Comparison of Standard Fine-Tuning and Reflection-Based Meta-Learning for Code Generation

Author: Arda Mülayim

1. Introduction

1.1 Background and Motivation

Large Language Models (LLMs) have demonstrated remarkable capabilities in code generation tasks, fundamentally transforming software development practices. Models like GitHub Copilot and ChatGPT have shown that LLMs can generate syntactically correct code from natural language descriptions. However, the optimal training methodology for teaching models to generate not just correct but semantically meaningful and robust code remains an open research question. Traditional supervised fine-tuning approaches, while effective for pattern matching, may not fully capture the nuanced understanding required for handling edge cases and debugging capabilities.

1.2 Research Approach

This study presents a comprehensive comparative analysis of two distinct training approaches for Java code generation using LLaMA 3.2 3B: Standard Supervised Fine-Tuning (SFT) and a novel Reflection-Based Meta-Learning approach. The key insight motivating this work stems from observing human learning processes in programming education. When human programmers learn to code, they don't merely memorize correct examples—they actively learn from their mistakes through feedback, reflection, and understanding why certain approaches fail. This observation led me to hypothesize that incorporating this error-based learning paradigm into model training could significantly improve both code generation quality and semantic understanding.

1.3 Teacher-Student Paradigm

My approach introduces a teacher-student reflection paradigm where the model learns not only from correct implementations but also from structured error analysis. By training on 10,000 examples where

Claude-4-Sonnet acts as a teacher providing detailed feedback on incorrect attempts, I aim to instill deeper code comprehension beyond surface-level pattern matching.

1.4 Research Questions

The primary research questions guiding this investigation are:

1. Can a reflection-based meta-learning approach that explicitly incorporates error analysis outperform standard supervised fine-tuning for code generation tasks?
2. Does training on error analysis examples provide additional capabilities beyond the primary generation task, such as debugging abilities?
3. What is the impact of different training paradigms on model consistency, reliability, and understanding of code semantics?

1.5 Key Findings

Through extensive experiments on 100,000 examples from the Microsoft CodeXGLUE text-to-code dataset, augmented with Claude-4-Sonnet annotations, I demonstrate that the reflection approach not only achieves a 4.3% improvement in code generation quality but also develops emergent error analysis capabilities that were not explicitly trained for. This suggests that learning from mistakes with structured feedback creates more versatile and robust models.

2. Dataset and Methodology

2.1 Base Dataset and Preprocessing

2.1.1 Dataset Source

I utilized the Microsoft CodeXGLUE text-to-code dataset (Java subset) as my foundation, specifically from https://github.com/microsoft/CodeXGLUE/tree/main/Text-Code/text-to-code. The dataset contains natural language descriptions paired with corresponding Java method implementations from the train.json file. This dataset presents unique challenges due to its specialized formatting with field and element separators that need careful preprocessing.

2.1.2 Data Cleaning Process

The natural language descriptions in the original dataset contain special tokens (`concode_field_sep` and `concode_elem_sep`) that represent structural information. All descriptions underwent the following cleaning transformation:

```
def clean_nl(nl_description):
    cleaned = nl_description.replace("concode_field_sep", " | ")
    cleaned = cleaned.replace("concode_elem_sep", ", ")
    return ' '.join(cleaned.split())  # Remove extra spaces
```

This preprocessing step was crucial for both training and inference phases to ensure consistent input formatting. Without this cleaning, the models would struggle with the artificial separators that don't appear in natural programming contexts.

2.2 Meta Dataset Creation Process

To create reflection-based training examples, I developed a sophisticated three-stage pipeline using the claude_analysis_api.py script:

2.2.1 Stage 1: Zero-shot Generation

I used the base LLaMA 3.2 3B model (https://huggingface.co/meta-llama/Llama-3.2-3B) without any fine-tuning to generate initial code attempts for 10,000 randomly selected examples. This step was critical because it captured the types of errors and misconceptions that an untrained model would make, providing realistic mistake patterns for the reflection training.

