Re:Fine

A Eklavya 2025 Project

Anushree Bobade
Mehul Totala

Acknowledgements

We would like to express our heartfelt gratitude to our mentors, Atharva Kshirsagar and Prithvi Tambewagh, for their guidance, encouragement, and unwavering support throughout the journey of creating Re:fine. Your insights, patience, and expertise have been invaluable in shaping this project and helping us refine our ideas into reality.

We also extend our sincere thanks to SRA VJTI for organizing Eklavya 2025 and providing us with the platform to showcase our work. Your efforts in fostering innovation, learning, and collaboration have made this experience truly inspiring and enriching.

Finally, we would like to acknowledge the wider community, peers, and resources that have contributed—directly or indirectly—to the success of this project. Your support has been instrumental in helping Re:fine grow from an idea into a tool that empowers creators, thinkers, and doers.

Thank you.

Phase I

Chapter 1 — Aim and Abstract

Productivity, in software development, is about reducing the time and effort spent on understanding code so developers can focus on creating. Code is not just instructions for machines—it's a shared language between humans. If models can effectively summarize and explain code, we boost both speed and clarity.

Our project, **Re:Fine**, set out to achieve this by fine-tuning **CodeT5**. We began with supervised fine-tuning, training the model on code–summary pairs to establish a solid baseline. From there, we expanded into reinforcement learning, moving beyond imitation to actively rewarding outputs that were more accurate, readable, and useful.

To further strengthen the model, we experimented with **synthetic data generation** to supplement scarce labeled datasets. This allowed for greater diversity and better generalization.

Insights from our first project guided the design of our second iteration, where we introduced **Generalized Reinforcement Policy Optimization (GRPO)**. GRPO helped stabilize reinforcement learning and made the fine-tuning process more efficient.

In short, **Re:Fine** is not just about teaching LLMs to understand code, but about **refining them to make humans more productive with it**.

Chapter 2 — Machine Learning vs Deep Learning

Machine Learning

Machine Learning (ML) is a branch of artificial intelligence focused on creating systems that learn patterns from data and improve performance over time without being explicitly programmed. Traditional ML models include algorithms such as Linear Regression, Decision Trees, Random Forests, and Support Vector Machines (SVMs). These models often rely heavily on feature engineering — where humans design and select the right features from raw data before training. ML excels when data is structured and problem sizes are moderate.

Deep Learning

Deep Learning (DL) is a specialized subset of ML that uses artificial neural networks with multiple hidden layers to automatically learn features from raw data. Unlike classical ML, DL requires minimal feature engineering — the network itself extracts hierarchical features (edges → shapes → objects in images, or tokens → syntax → semantics in text). Deep learning has driven breakthroughs in computer vision, speech recognition, natural language processing, and generative AI.

DL models (e.g., CNNs, RNNs, Transformers) require large datasets and substantial computational resources, but they often outperform traditional ML in tasks with complex, high-dimensional data.

Key Differences between ML and DL

Aspect	Machine Learning	Deep Learning
Data Dependence	Performs well on smaller, structured datasets	Requires large datasets for good performance
Feature Engineering	Heavily dependent on human designed features	Learns features automatically from raw data
Computation	Less computationally intensive	Requires high computational power (GPUs/TPUs)
Interpretability	More interpretable, easier to debug	Often a “black box,” harder to interpret

Chapter 3 — Neural Networks

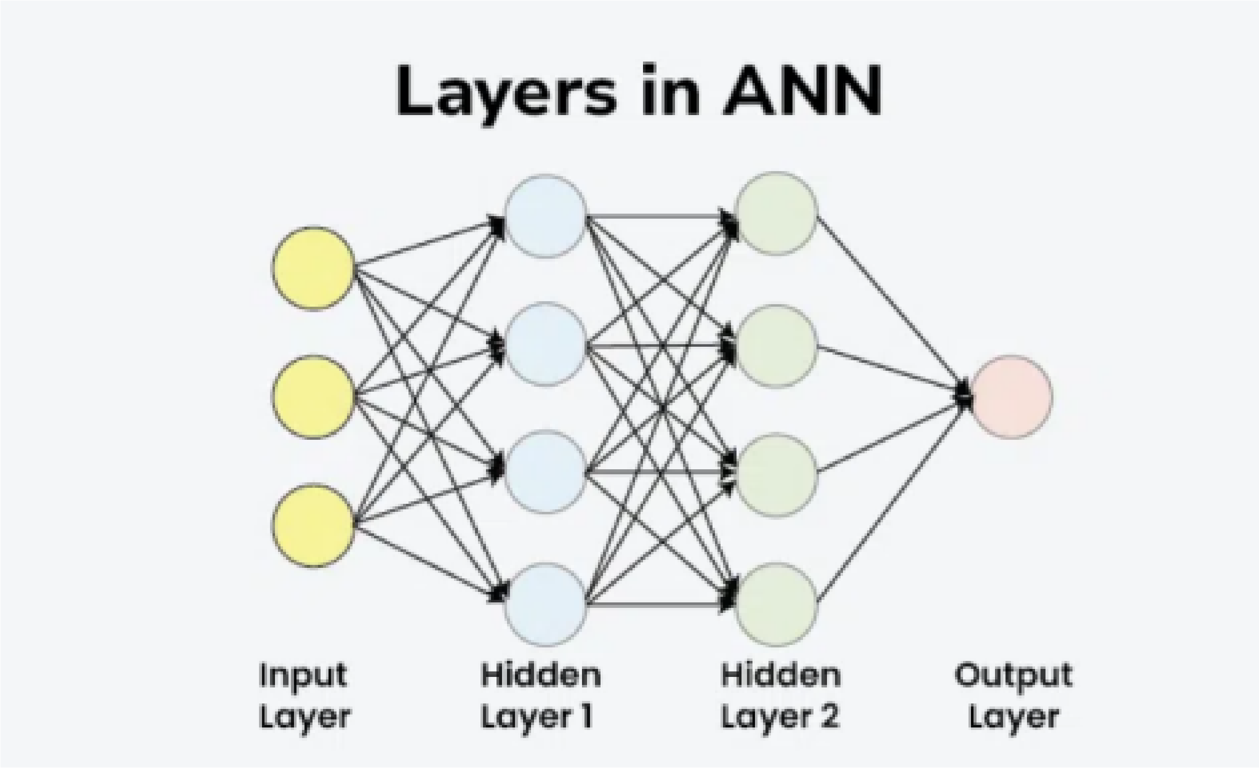
A neural network is a computational model inspired by the human brain's structure. Instead of neurons and synapses, it consists of mathematical units (nodes) and weighted connections. These networks form the backbone of deep learning because of their ability to learn complex, non-linear relationships directly from data.

Structure of a Neural Network

Input Layer – The entry point where raw data is fed in. For instance, pixel values in an image, or token embeddings in text.

Hidden Layers – One or more layers where the "real magic" happens. Each neuron in a hidden layer computes a weighted sum of inputs and applies an activation function. These layers extract patterns and transform raw data into higher-level representations.

Output Layer – Produces the final prediction. For classification, it could output probabilities across categories; for regression, a continuous value; for generation, a sequence of tokens.



Forward Propagation

Forward propagation is the process where data flows from input → hidden layers → output. At each stage, the network transforms the input into something more abstract. For example:

Early text layers detect word-level patterns.

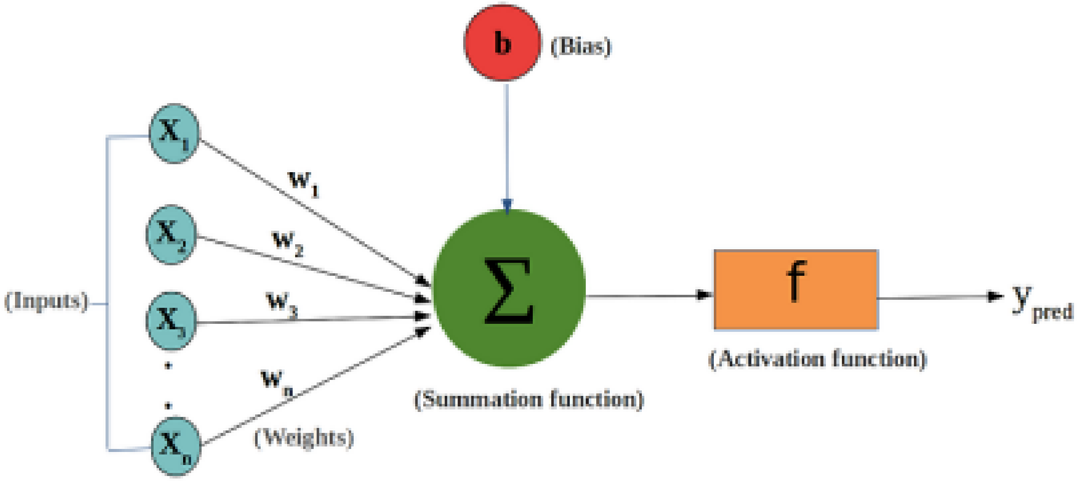
Deeper layers capture syntax and semantics.

Weights and Biases

Weights: Numbers that control the strength of connections between neurons.

Biases: Adjustable values that shift the activation threshold, allowing flexibility.

Together, these are the parameters that the network learns during training.



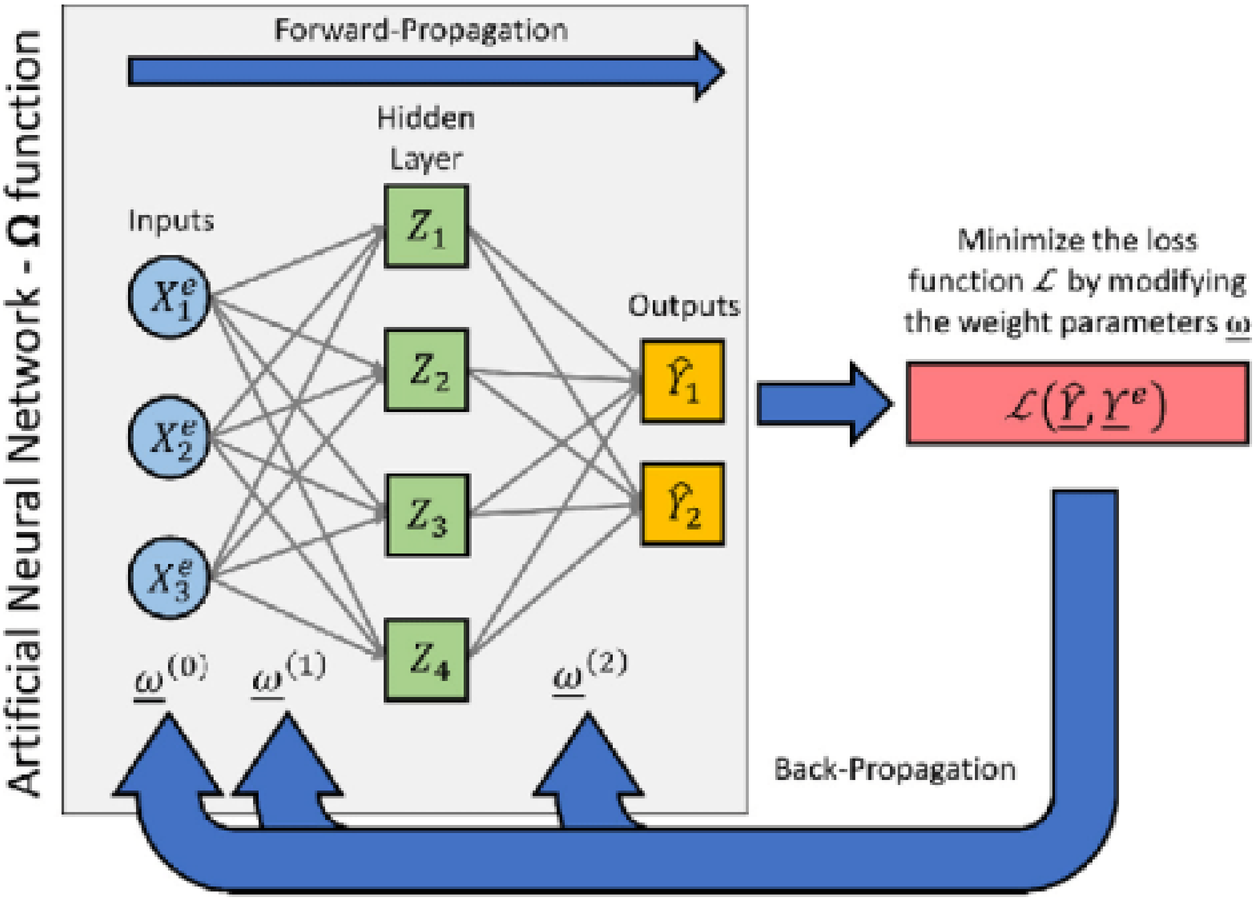
Backpropagation

Neural networks learn via backpropagation. After producing an output, the network compares it to the true label using a loss function. The error is then propagated backwards:

Gradients (slopes) are calculated for each weight.

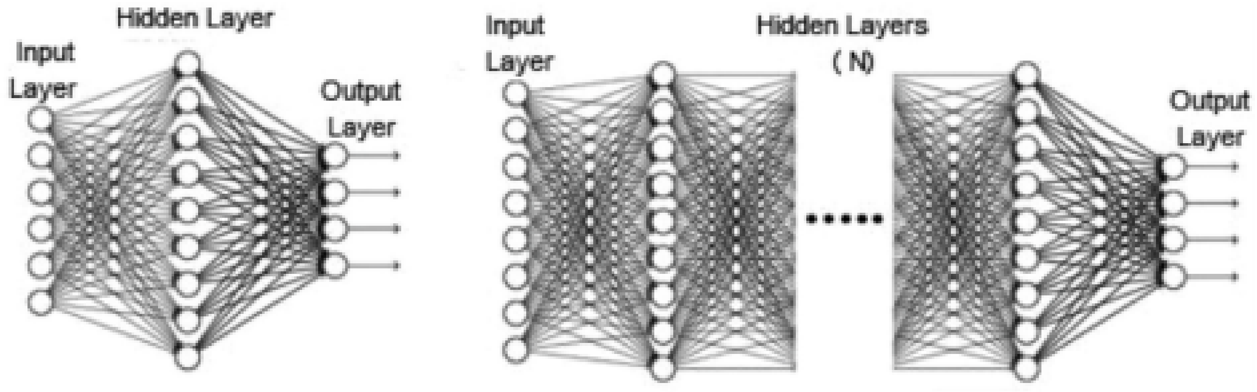
Parameters are updated using an optimization algorithm like gradient descent.

