968.1972 (cit. on p. 6).

[3] V. N. Vapnik. “An overview of statistical learning theory”. In: *Trans. Neur. Netw.* 10.5 (Sept. 1999), pp. 988–999. ISSN: 1045-9227. DOI: 10.1109/72.788640. URL: https://doi.org/10.1109/72.788640 (cit. on p. 6).

[4] V. Vapnik. “Principles of Risk Minimization for Learning Theory”. In: *Advances in Neural Information Processing Systems.* Ed. by J. Moody, S. Hanson, and R.P. Lippmann. Vol. 4. Morgan-Kaufmann, 1991. URL: https://proceedings.neurips.cc/paper_files/paper/1991/file/ff4d5fbbafdf976cfdc032e3bde78de5-Paper.pdf (cit. on p. 6).

[5] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. *Sequence to Sequence Learning with Neural Networks.* 2014. arXiv: 1409.3215 [cs.CL]. URL: https://arxiv.org/abs/1409.3215 (cit. on p. 8).

[6] In Jae Myung. “Tutorial on maximum likelihood estimation”. In: *J. Math. Psychol.* 47.1 (Feb. 2003), pp. 90–100. ISSN: 0022-2496. DOI: 10.1016/S0022-2496(02)00028-7. URL: https://doi.org/10.1016/S0022-2496(02)00028-7 (cit. on p. 8).

[7] Sean Welleck, Ilia Kulikov, Stephen Roller, Emily Dinan, Kyunghyun Cho, and Jason Weston. *Neural Text Generation with Unlikelihood Training.* 2019. arXiv: 1908.04319 [cs.LG]. URL: https://arxiv.org/abs/1908.04319 (cit. on p. 8).

[8] Long Ouyang et al. *Training language models to follow instructions with human feedback.* 2022. arXiv: 2203.02155 [cs.CL]. URL: https://arxiv.org/abs/2203.02155 (cit. on p. 8).

[9] Tze Lai. “Stochastic approximation”. In: *The Annals of Statistics* 31 (Apr. 2003). DOI: 10.1214/aos/1051027873 (cit. on p. 9).

[10] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* 2017. arXiv: `1412.6980 [cs.LG]`. URL: `https://arxiv.org/abs/1412.6980` (cit. on p. 9).

[11] Stuart Geman, Elie Bienenstock, and René Doursat. "Neural Networks and the Bias/Variance Dilemma". In: *Neural Computation* 4 (Jan. 1992), pp. 1–58. DOI: `10.1162/neco.1992.4.1.1` (cit. on p. 10).

[12] Lutz Prechelt. "Early Stopping - But When?" In: *Lecture Notes in Computer Science* (Mar. 2000). DOI: `10.1007/3-540-49430-8_3` (cit. on p. 10).

[13] N. Chomsky. "Three models for the description of language". In: *IRE Transactions on Information Theory* 2.3 (1956), pp. 113–124. DOI: `10.1109/TIT.1956.1056813` (cit. on p. 11).

[14] Peter F. Brown, John Cocke, Stephen A. Della Pietra, Vincent J. Della Pietra, Fredrick Jelinek, John D. Lafferty, Robert L. Mercer, and Paul S. Roossin. "A statistical approach to machine translation". In: *Comput. Linguist.* 16.2 (June 1990), pp. 79–85. ISSN: 0891-2017 (cit. on p. 11).

[15] Rico Sennrich, Barry Haddow, and Alexandra Birch. *Neural Machine Translation of Rare Words with Subword Units.* 2016. arXiv: `1508.07909 [cs.CL]`. URL: `https://arxiv.org/abs/1508.07909` (cit. on p. 11).

[16] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. *Efficient Estimation of Word Representations in Vector Space.* 2013. arXiv: `1301.3781 [cs.CL]`. URL: `https://arxiv.org/abs/1301.3781` (cit. on p. 12).

[17] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. "A neural probabilistic language model". In: *J. Mach. Learn. Res.* 3.null (Mar. 2003), pp. 1137–1155. ISSN: 1532-4435 (cit. on p. 12).

[18] George Kingsley Zipf. *The Psycho-Biology of Language: An Introduction to Dynamic Philology.* First edition published 1935 by Houghton Mifflin. Cambridge, MA: The M.I.T. Press, 1965 (cit. on p. 12).