2.2.2 Stage 2: Claude-4-Sonnet Analysis

For each example, I sent a carefully structured triplet to the Claude API:

- The cleaned natural language description
- The base model's zero-shot attempt
- The correct implementation from the dataset

Claude-4-Sonnet was prompted with a specific structured format requiring classification (CORRECT/INCORRECT/PARTIALLY CORRECT) followed by error analysis and learning insights, each limited to 150 words. While this structured approach ensured consistent annotations, it may have introduced unintended constraints (discussed in limitations). The annotation process for 10,000 examples cost approximately $65 in API fees.

Claude provided detailed analysis in two key components:

- **Error Analysis**: Specific identification of what the model did wrong, including syntax errors, logic flaws, and missing requirements
- **Learning Insights**: Root cause analysis of why these errors occurred and concrete suggestions for improvement

2.2.3 Stage 3: Reflection Format Creation

I structured the annotations into comprehensive teacher-student dialogues following this format:

```
{
  "id": 123,
  "nl": "sets the value of the routingnumber property",
  "code": "void function(String arg0) { this.routingNumber = arg0; }",
  "label": "meta",
  "base_model": "public class BankAccount { ... }",
  "error_analysis": "The model output is INCORRECT...",
  "learning_insights": "1) Recognize setter patterns..."
}
```

2.3 Final Dataset Composition

2.3.1 Dataset Statistics

The complete dataset (available at https://huggingface.co/datasets/Naholav/llama3.2-java-codegen-90sft-10meta-claude-v1) consists of:

- **Total Examples**: 100,000
- **SFT Examples**: 90,000 (90%) - Standard input-output pairs
- **Meta Examples**: 10,000 (10%) - Examples with reflection annotations
- **Test Set**: 100 examples (balanced difficulty, unseen during training)

2.3.2 Dataset Structure

The dataset structure includes eight fields:

- id (unique identifier)
- nl (natural language description)
- code (ground truth Java implementation)
- label ("sft" or "meta")
- base_model (zero-shot attempt - meta examples only)
- error_analysis (Claude's evaluation - meta examples only)
- learning_insights (improvement suggestions - meta examples only)
- raw_claude_response (complete API response - meta examples only)

2.4 Experimental Setup and Infrastructure

2.4.1 Model Architecture

LLaMA 3.2 3B (3 billion parameters) was chosen as the base model for its balance between capability and computational efficiency. The model uses RoPE positional embeddings and supports a context length of 2048 tokens.

2.4.2 Training Configuration

Both training approaches utilized identical infrastructure and hyperparameters to ensure fair comparison:

```
{
    "learning_rate": 2e-5,
    "batch_size": 8,
    "gradient_accumulation_steps": 6,  # Effective batch size: 48
    "num_epochs": 3,
    "max_length": 2048,
    "precision": "float32",  # For NaN stability
    "optimizer": "AdamW",
    "scheduler": "cosine_with_warmup",
    "warmup_ratio": 0.03,
    "weight_decay": 0.01,
    "max_grad_norm": 1.0,
    "early_stopping_patience": 3,
    "eval_frequency": 5  # evaluations per epoch
}
```

Both models employed early stopping with a patience of 3 evaluations to prevent overfitting and ensure optimal performance. The models were evaluated 5 times per epoch on the validation set, and training was stopped if validation loss did not improve for 3 consecutive evaluations. The best checkpoints were automatically saved based on validation performance.

2.4.3 Hardware Specifications

- GPU: NVIDIA A100 80GB SXM
- Peak Memory Usage: ~70-75GB VRAM
- Framework: PyTorch 2.0+ with Transformers library
- Additional optimizations: Flash Attention, gradient checkpointing

2.5 Evaluation Methodology

For evaluation, I created a carefully curated test set of 100 diverse examples (codexglue_test_100_samples.json) representing various difficulty levels and programming patterns.

2.5.1 Prompt Standardization

Both models received identical prompts following the format:

```
prompt = f"""You are an expert Java programmer. Generate a complete,
working Java method for the given description.

Task: {task}

Requirements:
- Write a complete Java method
- Use proper syntax and naming conventions
- Include return statements where needed
- Keep it concise but functional

```java
"""
```

2.5.2 Generation Parameters

To ensure deterministic and comparable outputs:

```
generation_config = {
    "max_new_tokens": 1024,
    "temperature": 0.0,  # Deterministic generation
    "do_sample": False,
    "top_p": 0.9,
    "top_k": 50,
    "repetition_penalty": 1.1,
    "num_beams": 5  # Beam search for quality
}
```

2.5.3 Evaluation Design Choice

To ensure fair comparison, both models received identical standard SFT prompts during evaluation, despite their different training paradigms. The reflection model was trained on two types of prompts: 90% standard SFT format and 10% teacher-student reflection format. However, during inference, only the standard prompt was used for both models. This design choice was made to isolate the impact of the training methodology rather than prompt engineering, but it potentially underestimates the reflection model's capabilities since it was not tested with reflection-aware prompts that could leverage its unique training.

2.5.4 Claude-4-Sonnet Scoring
```

Each generated output was sent to Claude for scoring on a 0-100 scale based on:

- Correctness of implementation
- Code quality and adherence to Java conventions
- Proper handling of edge cases
- Syntactic completeness and validity

3. Training Approaches and Results

3.1 Standard Supervised Fine-Tuning Implementation

3.1.1 Training Methodology

The SFT approach (implemented in train_sft.py) followed the traditional paradigm of next-token prediction with cross-entropy loss. The training process treated each example independently, with the model learning to map natural language descriptions directly to code implementations.

3.1.2 Training Process Details

- Dataset: 100,000 total examples with 10% held for validation (90k training, 10k validation)
- Each training example was seen exactly once per epoch
- Standard teacher forcing was employed during training
- Single loss tracking for all examples
- The model optimized purely for minimizing the difference between predicted and ground truth tokens
- Early stopping with patience of 3 evaluations was used
- No distinction was made between different types of errors or patterns

3.1.3 Training Dynamics

- The model converged steadily through epoch 2
- Best checkpoint saved at end of epoch 2 (step 3750)
- Epoch 3 showed signs of overfitting, validating the early stopping decision
- Final validation loss at best checkpoint: 0.6012 (sample average)
- Training duration: approximately 8 hours

The trained model available at https://huggingface.co/Naholav/llama-3.2-3b-100k-codeXGLUE-sft represents the epoch 2 checkpoint, before overfitting occurred. Training logs demonstrating this convergence pattern are available at https://github.com/naholav/sft-vs-reflection-llama3-codexglue.

3.2 Reflection-Based Meta-Learning Implementation

3.2.1 Core Innovation

My novel approach (implemented in train_meta.py) incorporated structured error reflection through teacher-student dialogues, fundamentally changing how the model learns from examples.

3.2.2 Reflection Training Format

```
<student_teacher_reflection>
LEARNING SCENARIO: {task description}
STUDENT: {incorrect attempt by base model}
TEACHER: {correct implementation}
FEEDBACK: {detailed analysis of what went wrong}
GUIDANCE: {explanation of why it happened and how to improve}
REFLECTION: {key patterns and lessons to internalize}
</student_teacher_reflection>
```

3.2.3 Dual Loss Tracking Innovation

The key technical innovation was implementing separate loss tracking for different example types. During both training and validation, the model computed separate losses for SFT and meta examples based on their labels:

- SFT validation loss: 0.5770 (lower than pure SFT's 0.6012)
- Meta validation loss: 0.3945 (appropriately low without excessive gap)
- Meta-SFT gap: -0.1825 (demonstrates effective knowledge transfer without overspecialization)

This dual tracking allowed precise monitoring of how the model learned from each data type, ensuring balanced performance across both standard code generation and reflection-based examples.

3.2.4 Training Characteristics

- Dataset: 100,000 total examples with balanced 10% validation split
  - Training: 81,000 SFT + 9,000 meta examples
  - Validation: 9,000 SFT + 1,000 meta examples
- Dual loss tracking enabled separate monitoring for each data type
- The model learned to recognize error patterns and their corrections
- Meta examples were weighted equally despite being only 10% of data
- Early stopping with patience of 3 evaluations prevented overfitting
- Best checkpoint saved at end of epoch 2 (step 3750)
- Epoch 3 exhibited overfitting signs in validation metrics
- Training duration: approximately 9 hours

The trained model available at https://huggingface.co/Naholav/llama-3.2-3b-100k-codeXGLUE-reflection represents the epoch 2 checkpoint. Both models' training logs at

https://github.com/naholav/sft-vs-reflection-llama3-codexglue clearly show the overfitting patterns in epoch 3, confirming that epoch 2 was the optimal stopping point.