This process continues over many iterations until the network minimizes its errors.


Artificial Neural Network - Ω function
Forward-Propagation
Hidden Layer
Inputs
X_1^e
X_2^e
X_3^e
Z_1
Z_2
Z_3
Z_4
Outputs
$\hat{Y}_1$
$\hat{Y}_2$
Minimize the loss function $\mathcal{L}$ by modifying the weight parameters $\underline{\omega}$
$\mathcal{L}(\underline{\hat{Y}}, \underline{Y}^e)$
$\underline{\omega}^{(0)}$
$\underline{\omega}^{(1)}$
$\underline{\omega}^{(2)}$
Back-Propagation

Shallow vs Deep Networks

Shallow networks: Have one hidden layer; useful for simpler problems. **Deep networks**: Contain many hidden layers; capable of learning highly complex functions, powering today's breakthroughs in AI.



(a) A Shallow Network

(b) A Deep Neural Network

Chapter 4 — Key Activation Functions

Activation functions are a critical component of neural networks. Their primary role is to introduce non-linearity into the model. Without an activation function, a neural network, no matter how many layers it has, would behave just like a single-layer linear regression model. By adding non-linearity, activation functions allow the network to learn complex patterns and relationships in the data, making it possible to solve sophisticated problems like image recognition and natural language processing. They essentially decide whether a neuron should be "activated" or not, based on the weighted sum of its inputs.

4.1 The Sigmoid Function

The sigmoid function is often used in the output layer of a binary classification model because it maps any real-valued number into a value between 0 and 1, which can be interpreted as a probability.

Formula:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Derivative:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

Key Characteristics:

Output Range: (0, 1)

Problem: Suffers from the "vanishing gradient" problem, where the gradient becomes extremely small for large positive or negative inputs, slowing down or halting the learning process.

4.2 Rectified Linear Unit (ReLU)

ReLU is the most widely used activation function in modern deep learning models, especially in hidden layers. It is computationally efficient and generally leads to faster training.

Formula:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

Derivative:

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

Key Characteristics:

Output Range: [0, ∞)

Advantage: Avoids the vanishing gradient problem for positive inputs and promotes sparse activations, which can be more efficient.

Problem: Can suffer from the "Dying ReLU" problem, where neurons become inactive and only output zero.

4.3 Softmax Function

The softmax function is exclusively used in the output layer of a multi-class classification model. It takes a vector of arbitrary real-valued scores (logits) and transforms them into a vector of values between 0 and 1 that sum to 1, representing a probability distribution across multiple classes.

Formula:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Key Characteristics:

Output Range: (0, 1) for each element.

Primary Use: Converts the final layer's outputs into a probability distribution, making it ideal for determining the most likely class in a classification task.

Chapter 5 — Linear Regression, Loss, and Cost Functions

Linear Regression

Linear Regression is a fundamental supervised learning algorithm used for predicting a continuous dependent variable (y) based on one or more independent variables (x).

The core idea is to find the best-fitting straight line, known as the regression line, that describes the relationship between the variables. This line is defined by its slope and intercept, which the algorithm learns from the training data.

The hypothesis function $h_\theta(x)$ represents the predicted value. For a simple case with one independent variable, the equation is:

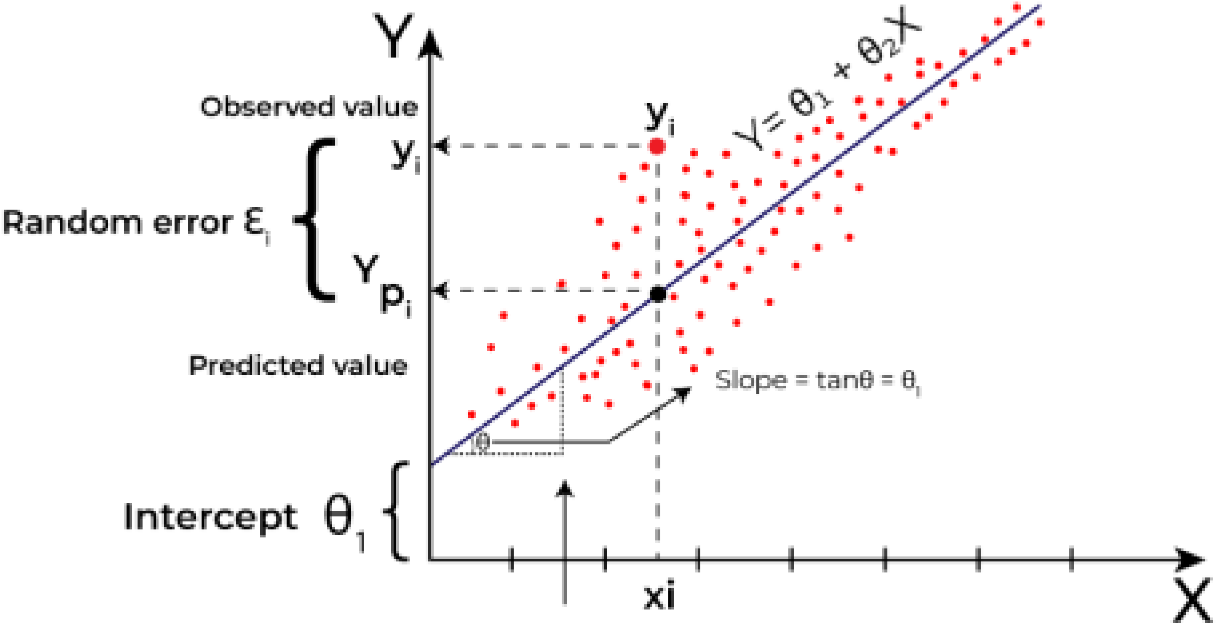
Formula:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

θ_0 is the y-intercept.

θ_1 is the slope of the line.

x is the input feature.


Y
Observed value
y_i
y_i
Random error ε_i
Yp_i
Predicted value
$Y = \theta_1 + \theta_2 X$
Slope = tanθ = θ_1
θ
Intercept θ_1
xi
X

Loss and Cost Functions

To find the "best-fitting" line, we need a way to measure how wrong our model's predictions are. This is where loss and cost functions come in.

Loss Function

A Loss Function calculates the error for a single training example. It measures the difference between the predicted value (y^) and the actual value (y). For linear regression, the most common loss function is the **Squared Error**.

Formula (Squared Error Loss):

$$L(\hat{y}, y) = (\hat{y} - y)^2$$

Squaring the error ensures the result is always positive.

Penalizes larger errors more significantly.

Cost Function

While the loss function works on a single example, the **Cost Function** measures the average error across the entire training dataset.

The goal of training is to find the parameters (θ_0, θ_1) that minimize this cost. The most common cost function for linear regression is the **Mean Squared Error (MSE)**.

Formula (Mean Squared Error):

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Where:

m is the total number of training examples.

Σ means summing the squared errors for every single example.

The $^{1}{}_{2m}$ term averages the error and simplifies gradient calculations.

Chapter 6 — Gradient Descent & Hyperparameters

Gradient Descent (GD)

Gradient Descent is an optimization algorithm used to minimize the loss function of a model by updating its parameters iteratively. Parameters are adjusted in the direction opposite to the gradient of the loss function, effectively taking "steps" downhill towards the lowest error.

Update formula:

$$\theta := \theta - \eta \nabla_{\theta} L(\theta)$$

Where:

θ → model parameters

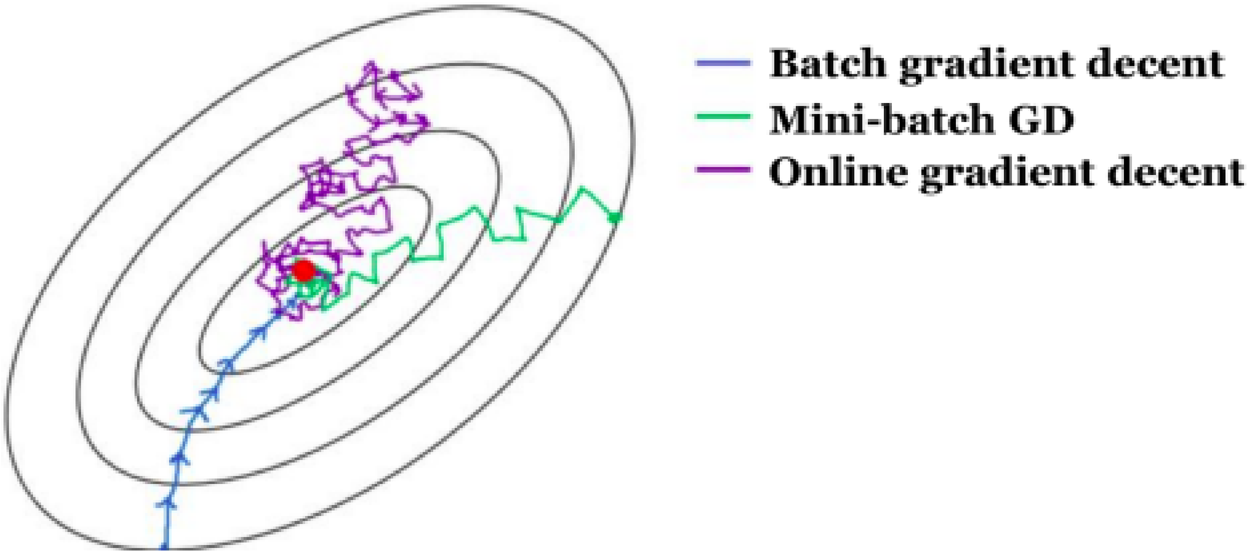
α → learning rate

∇J(θ) → gradient of the loss function w.r.t parameters

Types of Gradient Descent

Batch Gradient Descent: Uses the entire dataset. Stable but slow.

Stochastic Gradient Descent (SGD): Updates per example. Fast but noisy. **Mini-batch Gradient Descent**: Uses small subsets. Combines speed and stability.



Hyperparameters

Hyperparameters are configuration values set before training begins. They are not learned from the data but chosen to control how the model learns.

The choice of hyperparameters can significantly impact model performance.

Key Hyperparameters

Learning Rate (α): Determines step size for updates.

Too small: Learning is very slow, may get stuck.

Too large: Model may overshoot the minimum and fail to converge.

Batch Size: Number of samples per update.

Small: Updates noisy, helps escape local minima but unstable.

Large: Updates stable, but memory heavy and slower.

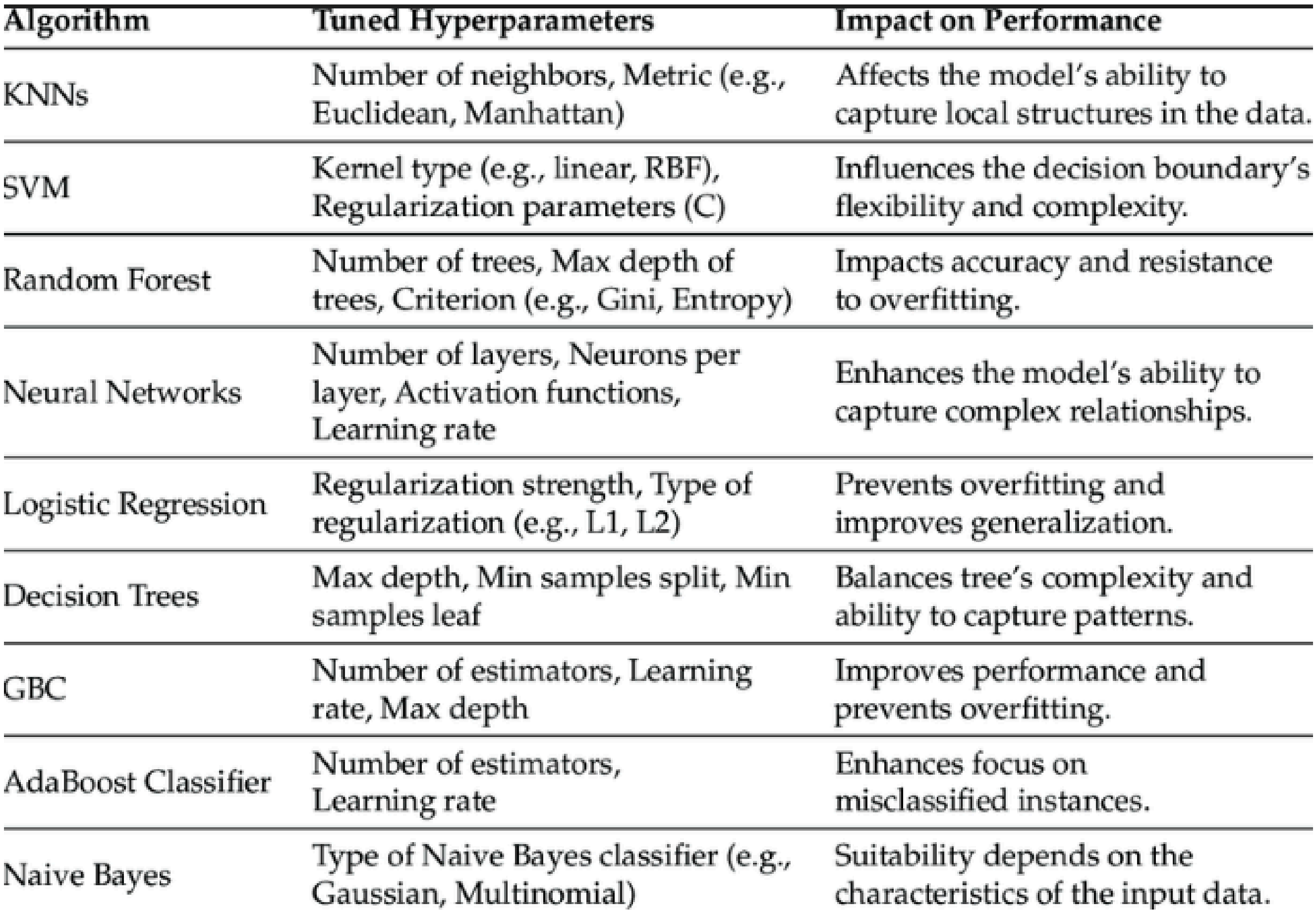
Epochs: Number of times the dataset is passed through.