[19] 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 of the Association for Computational Linguistics.* Ed. by Pierre Isabelle, Eugene Charniak, and Dekang Lin. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, July 2002, pp. 311–318. DOI: `10.3115/1073083.1073135`. URL: `https://aclanthology.org/P02-1040/` (cit. on p. 13).

[20] Chin-Yew Lin. "ROUGE: A Package for Automatic Evaluation of Summaries". In: *Text Summarization Branches Out.* Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81. URL: `https://aclanthology.org/W04-1013/` (cit. on p. 13).

[21] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. *BERTScore: Evaluating Text Generation with BERT*. 2020. arXiv: `1904.09675 [cs.CL]`. URL: `https://arxiv.org/abs/1904.09675` (cit. on p. 13).

[22] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2022. arXiv: `1312.6114 [stat.ML]`. URL: `https://arxiv.org/abs/1312.6114` (cit. on p. 14).

[23] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial nets". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'14. Montreal, Canada: MIT Press, 2014, pp. 2672–2680 (cit. on p. 14).

[24] Robin M. Schmidt. *Recurrent Neural Networks (RNNs): A gentle Introduction and Overview*. 2019. arXiv: `1912.05911 [cs.LG]`. URL: `https://arxiv.org/abs/1912.05911` (cit. on p. 19).

[25] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: `10.1162/neco.1997.9.8.1735`. URL: `https://doi.org/10.1162/neco.1997.9.8.1735` (cit. on p. 19).

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2023. arXiv: `1706.03762 [cs.CL]`. URL: `https://arxiv.org/abs/1706.03762` (cit. on p. 19).

[27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: `1810.04805 [cs.CL]`. URL: `https://arxiv.org/abs/1810.04805` (cit. on p. 22).

[28] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. "Improving language understanding by generative pre-training". In: (2018). URL: `https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf` (cit. on p. 22).

[29] Gunnar Carlsson. "Topology and Data". In: *Bulletin of The American Mathematical Society - BULL AMER MATH SOC* 46 (Apr. 2009), pp. 255–308. DOI: `10.1090/S0273-0979-09-01249-X` (cit. on p. 31).

[30] Afra Zomorodian and Gunnar Carlsson. "Computing Persistent Homology". In: *Discrete and Computational Geometry* 33 (Feb. 2005), pp. 249–274. DOI: `10.1007/s00454-004-1146-y` (cit. on p. 31).

[31] Morris Hirsch. "Computational Homology". In: *Bulletin of the American Mathematical Society* Volume 43 (Feb. 2006), 255=258 (cit. on p. 31).

[32] Yossi Rubner, Carlo Tomasi, and Leonidas Guibas. "The Earth Mover's Distance as a Metric for Image Retrieval". In: *International Journal of Computer Vision* 40 (Nov. 2000), pp. 99–121. DOI: `10.1023/A:1026543900054` (cit. on p. 33).

[33] Martin Arjovsky, Soumith Chintala, and Léon Bottou. *Wasserstein GAN*. 2017. arXiv: `1701.07875 [stat.ML]`. URL: `https://arxiv.org/abs/1701.07875` (cit. on p. 33).

[34] Marco Cuturi. "Sinkhorn distances: lightspeed computation of optimal transport". In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'13. Lake Tahoe, Nevada: Curran Associates Inc., 2013, pp. 2292–2300 (cit. on p. 33).

[35] Jeremy Howard and Sebastian Ruder. *Universal Language Model Fine-tuning for Text Classification*. 2018. arXiv: `1801.06146 [cs.CL]`. URL: `https://arxiv.org/abs/1801.06146` (cit. on p. 35).

[36] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. *Finetuned Language Models Are Zero-Shot Learners*. 2022. arXiv: `2109.01652 [cs.CL]`. URL: `https://arxiv.org/abs/2109.01652` (cit. on p. 35).

[37] Lingling Xu, Haoran Xie, Si-Zhao Joe Qin, Xiaohui Tao, and Fu Lee Wang. *Parameter-Efficient Fine-Tuning Methods for Pretrained Language Models: A Critical Review and Assessment*. 2023. arXiv: `2312.12148 [cs.CL]`. URL: `https://arxiv.org/abs/2312.12148` (cit. on p. 37).

[38] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. *Parameter-Efficient Transfer Learning for NLP*. 2019. arXiv: `1902.00751 [cs.LG]`. URL: `https://arxiv.org/abs/1902.00751` (cit. on p. 37).