3.3 Quantitative Results Analysis

3.3.1 Performance Metrics

The evaluation on 100 test examples revealed measurable improvements. While a 4.3% improvement might seem modest, it represents consistent better performance across diverse coding tasks. The high standard deviation in both models (approximately 31) indicates significant variation in task difficulty within the test set, with both models struggling on similar complex examples.

Table 1: Model Performance Comparison

Model	Average Score	Standard Deviation	Validation Loss
SFT Model	60.66 / 100	30.98	0.6012
Reflection Model	63.27 / 100	30.76	0.5770 (SFT) / 0.3945 (Meta)

Note: Both models were saved at the end of epoch 2 when validation performance peaked. Training logs show clear overfitting patterns in epoch 3, validating the early stopping decision. All reported metrics are from these epoch 2 checkpoints, which are the models available on Hugging Face.

3.3.2 Evaluation Context

Important Context on Evaluation Fairness: This 4.3% improvement is likely a conservative estimate of the reflection model's true capability. During training, the reflection model learned from both standard prompts (90%) and reflection-formatted prompts (10%), developing specialized understanding of error patterns and corrections. However, the evaluation used only standard SFT prompts for both models to ensure fairness. This means the reflection model was essentially tested in a "handicapped" state, not utilizing its full potential that could be unlocked with reflection-aware prompts.

3.4 Qualitative Findings: Emergent Capabilities

3.4.1 Debugging Capabilities Discovery

Beyond the quantitative metrics, I discovered dramatic differences in capabilities when testing the models on debugging tasks—a capability not explicitly trained for:

3.4.2 Reflection Model Capabilities

- Successfully identifies critical errors in existing code with high accuracy
- Provides accurate root cause analysis explaining why code fails

- Suggests viable fixes that address the actual problems
- Demonstrates genuine understanding of code logic and program flow
- Shows ability to reason about code correctness beyond pattern matching

3.4.3 SFT Model Limitations

- Very poor at detecting actual errors in code
- Frequently hallucinates problems that don't exist
- Provides generic or irrelevant fix suggestions
- Shows only superficial pattern matching without semantic understanding
- Unable to reason about why code might fail

3.4.4 Interactive Testing Insights

During interactive chat sessions, the performance gap was even more pronounced. When presented with buggy code to analyze, the reflection model demonstrated orders of magnitude better performance. While the SFT model frequently hallucinated non-existent issues and provided incorrect analyses, the reflection model accurately identified actual bugs and provided meaningful root cause analysis. This dramatic difference in analytical capability suggests that the reflection training created fundamentally different internal representations of code understanding.

3.4.5 Limitation in Code Regeneration

However, an interesting limitation emerged during interactive testing. When the reflection model was asked to analyze its own previously generated code and produce an improved version, it often either reproduced nearly identical code or made only minor modifications such as adding null checks. Despite its superior analytical capabilities in identifying problems, the model struggled to translate these insights into substantially improved code generation when using standard SFT prompts. This suggests that while the model developed strong analytical skills, the ability to leverage these insights for iterative code improvement may require specialized prompting strategies or additional training objectives.

3.4.6 Pipeline vs. Single-Step Generation

This limitation appears specifically in multi-step pipeline scenarios (generate → analyze → regenerate). The model excels at steps 1 and 2 but struggles with step 3, suggesting that iterative refinement as a pipeline may not align well with the model's training. However, this limitation could potentially be addressed through prompt engineering that activates the model's meta-learning in a single generation step. Rather than treating generation and reflection as separate phases, prompts that integrate reflection-awareness from the beginning (e.g., "Generate a Java method while considering common pitfalls such as...") might allow the model to leverage its meta-training more effectively in one cohesive response.

This emergent debugging capability in the reflection model suggests that training on error analysis creates deeper semantic understanding that transfers to related tasks. The model learned not just to generate code but to understand what makes code correct or incorrect.