Too few: Underfitting.

Too many: Overfitting.

Momentum (β): Accelerates descent and dampens oscillations.

Regularization (λ): Penalizes large weights to prevent overfitting.



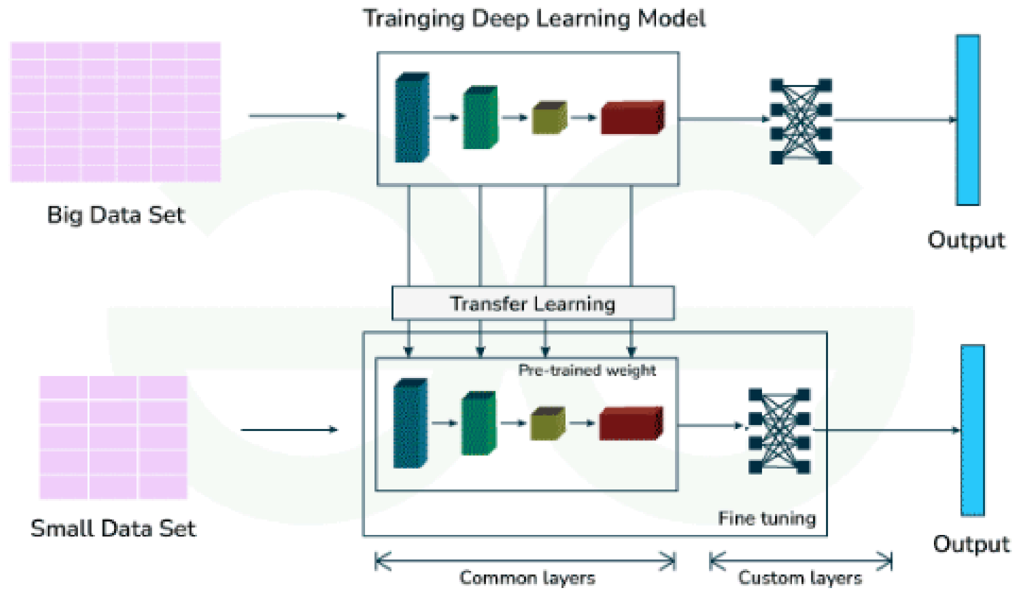
Algorithm	Tuned Hyperparameters	Impact on Performance
KNNs	Number of neighbors, Metric (e.g., Euclidean, Manhattan)	Affects the model's ability to capture local structures in the data.
SVM	Kernel type (e.g., linear, RBF), Regularization parameters (C)	Influences the decision boundary's flexibility and complexity.
Random Forest	Number of trees, Max depth of trees, Criterion (e.g., Gini, Entropy)	Impacts accuracy and resistance to overfitting.
Neural Networks	Number of layers, Neurons per layer, Activation functions, Learning rate	Enhances the model's ability to capture complex relationships.
Logistic Regression	Regularization strength, Type of regularization (e.g., L1, L2)	Prevents overfitting and improves generalization.
Decision Trees	Max depth, Min samples split, Min samples leaf	Balances tree's complexity and ability to capture patterns.
GBC	Number of estimators, Learning rate, Max depth	Improves performance and prevents overfitting.
AdaBoost Classifier	Number of estimators, Learning rate	Enhances focus on misclassified instances.
Naive Bayes	Type of Naive Bayes classifier (e.g., Gaussian, Multinomial)	Suitability depends on the characteristics of the input data.

Chapter 7 — Supervised Fine-Tuning (SFT)

Fine-Tuning

Fine-tuning is the process of adapting a pre-trained model to a specific task by continuing its training on a smaller, task-relevant dataset. Instead of training from scratch, the model leverages its pre-learned representations to achieve high performance efficiently.

Example:
A language model pre-trained on general text can be fine-tuned to classify legal documents into categories like *"Contract," "Litigation,"* or *"IP"* using labeled legal datasets.



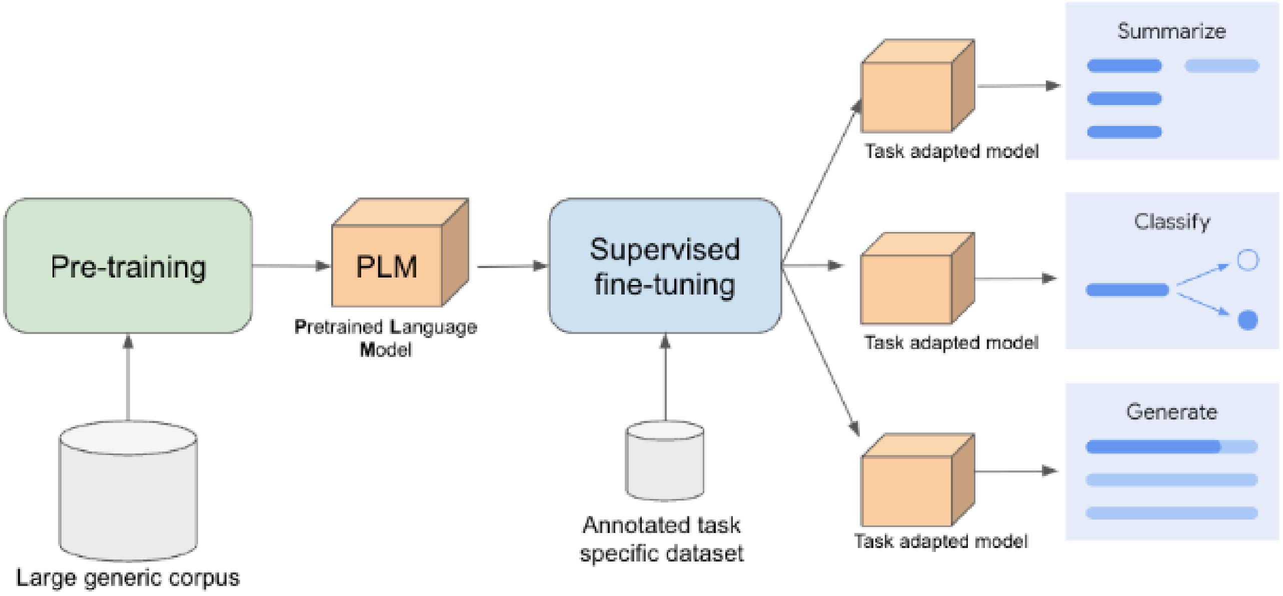
Fine tuning

Supervised Fine-Tuning (SFT)

Supervised Fine-Tuning uses labeled datasets where each input has a corresponding correct output. The model learns the mapping between inputs and outputs, allowing it to specialize in a particular task.

Process

1. **Pre-trained Model Selection**: Start with a model trained on a large, general dataset (e.g., T5, CodeT5).
2. **Dataset Preparation**: Curate high-quality labeled datasets relevant to the target task.
3. **Fine-Tuning**: Adjust model parameters to minimize the loss function. Optimization is typically done via gradient descent or adaptive methods.
4. **Evaluation**: Use task-specific metrics (accuracy, BLEU, ROUGE, etc.) to assess performance and iterate if needed.



Use Cases of SFT

Natural Language Processing: Sentiment analysis, NER, machine translation.

Code Intelligence: Code summarization, code completion, bug detection.

Computer Vision: Object detection, image classification.

Healthcare: Medical imaging classification, disease prediction.

Why SFT Matters

Specializes models for specific tasks.

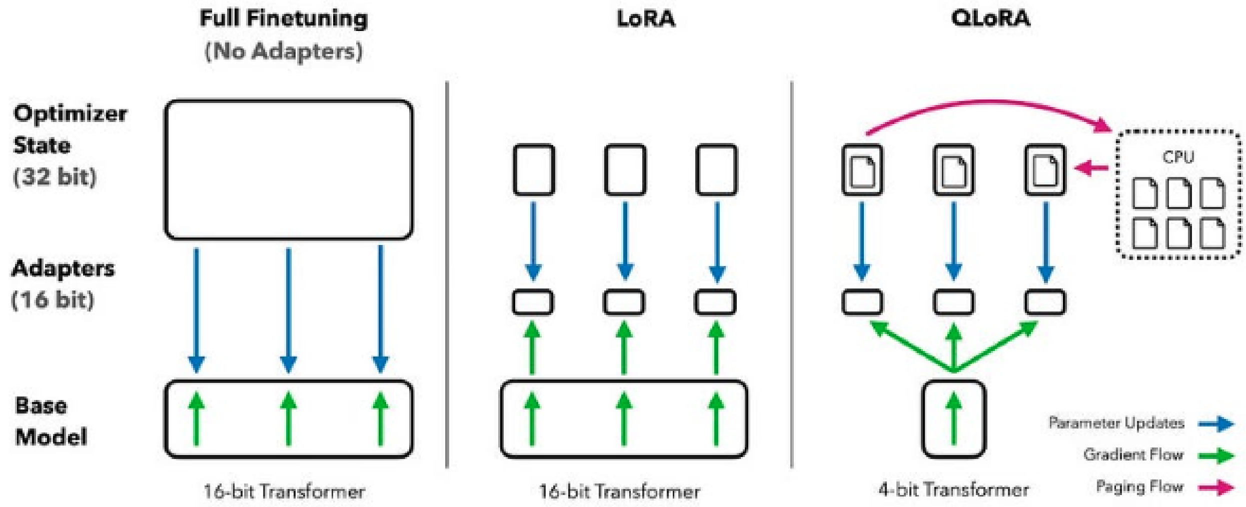
Requires less data and computation than training from scratch.

Flexible across domains and applications.

Chapter 8 — Low-Rank Adaptation (LoRA) & QLoRA

Introduction

Large language models (LLMs) are powerful but expensive to fine-tune. **LoRA (Low-Rank Adaptation)** and **QLoRA (Quantized LoRA)** reduce computational cost while preserving performance, enabling efficient adaptation of massive models on limited hardware.



LoRA (Low-Rank Adaptation)

LoRA introduces trainable low-rank matrices into transformer layers instead of updating all weights. This reduces trainable parameters and memory requirements.

Key Concepts

Pre-trained weights are frozen, only low-rank matrices are updated.

Captures task-specific information efficiently.

Applicable to NLP, code generation, summarization, etc.

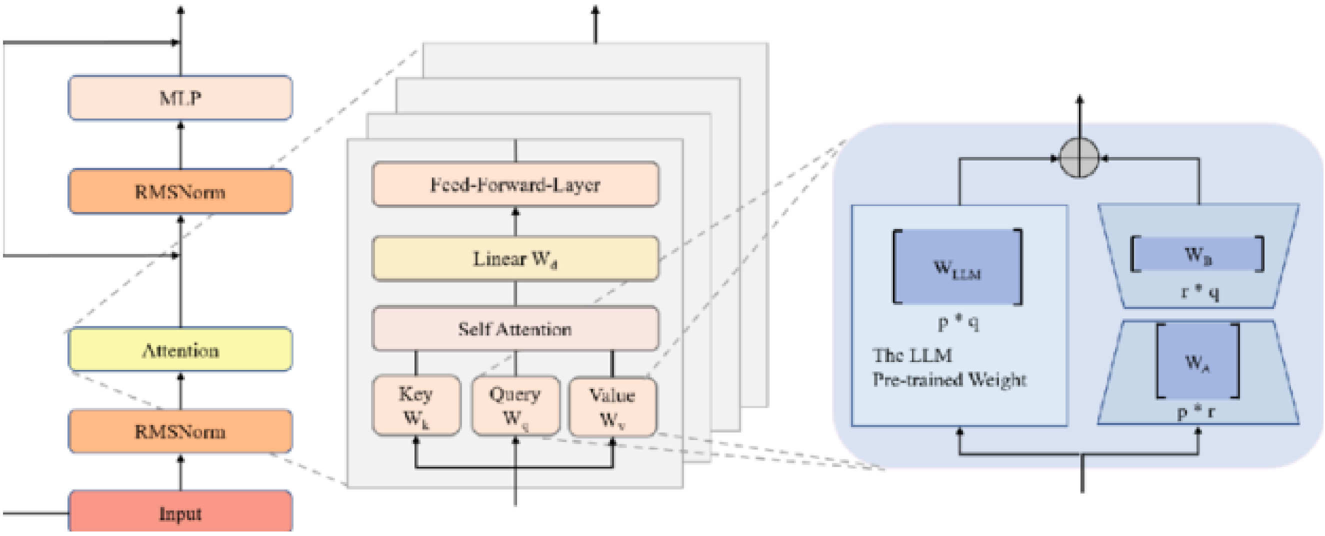
Mathematical Formulation:

$$W_0x + \Delta W x = W_0x + BAx$$

Where:

W_0 = pre-trained weight (frozen)

A, B = trainable low-rank matrices



QLoRA (Quantized LoRA)

QLoRA combines LoRA with weight quantization, representing weights in lower precision (e.g., 4-bit or 8-bit).