[39] Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. *Towards a Unified View of Parameter-Efficient Transfer Learning*. 2022. arXiv: `2110.04366 [cs.CL]`. URL: `https://arxiv.org/abs/2110.04366` (cit. on p. 37).

[40] Brian Lester, Rami Al-Rfou, and Noah Constant. *The Power of Scale for Parameter-Efficient Prompt Tuning*. 2021. arXiv: `2104.08691 [cs.CL]`. URL: `https://arxiv.org/abs/2104.08691` (cit. on p. 38).

[41] Elad Ben Zaken, Shauli Ravfogel, and Yoav Goldberg. *BitFit: Simple Parameter-efficient Fine-tuning for Transformer-based Masked Language-models*. 2022. arXiv: `2106.10199 [cs.LG]`. URL: `https://arxiv.org/abs/2106.10199` (cit. on p. 38).

[42] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: `2106.09685 [cs.CL]`. URL: `https://arxiv.org/abs/2106.09685` (cit. on pp. 39, 41).

[43] Dan Biderman et al. *LoRA Learns Less and Forgets Less*. 2024. arXiv: `2405.09673 [cs.LG]`. URL: `https://arxiv.org/abs/2405.09673` (cit. on p. 40).

[44] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. 2017. arXiv: `1712.05877 [cs.LG]`. URL: `https://arxiv.org/abs/1712.05877` (cit. on p. 46).

[45] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. *QLoRA: Efficient Finetuning of Quantized LLMs*. 2023. arXiv: `2305.14314 [cs.LG]`. URL: `https://arxiv.org/abs/2305.14314` (cit. on p. 48).

[46] Shubham Pundhir and Ganesh Bagler. *The Digital Sous Chef – A Comparative Study on Fine-Tuning Language Models for Recipe Generation*. 2025. arXiv: `2508.14718 [cs.CL]`. URL: `https://arxiv.org/abs/2508.14718` (cit. on pp. 50, 51, 60).

[47] Michał Bień, Michał Gilski, Martyna Maciejewska, Wojciech Taisner, Dawid Wisniewski, and Agnieszka Lawrynowicz. "RecipeNLG: A Cooking Recipes Dataset for Semi-Structured Text Generation". In: *Proceedings of the 13th International Conference on Natural Language Generation*. Ed. by Brian Davis, Yvette Graham, John Kelleher, and Yaji Sripada. Dublin, Ireland: Association for Computational Linguistics, Dec. 2020, pp. 22–28. DOI: `10.18653/v1/2020.inlg-1.4`. URL: `https://aclanthology.org/2020.inlg-1.4/` (cit. on pp. 50, 52, 53, 58, 60, 63).

[48] Shreyas Chaudhari, Pranjal Aggarwal, Vishvak Murahari, Tanmay Rajpurohit, Ashwin Kalyan, Karthik Narasimhan, Ameet Deshpande, and Bruno Castro da Silva. *RLHF Deciphered: A Critical Analysis of Reinforcement Learning from Human Feedback for LLMs*. 2024. arXiv: `2404.08555 [cs.LG]`. URL: `https://arxiv.org/abs/2404.08555` (cit. on pp. 50, 58).

[49] Anneketh Vij, Changhao Liu, Rahul Anil Nair, Theodore Eugene Ho, Edward Shi, and Ayan Bhowmick. *Fine-tuning Language Models for Recipe Generation: A Comparative Analysis and Benchmark Study*. 2025. arXiv: `2502.02028 [cs.CL]`. URL: `https://arxiv.org/abs/2502.02028` (cit. on pp. 51, 53–55, 61).

[50] Kanghee Park, Timothy Zhou, and Loris D'Antoni. *Flexible and Efficient Grammar-Constrained Decoding*. 2025. arXiv: `2502.05111 [cs.CL]`. URL: `https://arxiv.org/abs/2502.05111` (cit. on pp. 51, 59).

[51] David Noever and Samantha Elizabeth Miller Noever. *The Multimodal And Modular Ai Chef: Complex Recipe Generation From Imagery*. 2023. arXiv: `2304.02016 [cs.CL]`. URL: `https://arxiv.org/abs/2304.02016` (cit. on p. 52).

[52] Ruoxuan Zhang, Hongxia Xie, Yi Yao, Jian-Yu Jiang-Lin, Bin Wen, Ling Lo, Hong-Han Shuai, Yung-Hui Li, and Wen-Huang Cheng. *RecipeGen: A Benchmark for Real-World Recipe Image Generation.* 2025. arXiv: 2503.05228 [cs.CV]. URL: https://arxiv.org/abs/2503.05228 (cit. on p. 53).