3.5 Implementation Repository

All code, training scripts, logs, and evaluation results are available at https://github.com/naholav/sft-vs-reflection-llama3-codexglue. The repository structure includes:

```
├── create meta dataset and test dataset/
│   ├── claude_analysis_api.py      # Meta dataset creation
│   └── codexglue_test_100_samples.json
├── training_scripts/
│   ├── train_sft.py                # SFT implementation
│   └── train_meta.py               # Reflection training
├── evaluation/
│   └── meta_vs_sft_learning_claude_result.csv
└── logs/
    ├── 100k_sft_training.log
    └── 100k_meta_training.log
```

4. Conclusion

This study demonstrates that reflection-based meta-learning can meaningfully improve code generation performance compared to standard supervised fine-tuning. The 4.3% improvement in generation quality, while modest in absolute terms, comes with the significant added benefit of emergent error analysis capabilities that were not explicitly trained for. This validates my core hypothesis that learning from mistakes with structured feedback enhances both task performance and model understanding.

4.1 Key Contributions

4.1.1 Novel Training Methodology

I introduced a reflection-based meta-learning approach that incorporates structured error analysis through teacher-student dialogues. This paradigm shift from pure imitation learning to error-aware training represents a new direction for code generation models.

4.1.2 Empirical Validation

Through rigorous evaluation on the CodeXGLUE Java dataset, I provided empirical evidence that error-based learning improves code generation quality. The consistent improvement across diverse tasks suggests the approach's generalizability.

4.1.3 Emergent Capabilities

I discovered that training on error analysis examples leads to emergent debugging abilities, despite the model never being explicitly trained for error detection tasks. This suggests that reflection training creates more versatile models.

4.1.4 Resource Contribution

I released a high-quality dataset containing 10,000 teacher-student reflection examples annotated by Claude-4-Sonnet, along with trained models and complete implementation code, enabling reproducibility and further research.

4.2 Implications for Future LLM Training

The success of this approach has several important implications:

4.2.1 Deep Understanding vs. Surface Patterns

As language models become more powerful, training methodologies that develop genuine understanding rather than sophisticated memorization will become increasingly crucial. The reflection model's debugging capabilities demonstrate that it learned underlying principles rather than just statistical patterns. However, my observation that the model struggles to translate analytical insights into improved code generation reveals that understanding and generation remain partially decoupled skills that may require targeted training approaches.

4.2.2 Human-Inspired Learning

This work suggests that mimicking human learning processes—particularly the critical role of learning from mistakes with explicit feedback—can be an effective strategy for training models on complex tasks requiring both syntactic correctness and semantic understanding.

4.2.3 Efficiency of Meta Examples

Despite comprising only 10% of the training data, the meta examples had a disproportionate positive impact, suggesting that quality and diversity of training data matter more than pure quantity. The $65 investment in API costs for creating 10,000 high-quality reflection examples proved highly cost-effective given the performance improvements and emergent capabilities.

4.2.4 Optimal Meta-SFT Ratio

Through empirical testing, I found that the 10% meta ratio was crucial for success. Experiments with higher meta percentages revealed significant challenges: 100% meta data led to severe overfitting, while a 50-50 split caused both training and validation losses for meta examples to drop too rapidly compared to SFT examples, creating an undesirable gap. This loss imbalance across both training and validation indicates the model becoming too specialized on meta examples at the expense of general code generation ability. Based on these observations, I recommend keeping meta examples between 10-12.5% of the total dataset to maintain balanced learning across both data types.

4.2.5 Analysis-Generation Gap

My findings reveal an interesting dichotomy: the reflection model developed strong analytical capabilities but showed limitations in leveraging these insights for iterative code improvement. This suggests that error analysis and code generation, while related, may require different cognitive pathways that need to be explicitly connected through training or prompting strategies.

4.2.6 Pipeline Limitations and Opportunities

The observed difficulty in multi-step pipelines (particularly the regeneration phase) contrasts with the model's strong performance in single-step tasks. This suggests that the reflection training may be most effective when activated holistically rather than sequentially. The meta-knowledge acquired during training appears to be better suited for integrated, single-step generation where reflection happens implicitly during the initial code creation, rather than as a separate post-hoc improvement phase. This insight opens new avenues for prompt engineering that could unlock the model's full potential without requiring complex multi-turn interactions.