Benefits

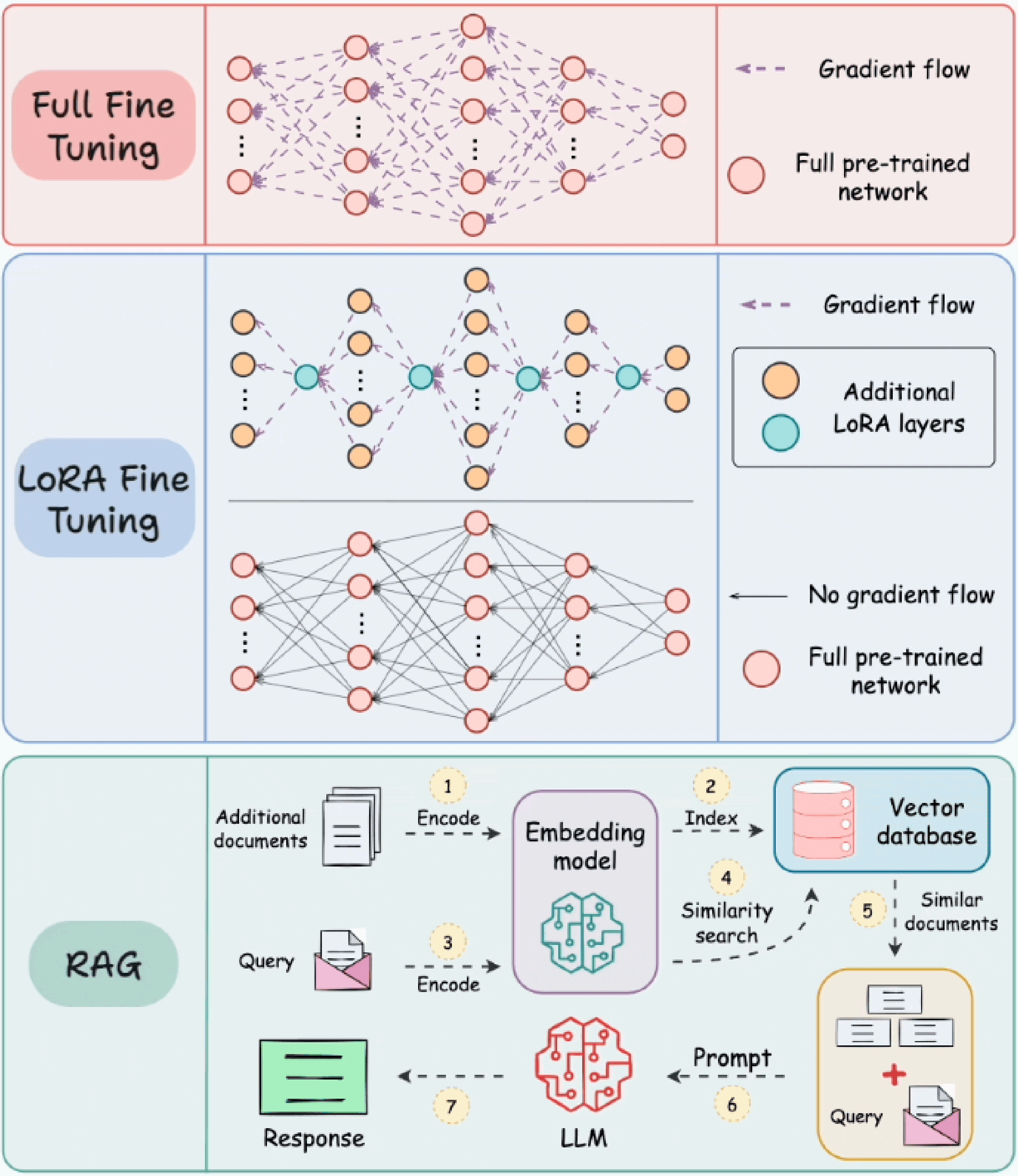
Reduces memory footprint.

Allows fine-tuning of very large models on smaller GPUs.

Maintains performance despite quantization.

Full Fine Tuning, LoRA and RAG

blog.DailyDoseofDS.com



Chapter 9 — Our Project:

Codeuctivity Introduction

Codeuctivity is a project aimed at improving developer productivity by enabling automatic code summarization. By converting functions into natural-language explanations, the goal is to help developers quickly comprehend large codebases, ease onboarding, and improve documentation workflows.

Model Selection: CodeT5

We selected **CodeT5-base**, a transformer-based encoder–decoder model pre-trained specifically for code understanding and generation tasks (*Wang et al., EMNLP 2021*).

Key Advantages

Pre-trained on diverse programming languages.

Robust understanding of syntax and semantics.

Compatible with **LoRA** adapters, enabling efficient fine-tuning.

Reference: CodeT5 paper

Dataset: CodeXGLUE (Code-to-Text)

We used the HuggingFace dataset `code_x_glue_ct_code_to_text`, focusing on the **Python subset** for function-to-docstring summarization.

Dataset Characteristics

Attribute	**Details**
Programming Language	Python
Training Samples	~251,000
Validation Samples	~13,000
Test Samples	~15,000
Task	Code summarization (function → docstring)

Chapter 10 — Fine-Tuning CodeT5 Using LoRA LoRA Configuration

To make fine-tuning efficient, we applied **LoRA adapters** with the following setup:

Rank (r): 16

Scaling factor (α): 32

Dropout: 0.1

Trainable parameters: ~0.8% of total model parameters

This significantly reduced training cost and memory usage compared to full fine-tuning.

Training Setup

Base Model: `Salesforce/codet5-base`

Batch Size: 4

Learning Rate: 5e-4

Epochs: 1 (pilot run)

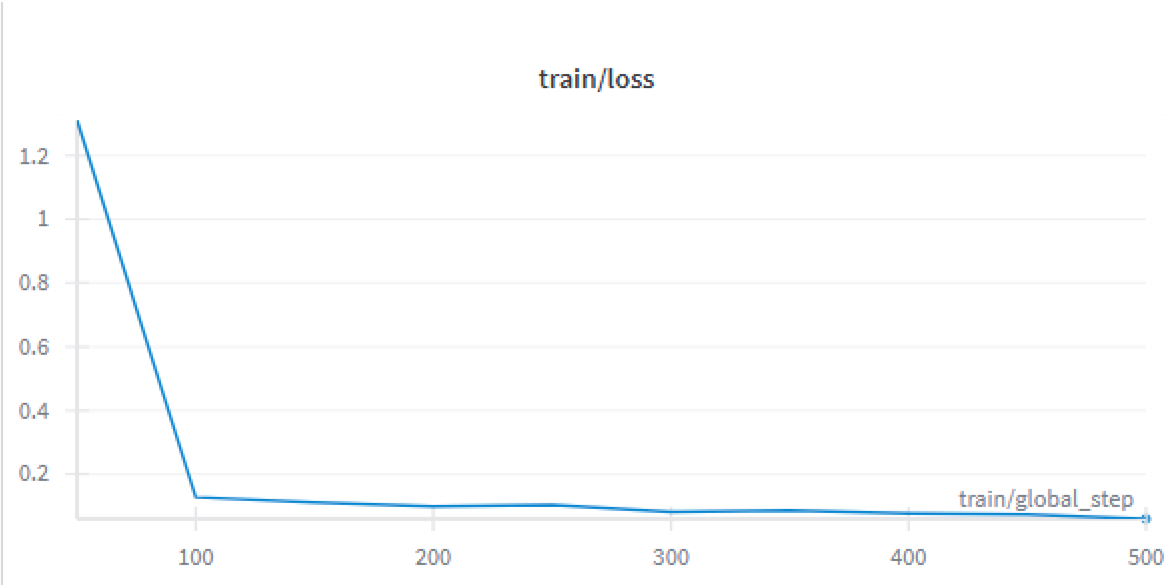
Logging & Evaluation: HuggingFace `Trainer` API + Weights & Biases (W&B) tracking

Training Runs

1. **Trial Run (500 steps)**

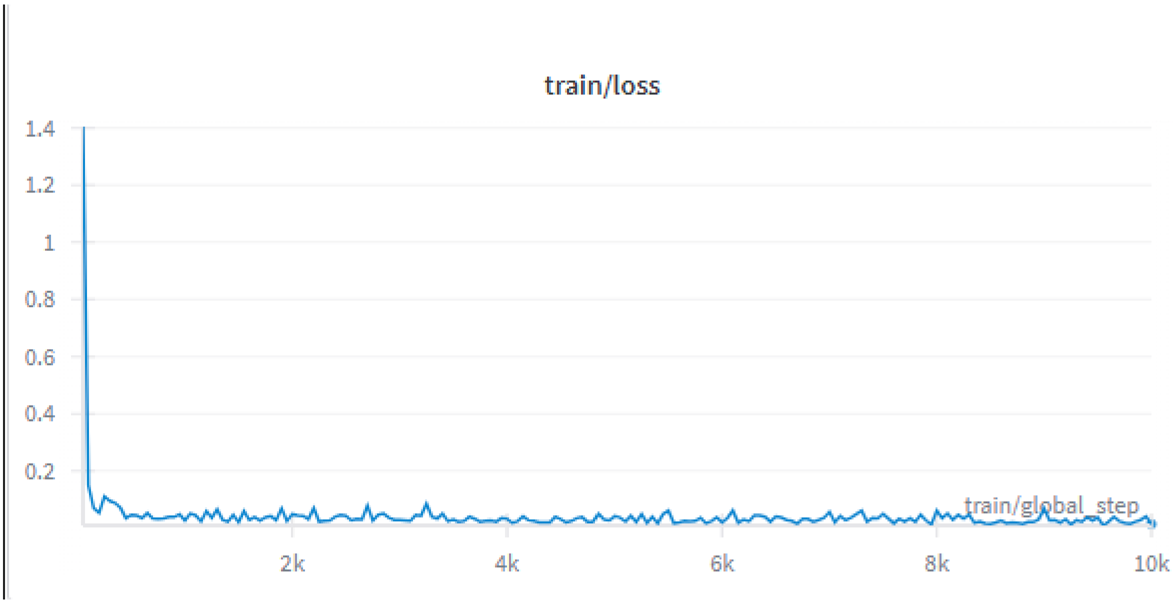
   Purpose: Validate pipeline setup.

   Result: Quick convergence, confirmed learning signal.



2. **Full Run (10,000 steps)**

Purpose: Extended training for better generalization.
Result: Lower validation loss and improved summaries.

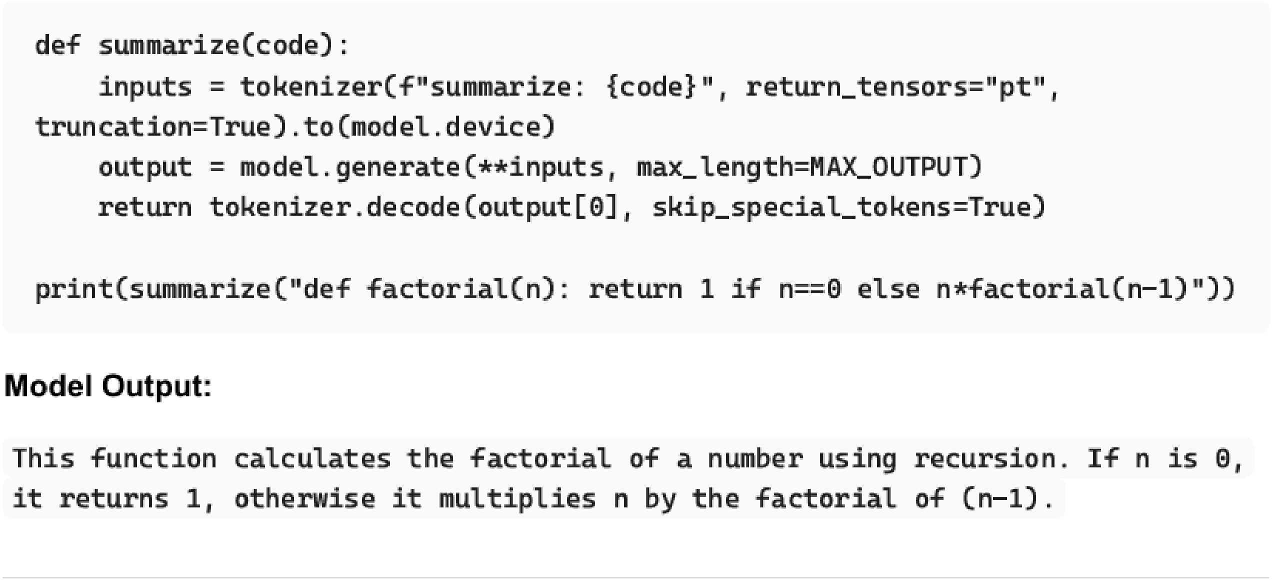


Final Losses:

Training Loss: ~0.0137

Validation Loss: ~0.0654

Inference Example



```
def summarize(code):
    inputs = tokenizer(f"summarize: {code}", return_tensors="pt",
truncation=True).to(model.device)
    output = model.generate(**inputs, max_length=MAX_OUTPUT)
    return tokenizer.decode(output[0], skip_special_tokens=True)

print(summarize("def factorial(n): return 1 if n==0 else n*factorial(n-1)"))
```

Model Output:

```
This function calculates the factorial of a number using recursion. If n is 0,
it returns 1, otherwise it multiplies n by the factorial of (n-1).
```

Advantages of LoRA Fine-Tuning in Codeuctivity

Parameter Efficiency: Only ~0.8% of model weights updated.

Faster Training: Shorter epochs with stable convergence.

Lower Memory Footprint: Enabled fine-tuning on consumer-grade GPUs.

Improved Summaries: Outputs were more fluent and precise compared to baseline CodeT5.

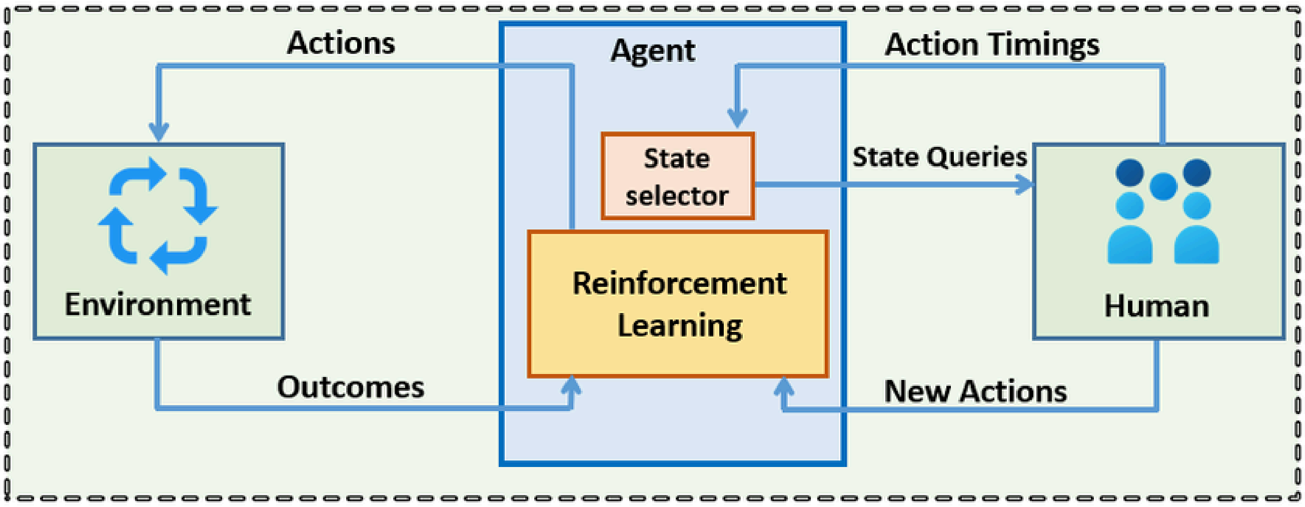
Phase II: some ReaL work done :)

Chapter 1: Reinforcement Learning

1.1 Introduction

Reinforcement Learning (RL) is a paradigm of machine learning where an **agent** interacts with an **environment** to maximize long-term cumulative reward. Unlike supervised learning, which depends on labeled data, RL relies on **trial-and-error interaction** guided by rewards and penalties.