[53] Ruoxuan Zhang, Jidong Gao, Bin Wen, Hongxia Xie, Chenming Zhang, Hong-Han Shuai, and Wen-Huang Cheng. *RecipeGen: A Step-Aligned Multimodal Benchmark for Real-World Recipe Generation.* 2025. arXiv: 2506.06733 [cs.CV]. URL: https://arxiv.org/abs/2506.06733 (cit. on pp. 53, 55, 58, 60).

[54] Ehud Reiter. "A Structured Review of the Validity of BLEU". In: *Computational Linguistics* 44.3 (Sept. 2018), pp. 393–401. DOI: 10.1162/coli_a_00322. URL: https://aclanthology.org/J18-3002/ (cit. on p. 54).

[55] Emerson G. Escolar, Yuta Shimada, and Masahiro Yuasa. *A topological analysis of the space of recipes.* 2024. arXiv: 2406.09445 [math.AT]. URL: https://arxiv.org/abs/2406.09445 (cit. on p. 55).

[56] Qingquan Zhang, Qiqi Duan, Bo Yuan, Yuhui Shi, and Jialin Liu. *Exploring Accuracy-Fairness Trade-off in Large Language Models.* 2024. arXiv: 2411.14500 [cs.CL]. URL: https://arxiv.org/abs/2411.14500 (cit. on pp. 58, 59).

[57] Qiyao Ma, Yunsheng Shi, Hongtao Tian, Chao Wang, Weiming Chang, and Ting Yao. *From Faithfulness to Correctness: Generative Reward Models that Think Critically.* 2025. arXiv: 2509.25409 [cs.CL]. URL: https://arxiv.org/abs/2509.25409 (cit. on pp. 58, 60).

[58] Mohaddeseh Bastan, Mihai Surdeanu, and Niranjan Balasubramanian. "NEUROSTRUCTURAL DECODING: Neural Text Generation with Structural Constraints". In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* Ed. by Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki. Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 9496–9510. DOI: 10.18653/v1/2023.acl-long.528. URL: https://aclanthology.org/2023.acl-long.528/ (cit. on p. 59).

[59] Xiang Chen and Xiaojun Wan. *Evaluating, Understanding, and Improving Constrained Text Generation for Large Language Models.* 2024. arXiv: 2310.16343 [cs.CL]. URL: https://arxiv.org/abs/2310.16343 (cit. on p. 59).

[60] James Y. Huang, Sailik Sengupta, Daniele Bonadiman, Yi-an Lai, Arshit Gupta, Nikolaos Pappas, Saab Mansour, Katrin Kirchhoff, and Dan Roth. *DeAL: Decoding-time Alignment for Large Language Models.* 2024. arXiv: 2402.06147 [cs.AI]. URL: https://arxiv.org/abs/2402.06147 (cit. on p. 59).

[61] Darren Yow-Bang Wang, Zhengyuan Shen, Soumya Smruti Mishra, Zhichao Xu, Yifei Teng, and Haibo Ding. *SLOT: Structuring the Output of Large Language Models.* 2025. arXiv: 2505.04016 [cs.CL]. URL: https://arxiv.org/abs/2505.04016 (cit. on p. 59).

[62] Jun Ma. *Segmentation Loss Odyssey.* 2020. arXiv: `2005.13449 [eess.IV]`. URL: `https://arxiv.org/abs/2005.13449` (cit. on p. 59).

[63] Daniele Rege Cambrin, Giuseppe Gallipoli, Irene Benedetto, Luca Cagliero, and Paolo Garza. "Beyond Accuracy Optimization: Computer Vision Losses for Large Language Model Fine-Tuning". In: *Findings of the Association for Computational Linguistics: EMNLP 2024.* Association for Computational Linguistics, 2024, pp. 12060–12079. DOI: `10.18653/v1/2024.findings-emnlp.704`. URL: `http://dx.doi.org/10.18653/v1/2024.findings-emnlp.704` (cit. on p. 59).

[64] Demis Hassabis and Koray Kavukcuoglu. *Introducing Gemini 2.0: our new AI model for the agentic era.* Accessed: 2025-10-31. Dec. 2024. URL: `https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/` (cit. on p. 64).