4.3 Limitations and Future Directions

4.3.1 Current Limitations

- The evaluation used standard prompts for both models, potentially underestimating the reflection model's capabilities with specialized prompts
- The approach was tested only on Java method generation, not full programs or other languages
- The 100-example test set, while diverse, may not capture all edge cases
- The teacher model (Claude) was prompted with a rigid format that may have limited response diversity

4.3.2 Teacher Response Pattern Limitation

During meta dataset creation, Claude was prompted with a highly structured format requiring specific word limits and classification patterns. This rigid prompting likely resulted in lower variance in teacher responses, potentially causing the student model to overfit to Claude's specific writing patterns rather than learning generalizable error analysis skills. The consistent structure in teacher feedback may have

inadvertently taught the model to mimic Claude's response style rather than developing true analytical capabilities.

4.3.3 Future Research Directions

4.3.3.1 Single-Step Reflection-Aware Prompting

The most immediate opportunity lies in developing prompts that activate the model's meta-learning capabilities in a single generation step rather than as a multi-step pipeline. Since the model struggles with the third step of a generate-analyze-regenerate pipeline but has internalized reflection knowledge, prompts that trigger this knowledge upfront could be more effective. For example, prompts like "Generate a Java method for X while avoiding common errors such as null pointer exceptions and edge cases" might allow the model to apply its meta-training proactively rather than reactively. This approach could bypass the observed limitation in iterative improvement while still leveraging the model's enhanced understanding.

4.3.3.2 Iterative Code Improvement Training

My observation that the reflection model excels at analysis but struggles with iterative improvement suggests a need for training objectives that explicitly teach models to generate improved versions based on identified errors. This could involve multi-turn training where models learn to analyze, critique, and then regenerate better solutions.

4.3.3.3 Scaling Studies

Test the approach on larger models (7B, 13B, 70B parameters) to understand how reflection-based training scales with model capacity.

4.3.3.4 Multi-Language Extension

Extend the methodology to other programming languages, particularly dynamically typed languages where error patterns differ significantly.

4.3.3.5 Curriculum Learning

Develop strategies for progressively introducing more complex errors and patterns, similar to how human programmers advance from simple to complex debugging scenarios.

4.3.3.6 Interactive Learning

Explore whether models can generate their own reflection examples through self-play, reducing dependence on external teacher models.

4.3.3.7 Teacher Prompt Diversification

Future iterations should employ more flexible teacher prompting strategies with higher temperature settings to increase response variance. Instead of rigid word limits and required classifications, allowing Claude to provide more natural, varied feedback could prevent student models from overfitting to specific response patterns. This would likely improve the model's ability to generalize error analysis skills beyond mimicking the teacher's style.

4.4 Final Remarks

This work represents a step toward more sophisticated training paradigms for code generation models. By demonstrating that models can learn not just from correct examples but from understanding why incorrect attempts fail, I open new avenues for creating more robust, interpretable, and capable code generation systems. The availability of my models, datasets, and implementation details at the provided links enables the community to build upon this foundation and explore the full potential of reflection-based learning for code intelligence tasks.

References

[1] Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S. K., Fu, S., & Liu, S. (2021). CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks* (NeurIPS 2021). https://github.com/microsoft/CodeXGLUE

[2] Meta AI. (2024). LLaMA 3.2: Open Foundation and Fine-Tuned Chat Models. Meta AI Research. https://huggingface.co/meta-llama/Llama-3.2-3B

[3] Anthropic. (2024). Claude 4 Sonnet: Constitutional AI Assistant. Anthropic. https://www.anthropic.com

[4] Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., & Rush, A. M. (2020). Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (pp. 38-45).

[5] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., & Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32* (pp. 8024-8035).

[6] Dao, T., Fu, D., Ermon, S., Rudra, A., & Ré, C. (2022). FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems 35* (pp. 16344-

16359).

[7] Ho, N., Schmid, L., & Yun, S.-Y. (2022). Large Language Models Are Reasoning Teachers. *arXiv preprint arXiv:2212.10071*. https://arxiv.org/pdf/2212.10071

[8] Loshchilov, I., & Hutter, F. (2019). Decoupled Weight Decay Regularization. In *International Conference on Learning Representations* (ICLR 2019).

[9] Zhang, J., Zhao, Y., Saleh, M., & Liu, P. J. (2020). PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization. In *Proceedings of the 37th International Conference on Machine Learning* (ICML 2020).

[10] Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv preprint arXiv:1412.3555*.