RL is particularly suited for sequential decision-making tasks where each action influences not only the immediate outcome but also future opportunities.



1.2 Core Components

The main building blocks of RL are:

Agent: The learner or decision-maker.

Environment: The system with which the agent interacts.

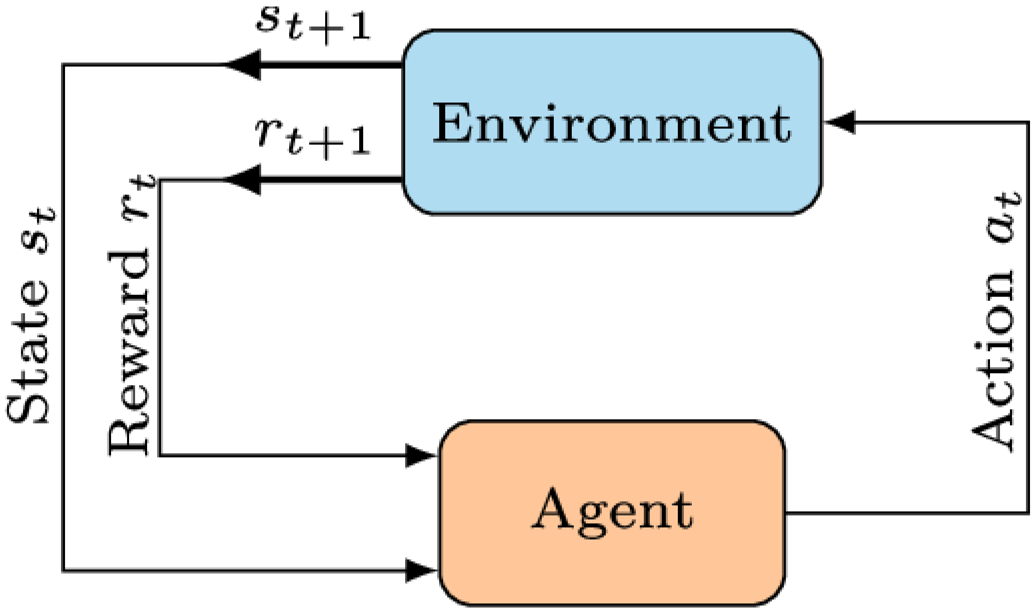
State (s**)**: A representation of the environment at a given time.

Action (a**)**: A decision taken by the agent.

Reward (r**)**: A scalar feedback signal.

Policy (π**)**: A strategy mapping states to actions.

Value Functions (V (s)**,** Q(s, a)**)**: Estimates of long-term returns.



1.3 Formal Framework

RL problems are typically formalized as **Markov Decision Processes (MDPs)**. An MDP is defined by the tuple:

$$M = (S, A, P, R, \gamma)$$

S: set of states

A: set of actions

P(s′∣ s, a): probability of moving to next state s′ from state s when action a is taken

R(s, a): reward function

γ: discount factor, where 0 ≤ γ < 1

The agent’s goal is to learn a policy π(a ∣ s) that maximizes the expected

return: $J(\pi) = E\tau \sim \pi[t = 0 \sum \infty \gamma^t r_t]$

where τ is a trajectory of states, actions, and rewards.

1.4 Types of RL Methods

Value-Based Methods: Learn value functions (e.g., Q-learning, DQN). **Policy-Based Methods**: Optimize the policy directly (e.g., REINFORCE, PPO, GRPO). **Actor–Critic Methods**: Combine both policy optimization and value estimation.

1.5 Exploration vs Exploitation

A central challenge in RL is the **exploration–exploitation trade-off**:

- **Exploration**: Trying new actions to discover better strategies.
- **Exploitation**: Using current knowledge to maximize reward.

Common strategies include ϵ-greedy exploration, entropy regularization, and Thompson Sampling.

1.6 Applications and Challenges

- **Applications**:
  - Robotics (navigation, manipulation)
  - Games (Atari, Chess, Go, Poker)
  - Natural Language Processing (RLHF, style transfer)
  - Recommender systems (personalization, adaptive content)
  - Autonomous driving (planning and control)
- **Challenges**:
  - Sample inefficiency (requires large data)
  - Sparse or delayed rewards
  - Training instability
  - Reward mis-specification
  - Ethical and safety concerns

1.7 Summary

Reinforcement Learning provides a principled framework for sequential decision-making. By modeling problems as MDPs and optimizing policies to maximize expected return, RL enables agents to adapt dynamically to complex environments. Despite its breakthroughs in domains like robotics, games, and AI alignment, challenges such as efficiency, stability, and safety remain open areas of research.

Chapter 2: Reinforcement Learning for Large Language Models

2.1 Introduction

Large Language Models (LLMs) demonstrate impressive general-purpose reasoning and text generation abilities, yet they are often **misaligned with user intent**. Reinforcement Learning (RL) has emerged as a critical technique for aligning LLMs with human-defined objectives such as **helpfulness, safety, style control, and reasoning depth**.

Recent advances — notably **DeepSeek-R1** and **OpenPipe's ART** — illustrate how RL can be adapted specifically for LLM fine-tuning. These works highlight both the **power** and the **flexibility** of RL-driven alignment, whether using preference modeling or rule-based approaches.

2.2 Insights from the DeepSeek-R1 Paper

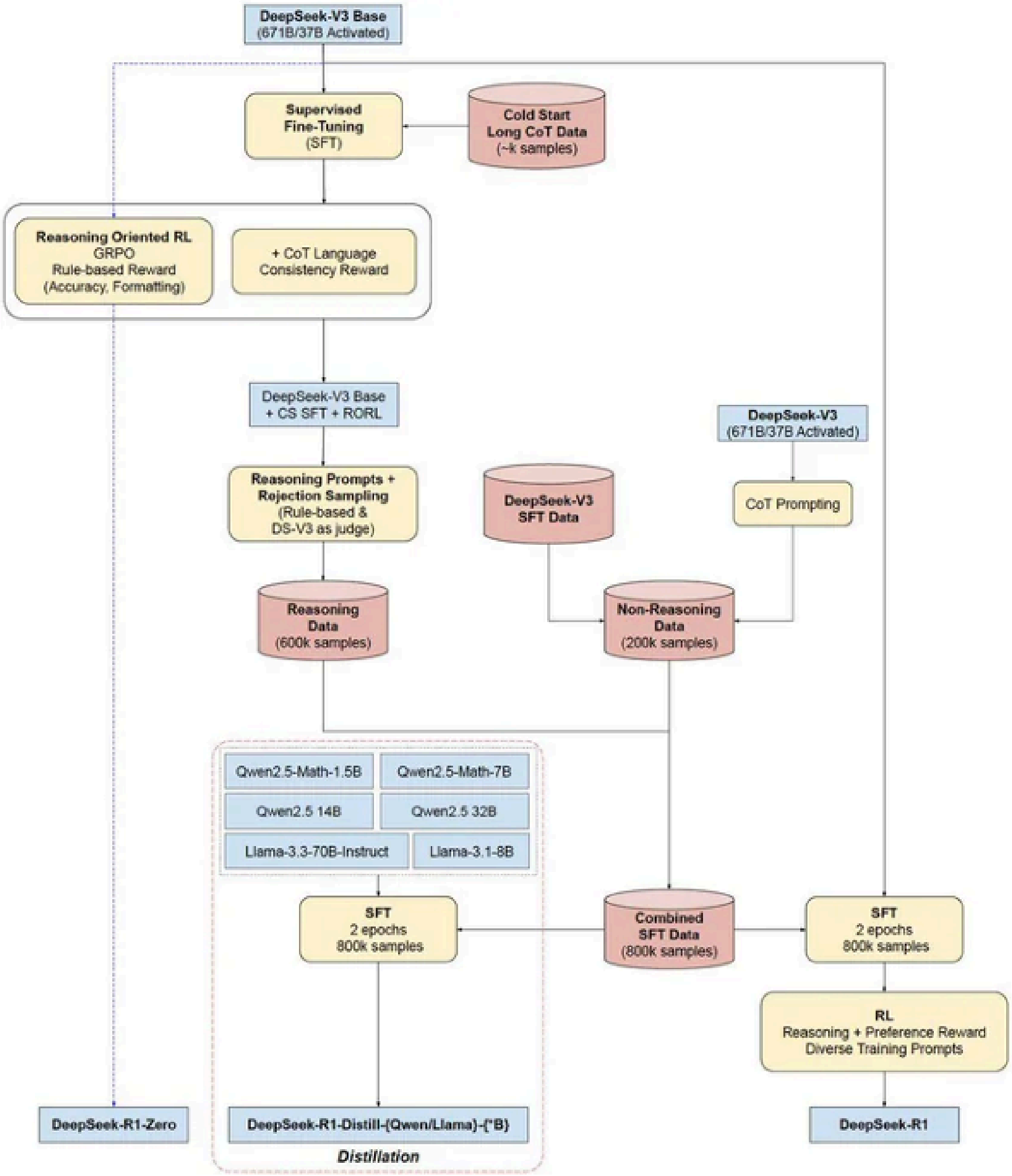
The **DeepSeek-R1** research (2025) presents an LLM fine-tuned with reinforcement learning to enhance reasoning performance. Key contributions include:

Reasoning-Centric Training: Instead of focusing purely on instruction-following, DeepSeek-R1 emphasizes **step-by-step reasoning quality**.

RL for Long-Term Objectives: The agent is trained not just to generate fluent text but to maintain **logical consistency** across multi-step problems.

Scalable Training Loop: By combining **synthetic data** with preference learning and RL optimization, DeepSeek achieves robust performance without requiring massive human annotation.

The DeepSeek-R1 paper demonstrates that RL is not only about stylistic alignment (as in RLHF) but also about **cognitive alignment** — steering LLMs to think in structured, human aligned ways.



2.3 From RLHF to Modular RL

Traditionally, **Reinforcement Learning with Human Feedback (RLHF)** has been the dominant framework:

1. **Supervised Fine-Tuning (SFT)** with curated datasets.
2. **Reward Modeling** from human preference rankings.
3. **Policy Optimization** (commonly PPO).

While effective, RLHF has **limitations**:

Expensive due to human feedback.

Opaque reward models.

Computationally heavy optimization.

DeepSeek-R1 refines this by shifting reward functions toward **reasoning quality** rather than just preference rankings, making it an important evolution of RLHF.

2.4 Rule-Based RL and the ART Framework

A parallel line of work, led by **OpenPipe's ART (Alignment and Reinforcement Trainer)**, reimagines RL for LLMs in a **rule-based, modular form**.

Programmatic Rewards: Instead of opaque preference models, rules can be defined (e.g., "response must be under 50 words" or "must contain a citation"). **Synthetic Data**: Prompts and test cases are generated automatically, reducing dependence on curated datasets.

Flexible Training: By modifying the **task description**, one can train models for entirely new behaviors.

Lightweight RL Algorithms: Methods such as **GRPO (Generalized REINFORCE with Policy Optimization)** allow stable optimization without PPO's complexity.

The ART framework highlights the **engineering pragmatism** of RL for LLMs: making fine tuning interpretable, efficient, and customizable.

2.5 RL Algorithms in LLM Training

RL algorithms adapted for LLMs include:

PPO (Proximal Policy Optimization): Widely used in RLHF, but computationally demanding.

GRPO (Generalized REINFORCE with Policy Optimization): A simpler alternative with KL-regularization to prevent divergence from the base model.

The GRPO objective is:

$$J(\pi\theta) = E_(s, a) \sim \pi\theta[r(s, a) - \beta \cdot KL(\pi_\theta \parallel \pi_r ef)]$$

where:

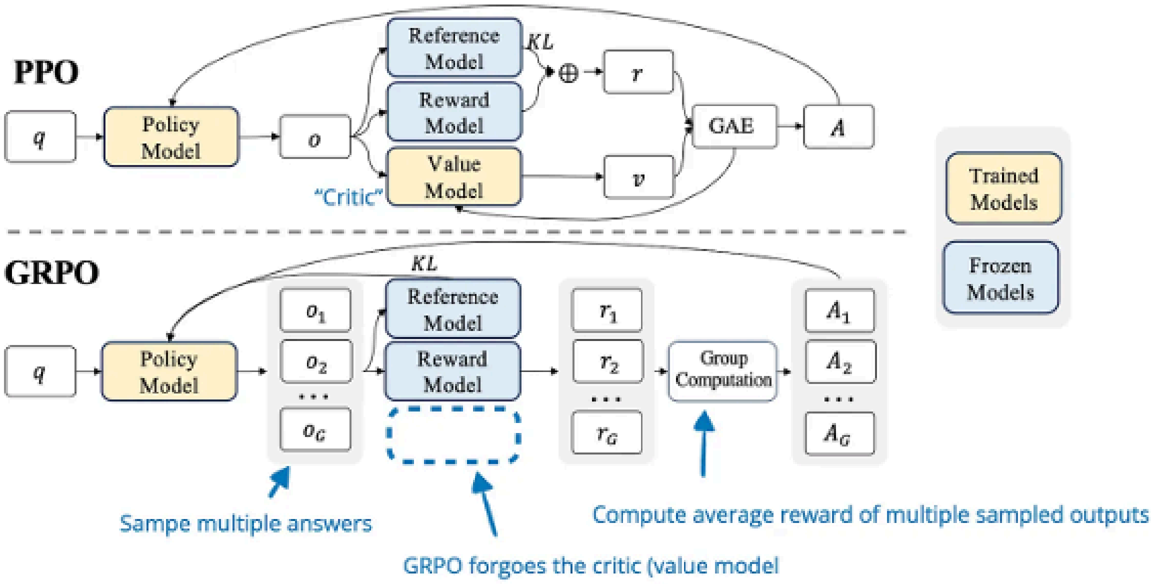
π_θ: the fine-tuned policy

π_{ref}: the reference/base model

r(s, a): the reward signal

β: the KL penalty coefficient

This balances **improvement** (reward maximization) with **stability** (staying close to the original model).



2.6 Summary

Reinforcement Learning has become essential in shaping LLM behavior. The **DeepSeek-R1 paper** demonstrates RL's potential for **reasoning alignment**, while **ART** introduces **rule based modular training** for flexible alignment. Together, they illustrate how RL can move beyond supervised paradigms toward **interpretable, efficient, and reasoning-focused fine-tuning**.

Both approaches — preference-based (DeepSeek) and rule-based (ART + RULER) — enrich the toolkit for LLM alignment, marking RL as a cornerstone of the next generation of model training.

Chapter 3: Demystifying GRPO (Group Relative Policy Optimization)

3.1 Introduction

In reinforcement learning (RL) for large language models (LLMs), *Group Relative Policy Optimization (GRPO)* has emerged as an efficient and effective alternative to classical methods such as PPO. Originally introduced in reasoning-focused training (e.g., DeepSeekMath), GRPO reduces compute and memory requirements while maintaining strong alignment performance.

3.2 Motivation: From PPO to GRPO

Proximal Policy Optimization (PPO) has been the dominant algorithm for RLHF but faces key challenges:

Requires a critic (value) network, which increases complexity and memory usage.

Advantage estimation is expensive and unstable.

Training large models with PPO is resource-intensive.

GRPO addresses these issues by:

Removing the critic network entirely.

Computing **advantages** using group-normalized rewards.

Simplifying optimization while retaining PPO’s stability benefits.

3.3 How GRPO Works

The GRPO training loop works as follows:

1. **Sampling**: For a given input (prompt) q, sample a **group** of outputs o_1, o_2,…, o_G from the old policy $\pi_{\theta_{old}}$.
2. **Reward Assignment**: Compute a reward r_i for each output o_i.
3. **Advantage Estimation**: Normalize rewards within the group to compute the advantage:

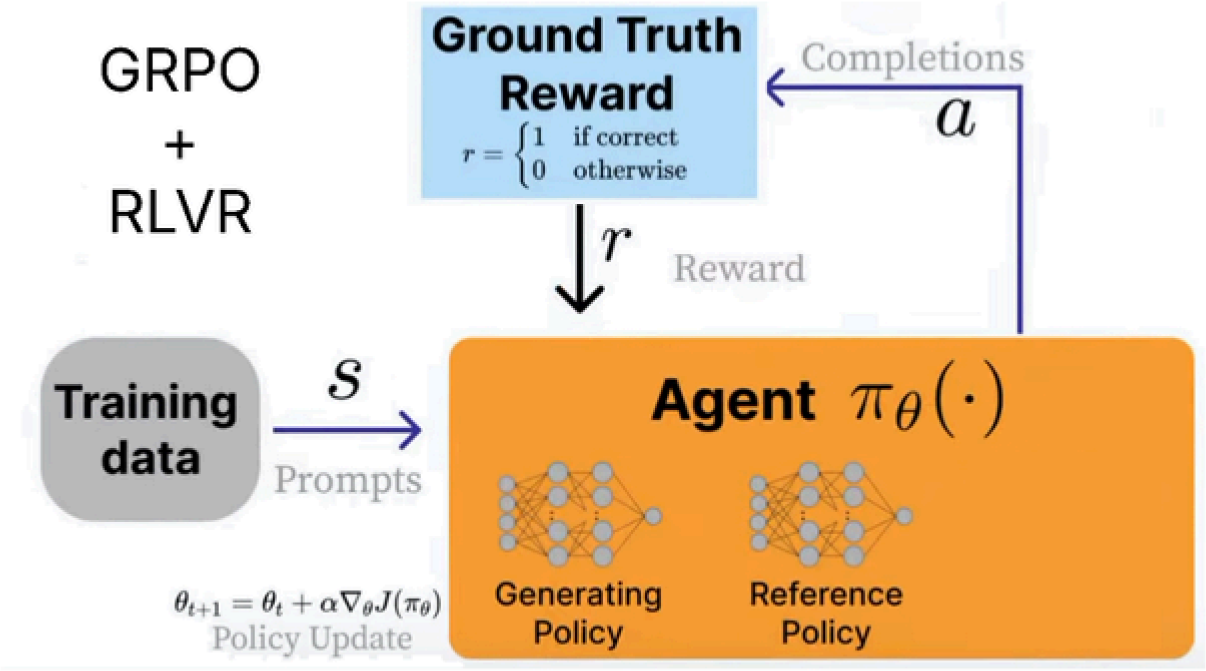
$$Ai = ri - mean(r1, \ldots, rG)std(r1, \ldots, rG)A_i = \frac{r_i - \mathrm{mean}(r_1, \ldots, r_G)}{\mathrm{std}(r_1, \ldots, r_G)}$$

4. **Policy Update**: Update the policy π_θ using a clipped surrogate objective with KL regularization:

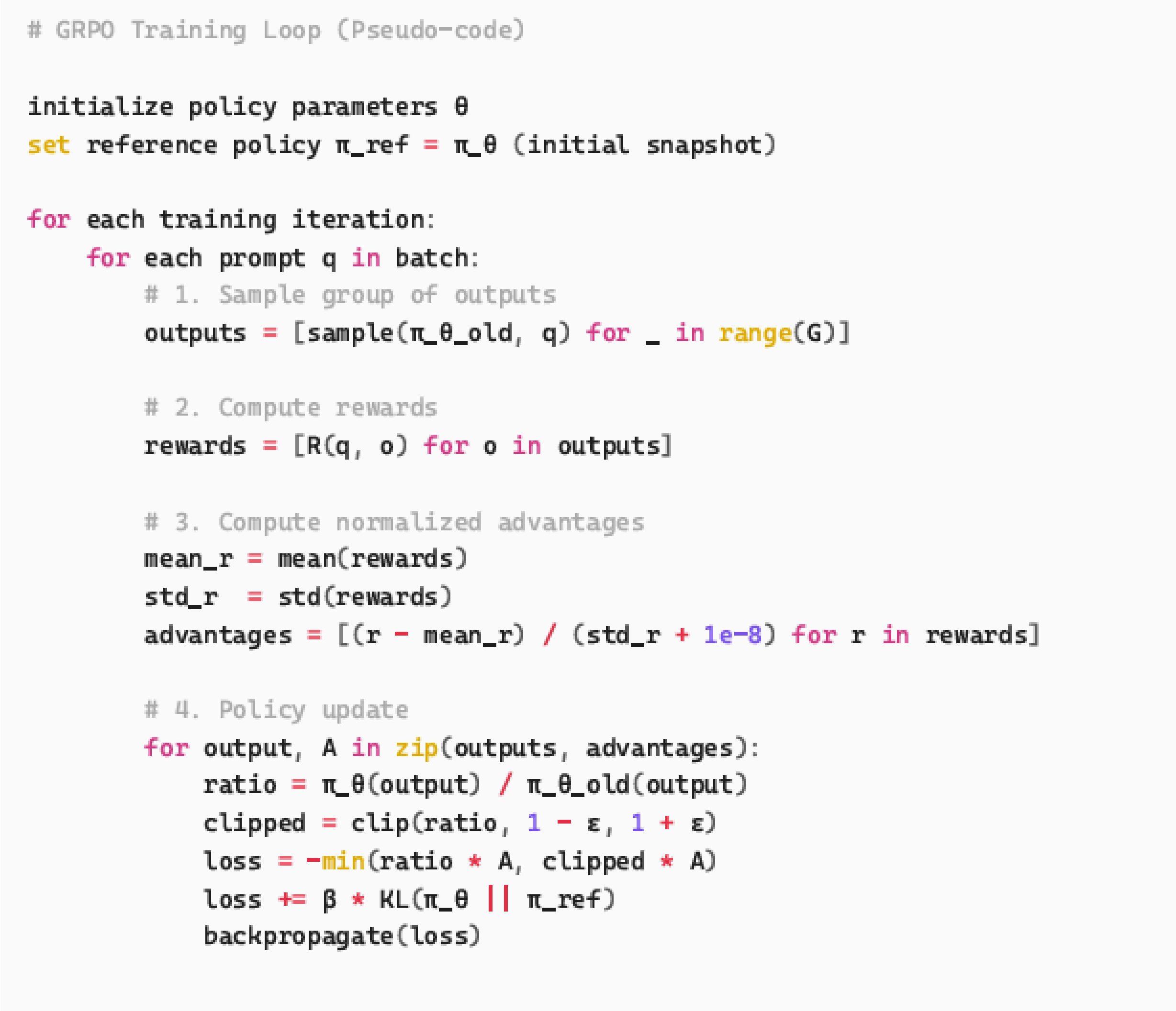
$$J(\pi\theta) = E(s,a) \sim \pi\theta[min(rt(\theta)At, clip(rt(\theta), 1-\epsilon, 1+\epsilon)At) - \beta KL(\pi\theta \parallel \pi ref)]$$

where:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio.
- ϵ controls the clipping range.
- β is the KL penalty coefficient.


GRPO
+
RLVR
Ground Truth Reward
$r = \begin{cases} 1 & \text{if correct} \\ 0 & \text{otherwise} \end{cases}$
Completions
a
r
Reward
Training data
s
Prompts
Agent $\pi_\theta(\cdot)$
Generating Policy
Reference Policy
$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\pi_\theta)$
Policy Update

3.4 Pseudo-Code for GRPO



```
# GRPO Training Loop (Pseudo-code)

initialize policy parameters θ
set reference policy π_ref = π_θ (initial snapshot)

for each training iteration:
    for each prompt q in batch:
        # 1. Sample group of outputs
        outputs = [sample(π_θ_old, q) for _ in range(G)]

        # 2. Compute rewards
        rewards = [R(q, o) for o in outputs]

        # 3. Compute normalized advantages
        mean_r = mean(rewards)
        std_r  = std(rewards)
        advantages = [(r - mean_r) / (std_r + 1e-8) for r in rewards]

        # 4. Policy update
        for output, A in zip(outputs, advantages):
            ratio = π_θ(output) / π_θ_old(output)
            clipped = clip(ratio, 1 - ε, 1 + ε)
            loss = -min(ratio * A, clipped * A)
            loss += β * KL(π_θ || π_ref)
            backpropagate(loss)
```

3.5 Advantages of GRPO

Benefit	Explanation
No Critic Model	Removes the need for a value network, reducing complexity.
Stable Advantage Estimation	Group-normalized rewards act as a variance-reducing baseline.
Compute Efficiency	Lower memory and compute costs than PPO.
Effective in Reasoning Tasks	Demonstrated strong results on math and logic benchmarks.

3.6 Recent Extensions

S-GRPO: Applies decaying rewards during chain-of-thought to encourage concise reasoning.

Spectral Policy Optimization: Improves diversity when all sampled outputs are poor.

AGPO (Adaptive GRPO): Adjusts reward normalization and KL penalties dynamically for stability.

3.7 Summary

GRPO simplifies reinforcement learning for LLMs by removing the critic and using **group relative reward normalization** for advantage estimation. It balances stability and efficiency, making it highly suitable for large-scale LLM fine-tuning. With recent extensions, GRPO continues to evolve as a core method for aligning LLMs in reasoning and stylistic tasks.

Chapter 4: RULER — Rule-Based Rewards for LLM Fine-Tuning

4.1 Introduction

A core challenge in applying reinforcement learning to large language models (LLMs) is crafting appropriate reward functions. Traditional methods involve training complex reward models or hand-crafting scoring rules—both costly and inflexible.

RULER (Relative Universal LLM-Elicited Rewards) offers a compelling alternative: an **LLM-as-judge** evaluates multiple output trajectories and ranks them, requiring no labeled data or manual reward engineering. It synergizes naturally with GRPO, which only relies on relative scores within grouped outputs.

4.2 Why RULER Works

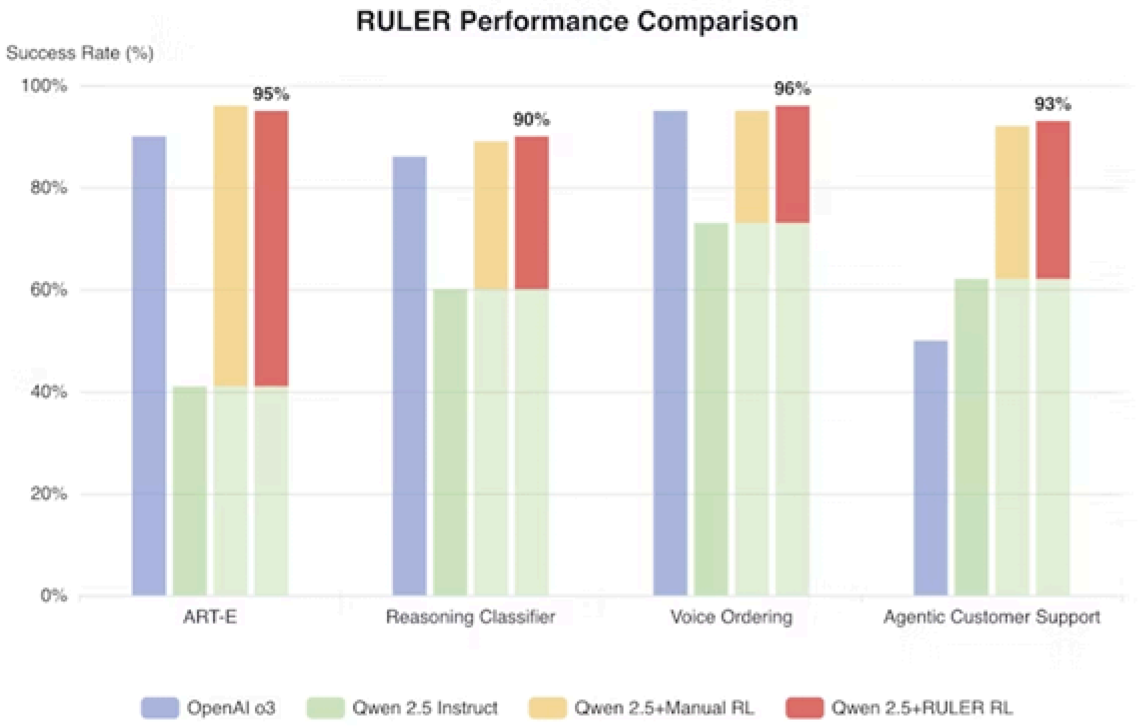
RULER works effectively due to two principles:

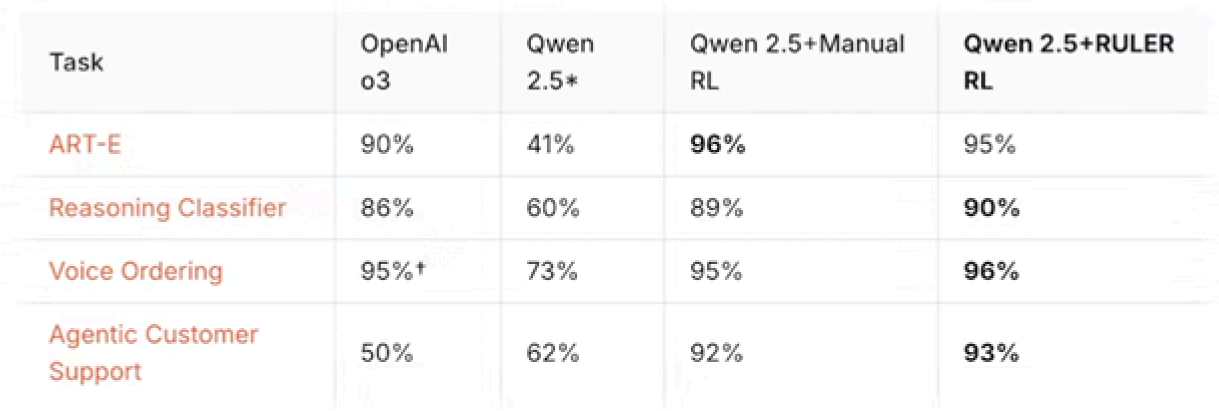
1. **Relative scoring is easier for LLMs**: It’s simpler for an LLM to compare and rank outputs than to assign absolute quality scores.
2. **GRPO only needs relative information**: GRPO’s normalized advantage calculation requires only comparative ranking, not calibrated reward values.

Here’s how the RULER process unfolds:

1. Generate *N* candidate trajectories from the agent for a given prompt. 2. Pass all
trajectories to RULER, which compares and deduplicates redundant prefixes. 3. An LLM
judge assigns each trajectory a score in [0,1] based on adherence to task specific
criteria (e.g., fluency, style, coherence).

4. These scores serve as direct rewards for GRPO updates.





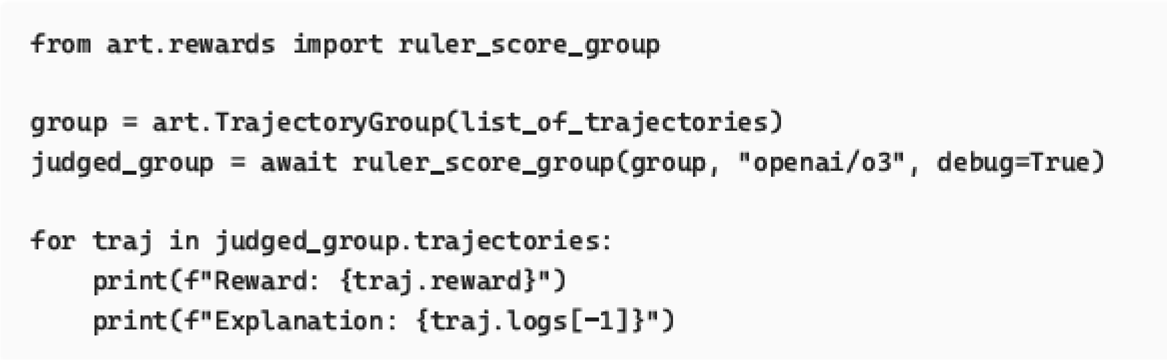
Task	OpenAI o3	Qwen 2.5*	Qwen 2.5+Manual RL	**Qwen 2.5+RULER RL**
ART-E	90%	41%	**96%**	95%
Reasoning Classifier	86%	60%	89%	**90%**
Voice Ordering	95%†	73%	95%	**96%**
Agentic Customer Support	50%	62%	92%	**93%**

4.3 Key Benefits of RULER

Feature	**Benefit**
No labeled data	Removes need for human-annotated preference datasets.
Task-agnostic	Works across diverse tasks without re-engineering.
Rapid development	Cut reward design time by ~2–3× compared to manual methods.
High performance	Matches or exceeds success of hand-crafted rewards in most tasks.

4.4 Integration with ART Framework

Within the **ART (Agent Reinforcement Trainer)** ecosystem, using RULER is seamless:



```python
from art.rewards import ruler_score_group

group = art.TrajectoryGroup(list_of_trajectories)
judged_group = await ruler_score_group(group, "openai/o3", debug=True)

for traj in judged_group.trajectories:
    print(f"Reward: {traj.reward}")
    print(f"Explanation: {traj.logs[-1]}")
```

This makes adopting rule-based rewards as simple as a one-line function call—no extra reward function coding needed.

4.5 Applications and Cautions

RULER shines in scenarios like multi-step reasoning, style enforcement, and workflow driven tasks. It’s excellent for rapid experimentation and semantic/objective alignment.

However, be cautious when using RULER in **high-stakes or safety-critical environments**. LLM judges can be inconsistent, sensitive to prompt phrasing, and vulnerable to adversarial outputs or reward hacking. Mitigations include:

Calibration of the judge (e.g., temperature control, ensemble voting)

Decomposing rewards into subcomponents (e.g., coherence, safety)

Hybrid evaluation combining rule-based and human review for critical use cases

4.6 Summary

RULER provides a **flexible, interpretable, and efficient** way to generate reward signals for LLMs, especially when paired with GRPO in the ART framework. It enables quick deployment across various tasks, bypassing the challenges of reward engineering and preference modeling. While powerful, practical deployment should include validation steps to ensure robustness and safety.

Chapter 5: Case Study — First Attempt at GRPO with TRL

5.1 Objective

The goal of this project was to explore reinforcement learning (RL) for language model alignment using **Group Relative Policy Optimization (GRPO)** in the Hugging Face **TRL library**. The chosen experiment was to fine-tune a lightweight model (**Qwen2-0.5B**) on a summarization task, hoping GRPO would improve output quality by optimizing against reward signals rather than relying solely on supervised fine-tuning.

5.2 Experimental Setup

Dataset: `trl-lib/tldr` summarization dataset.

Model: Qwen2-0.5B.

Framework: Hugging Face **TRL** with `GRPOTrainer` .

Workflow:

1. Install TRL.
2. Load TL;DR dataset.
3. Initialize GRPO configuration and trainer.
4. Train and save checkpoint (`Qwen2-0.5B-GRPO-final`).
5. Run inference on test inputs.

5.3 Observations

The model did not show meaningful improvements over the base checkpoint.

Generated summaries were often repetitive or low-quality.

Reward signals were weak or inconsistent, leading to unstable training. The process still **depended heavily on an existing dataset**, which contradicted the original aim of exploring RL as a more flexible alignment method.

Table: Inference Results from Mini-Project (Qwen2-0.5B with GRPO Trainer)

Input (Article Snippet)	**Base Model Output**	**GRPO Fine Tuned Output**
"The stock market experienced heavy volatility today, with tech shares swinging widely. Investors remain cautious."	**TL;DR:** The stock market was volatile, tech shares fluctuated, investors cautious.	**TL;DR:** Stocks moved a lot. Investors scared.

Input (Article Snippet)	**Base Model Output**	**GRPO Fine Tuned Output**
"Scientists have developed a new vaccine that shows promising results in early trials, potentially preventing a major illness."	**TL;DR:** New vaccine works in early trials, may prevent disease.	**TL;DR:** A vaccine was made. Trials happened.
"The local football team secured a dramatic late win, sending fans into celebration across the city."	**TL;DR:** Local team won late, fans celebrated.	**TL;DR:** The team won. Fans were happy.
"Heavy rains have caused flooding in several regions, with emergency crews working around the clock to evacuate residents."	**TL;DR:** Flooding hit regions, emergency crews evacuated people.	**TL;DR:** Flood. Crews help.
"A new policy has been announced to reduce carbon emissions, focusing on renewable energy investment and stricter industry regulations."	**TL;DR:** Policy to cut emissions via renewable investment and stricter rules.	**TL;DR:** Policy for emissions. Energy rules.

5.4 Mentor Feedback

When discussing these results with a mentor, an important insight emerged:

> **This approach still relies on a fixed dataset.**
>
> The real power of RL in LLMs comes from **synthetic data generation** and **rule-based evaluation** rather than fine-tuning on existing corpora.
>
> Approaches like **RULER + GRPO** (or frameworks such as ART) allow models to self generate prompts, evaluate their own outputs, and learn without requiring curated datasets.

This feedback highlighted why the initial attempt felt limited: it was closer to **classic supervised fine-tuning with reinforcement signals added**, rather than a fully RL-driven workflow.

5.5 Lessons Learned

TRL’s GRPOTrainer is useful but not sufficient on its own: without synthetic generation and carefully designed rewards, it risks collapsing into dataset-bound training.

Dataset-free RL is more aligned with modern research: methods like DeepSeek-R1 and ART demonstrate how synthetic prompts and rule-based rewards can drive alignment without curated datasets.

Future direction: move beyond static datasets toward **synthetic data generation, RULER-based evaluation, and flexible task descriptions** that can adapt dynamically.

5.6 Summary

This first GRPO attempt with TRL was a valuable **failure**. It showed the fragility of RL training when tethered to fixed datasets and underscored the need for **synthetic, rule based rewards** in LLM alignment. With this lesson, the next phase of exploration shifts toward dataset-free RL, leveraging tools like **RULER** and **ART** to unlock the real potential of reinforcement learning for LLM fine-tuning.

Chapter 6 — Refine: **Synthetic RL Fine-Tuning for LLM Alignment**

6.1 Abstract

Refine is a dataset-free, synthetic-data reinforcement learning pipeline designed to fine-tune large language models (LLMs) for targeted behavior (e.g., style transfer, tone control, and reasoning). The system combines three key components: (1) synthetic prompt generation via an external LLM API, (2) rule-based evaluation using an LLM-as-judge (RULER), and (3) policy optimization using GRPO — all orchestrated with the ART (Alignment & Reinforcement Trainer) toolkit. This approach enables rapid task adaptation by changing a single task description and avoids costly human annotation by letting the model and auxiliary LLMs generate training inputs and rewards.

6.2 Motivation & Context

Prior experiments showed that naively applying GRPO on a static summarization dataset (TRL + Qwen2-0.5B) did not yield consistent improvements: reward signals were weak and the fine-tuned model often collapsed to short or trivial outputs. A mentor recommended pursuing a dataset-free approach: if the model can **generate its own training inputs** and the evaluation is **relative and programmatic**, RL can be leveraged to learn behaviors beyond what supervised fine-tuning alone provides. This insight motivated Refine's architecture: synthetic data + RULER + GRPO.

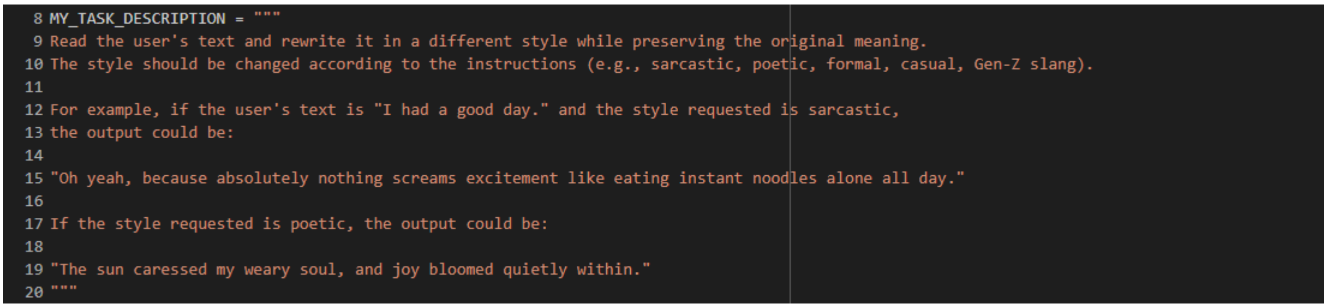
6.3 Contributions of Refine

1. **Dataset-free RL pipeline**: Models are trained using prompts generated on the fly via an API (OpenRouter or equivalent).
2. **Rule-based, LLM-assisted reward (RULER)**: An LLM grader ranks candidate outputs within trajectory groups, producing relative rewards well-suited for GRPO. 3. **GRPO optimization**: Effective group-relative advantage estimation eliminates the need for a critic and makes RL feasible at smaller compute budgets.
4. **Task modularity**: Changing the `TASK_DESCRIPTION` reconfigures the whole pipeline for a new behavior (style transfer, summarization, safety filters, etc.).
5. **Proof-of-concept**: A trained model (`rl-model-002`) capable of style-aware rewriting (Gen-Z slang, Shakespearean, haiku, formal diplomatic, sarcastic breakups) demonstrates practical capability.

6.4 Methodology

6.4.1 Task definition

A human-readable TASK_DESCRIPTION defines the objective (for example: “Rewrite this text in the requested style while preserving meaning”). The same pipeline is reused for other tasks by editing this single string.



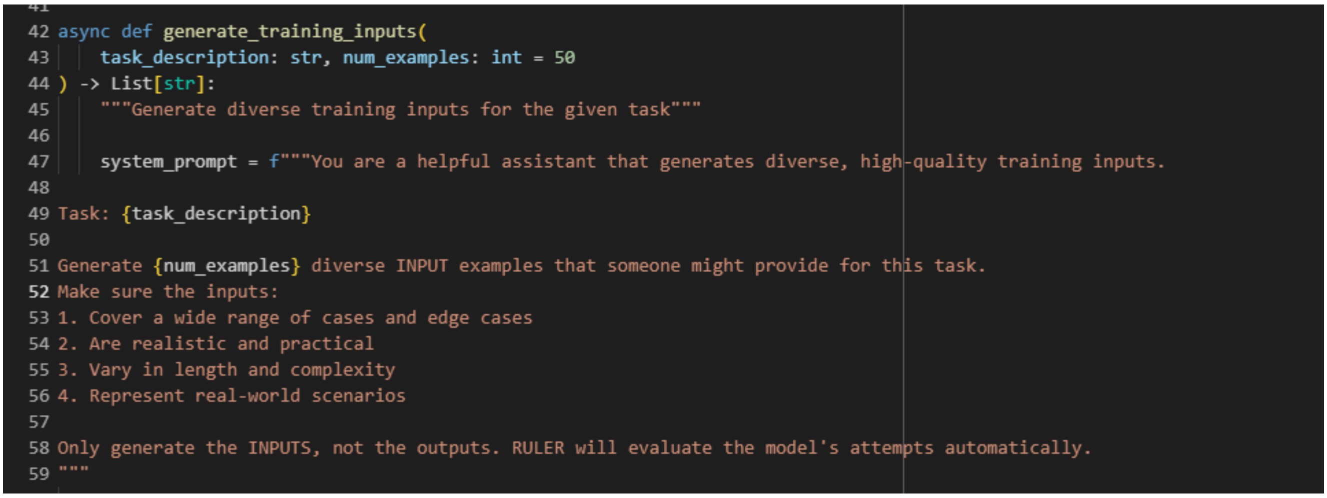
```
MY_TASK_DESCRIPTION = """
Read the user's text and rewrite it in a different style while preserving the original meaning.
The style should be changed according to the instructions (e.g., sarcastic, poetic, formal, casual, Gen-Z slang).

For example, if the user's text is "I had a good day." and the style requested is sarcastic,
the output could be:

"Oh yeah, because absolutely nothing screams excitement like eating instant noodles alone all day."

If the style requested is poetic, the output could be:

"The sun caressed my weary soul, and joy bloomed quietly within."
"""
```

6.4.2 Synthetic prompt generation

Use an external LLM (via OpenRouter / API) to generate both training and test prompts from the TASK_DESCRIPTION .

Generation settings (model name, temperature, top-k/top-p) are tunable in the notebook configuration.

This step creates a stream of diverse task examples without manual curation.



```
async def generate_training_inputs(
    task_description: str, num_examples: int = 50
) -> List[str]:
    """Generate diverse training inputs for the given task"""

    system_prompt = f"""You are a helpful assistant that generates diverse, high-quality training inputs.

Task: {task_description}

Generate {num_examples} diverse INPUT examples that someone might provide for this task.
Make sure the inputs:
1. Cover a wide range of cases and edge cases
2. Are realistic and practical
3. Vary in length and complexity
4. Represent real-world scenarios

Only generate the INPUTS, not the outputs. RULER will evaluate the model's attempts automatically.
"""
```

6.4.3 Candidate generation (trajectory groups)

For each prompt, sample a **group** of candidate responses (e.g., G = 8). Group diversity is important; sample temperature and nucleus sampling encourage variety.

6.4.4 Rule-based evaluation (RULER)

Send the whole candidate group to a judge LLM (RULER), which compares outputs and assigns relative scores.

Scoring criteria are task-specific (style adherence, semantic fidelity, brevity, safety).

RULER returns normalized scores in [0,1] for each trajectory.

6.4.5 Policy optimization with GRPO

Convert RULER scores into group-relative advantages: normalize within each group (mean/std).

Use the GRPO surrogate objective with optional KL penalty to update the policy parameters.

No separate value network (critic) is required — the group normalization serves as a baseline.

6.5 Inference Results

After training, the fine-tuned model (`rl-model-002`) was tested on five unseen prompts. The outputs demonstrate **style transfer capability** while preserving semantic meaning.

Side-by-Side Comparison

Test Input (with style instruction)	**Base Model Output**	**Refine (GRPO fine-tuned) Output**
The server is down, nobody can log in remotely. Major issue. (Style: humorous Gen-Z slang)	*“The server is down. Nobody can log in. This is a major issue.”*	*“The server is down, so we’re all stuck at home. Total chaos for the team — someone fix this pronto ��.”*
Your manuscript does not meet our publication standards. (Style: Shakespearean tragic monologue)	*“We cannot publish your manuscript. Thank you for your submission.”*	*“Ah, poor soul, alas, thy work finds no favor with our council. Accept this with dignity, for we must part ways with heavy heart.”*
Hey dude, wanna grab burritos later? Maybe around 7? (Style: formal diplomatic correspondence)	*“Let’s meet at 7 for burritos.”*	*“Dear Sir, I would be honored to join you for burritos at precisely 7 PM. Kindly confirm your availability.”*
The cat knocked over my coffee mug; brown liquid everywhere. (Style: haiku)	*“The cat spilled coffee. My keyboard is wet.”*	*“The cat again spills, / coffee staining the keyboard, / morning chaos reigns.”*

We confirm receipt of your job application. (Style: sarcastic breakup text)	*“We received your job application.”*	*“Congrats, you’ve officially been dumped — I mean, your application is received. Don’t wait up for a call.”*

Observations

Base Model: Outputs are literal, flat, and style-agnostic.

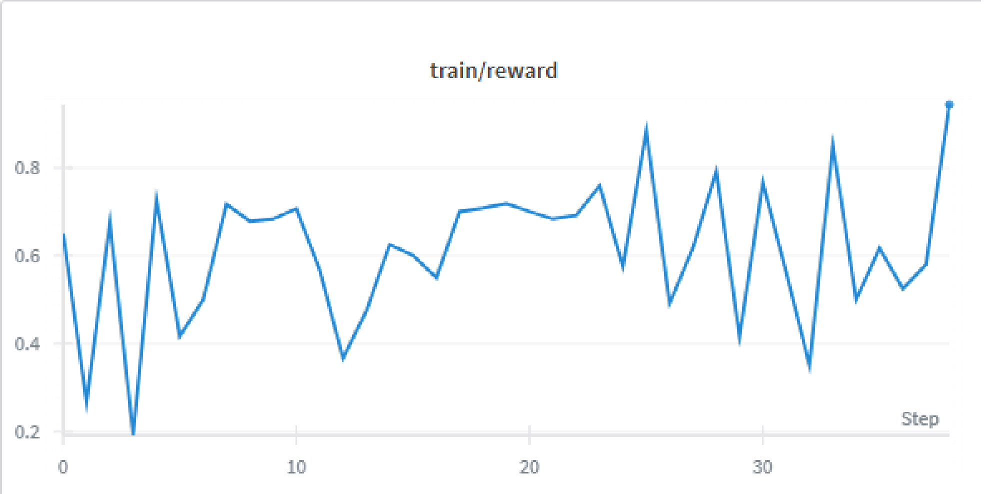
Refine: Outputs exhibit strong style adaptation while maintaining semantic fidelity.

Improvement: Refine shows **clear stylistic control** (humor, Shakespearean, haiku, sarcasm) — capabilities absent in the base model.

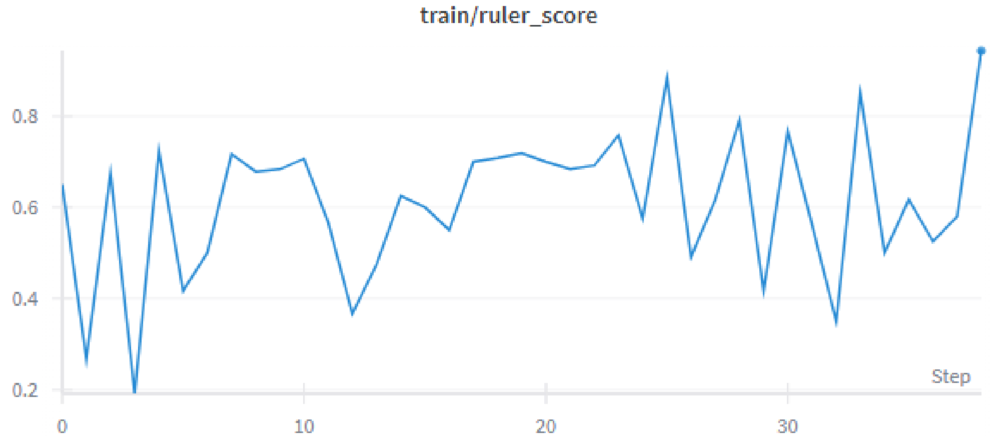
6.6 Quantitative Tracking (placeholders)

To complement qualitative outputs, the following graphs should be inserted:

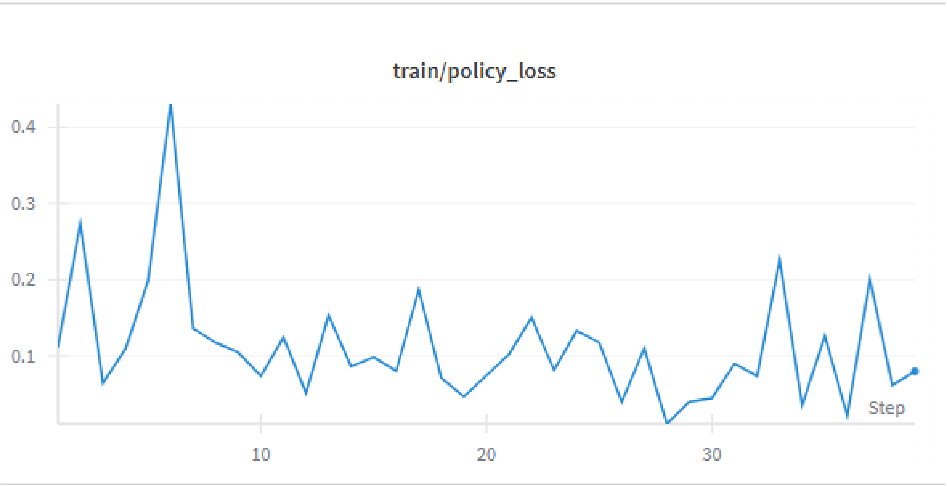
Reward vs. training step



RULER vs. training step



Policy loss vs. training step



6.7 Limitations & Future Work

Judge sensitivity: RULER’s evaluation depends on judge model stability.

Reward hacking risk: Model could exploit superficial scoring cues.

Compute/API costs: Synthetic generation and ruling are non-trivial in expense.

Next steps: judge ensembles, multi-objective rewards, LoRA+GRPO hybrids, and human validation studies.

6.8 Conclusion

Refine demonstrates how dataset-free reinforcement learning, powered by synthetic data and rule-based evaluation, can align LLM behavior effectively. By decoupling from fixed datasets and enabling task modularity through prompt-based task definitions, Refine provides a blueprint for scalable, transparent, and flexible LLM fine-tuning.

Chapter 7 — Conclusion and Reflections 7.1 Journey Overview

This project began with a simple curiosity: *how can we fine-tune large models to perform specific tasks more effectively?* Over the course of two months, the journey evolved from foundational machine learning concepts to advanced reinforcement learning for LLM alignment. Along the way, we explored supervised fine-tuning, LoRA/QLoRA, reinforcement learning with GRPO, rule-based rewards via RULER, and synthetic data generation using external LLMs.

The journey can be divided into **two distinct phases**:

1. **Codeuctivity Project** (CodeT5 + LoRA/QLoRA for code summarization)
   - Focused on understanding the fundamentals of fine-tuning.
   - Gave practical exposure to PyTorch, gradient descent, neural networks, and transformers.
   - Used LoRA and QLoRA to fine-tune CodeT5 on code summarization tasks.
   - Outcome: gained confidence in supervised fine-tuning but realized its limits — heavily dataset-dependent, and not always flexible.
2. **Refine Project** (RL for LLMs using GRPO + RULER + synthetic data)
   - Introduced to reinforcement learning as a method for aligning LLMs.
   - Initial attempt with TRL’s GRPO trainer showed weaknesses: still dataset-bound, with little difference from SFT.
   - Shifted to a dataset-free RL pipeline using **synthetic data** (generated via OpenRouter API), **rule-based rewards** (RULER), and **policy optimization** (GRPO).
   - Outcome: successfully trained `rl-model-002` capable of **style transfer** (humorous, Shakespearean, diplomatic, haiku, sarcastic) — a powerful demonstration of controllable generation.

7.2 Key Learnings

From Codeuctivity (CodeT5 project)

- Practical understanding of fine-tuning techniques (LoRA, QLoRA).
- Hands-on experience in training pipelines with supervised objectives.
- Realized the dependence on static datasets and the limitations of supervised-only approaches.

From Refine (RL project)

Learned how reinforcement learning can align LLM outputs to complex objectives.

Understood the importance of reward design: synthetic data + rule-based rewards made the system flexible.

GRPO proved to be a lightweight yet effective algorithm for LLM policy optimization.

Discovered the modularity of ART: simply changing the task description could yield entirely new trained behaviors.

7.3 Broader Impact

This journey reflects the broader trend in AI research:

Moving **beyond supervised fine-tuning** toward **reinforcement learning for alignment**.

Exploring **synthetic data** as a way to overcome dataset bottlenecks.

Using **LLMs as judges** (RULER) to define interpretable, programmatic reward signals.

Emphasizing **modularity and adaptability**: a single pipeline (Refine) can be repurposed for many tasks.

7.4 Limitations

Codeuctivity Project: Success was bounded by dataset availability and supervised objectives.

Refine Project: While powerful, synthetic data pipelines depend on external APIs, can be costly, and rely on the stability of the judge LLM (RULER).

Human evaluation is still necessary to fully validate outputs in high-stakes scenarios.

7.5 Final Reflection

This project, **Refine**, represents both a technical milestone and a personal learning arc:

From learning the basics of gradient descent and PyTorch → to building supervised models with LoRA/QLoRA → to exploring reinforcement learning for LLMs. From failures (mini-project collapse with TRL's GRPO) → to breakthroughs (synthetic pipeline with RULER + GRPO).

The result is a system that demonstrates the **future of fine-tuning**: not constrained by datasets, but dynamically adaptable, explainable, and capable of aligning LLMs with nuanced human preferences.

Refine is not just a project, but a **proof of concept** that RL with synthetic data and rule based rewards can democratize fine-tuning — enabling anyone to train models for highly specific tasks without massive annotated datasets.

Building **Re:fine** has been a journey of curiosity, collaboration, and inspiration. This project wouldn't have been possible without the ideas, research, and resources shared by incredible minds and platforms in the tech and AI community.

We would like to express our gratitude to:

DeepSeek R1 Paper – for laying the foundation of advanced search and recommendation systems.
Read the paper here: https://arxiv.org/abs/2501.12948

ART Paper & ART Blogs – for insights into adaptive reasoning techniques and innovative AI research.
Read the paper here: https://arxiv.org/abs/2505.01441
Explore the blog here: https://art.openpipe.ai/getting-started/about

Andrew Ng's Courses – for the invaluable structured learning in AI, ML, and deep learning.
Access the courses here: https://www.coursera.org/instructor/andrewng

Medium Articles & Thought Leadership – for sharing practical insights, tutorials, and emerging trends in AI and productivity.
Explore Medium articles here:
https://medium.com/data-science-in-your-pocket/what-is-grpo-the-rl-algorithm-used-to-train-deepseek-12acc19798d3