FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Teodora Stojcheska

Fine-tuning Code Generation Models with Compiler Feedback

Faculty of Mathematics and Physics

Supervisor of the bachelor thesis:	Martin Pilát
Study programme:	Computer Science
Study branch:	Artificial Intelligence

Prague 2024

First and foremost, I would like to thank my supervisor Martin Pilát for his support and patience throughout the work of this thesis. He is always positive and willing to help solve all issues at hand, which helped tremendously.

Next I would like to thank Alexander Bezzubov for always giving me a push when needed and keeping me accountable. And for the invaluable guidance that he has offered.

Lastly I would like to thank JetBrains for the support, as this work was done in collaboration with them. Doing this thesis without the resources they offered like access to Cadence plugin would have been much more difficult.

Title: Fine-tuning Code Generation Models with Compiler Feedback

Author: Teodora Stojcheska

Department: Faculty of Mathematics and Physics

Supervisor: Martin Pilát, Department of Theoretical Computer Science and Mathematical Logic

Abstract: Large Language Models (LLMs) that are pre-trained on code have become the leading method for program synthesis. These models however mainly rely on predicting the next token, which may fail to capture the syntax and meaning of code. Coarse-Tuning Models of Code with Reinforcement Learning Feedback [1] paper hypothesized that teaching the model to provide compilable code by providing compiler feedback will improve performance on downstream tasks. This project expands on that idea with the main difference that compiler feedback is provided to the model in an RL free way, by using methods such as such as Kahneman Tversky Optimization (KTO) [2] and Direct Preference Optimization (DPO) [3]. Both methods use feedback to ground the LLM, albeit in slightly different contexts. This thesis seeks to assess the effectiveness of these approaches and their ability to generate compilable code ultimately contributing to advancements, in program synthesis.

Keywords: LLM compiler DPO KTO

Contents

Introduction

In recent years there have been major advancements in the machine learning field, especially in Natural Language Processing (NLP) with the rise of transformer models. Although originally meant for machine translation, the impact transformers have made is much bigger than just NLP, and spans over other domains including software engineering.

Code Generation Code generation is the task of creating code from natural language using ML models. This is important in software engineering since it accelerates the development process and allows for quick prototyping. Moreover it allows developers to focus on more complex and creative aspects of software by automating repetitive and boilerplate tasks.

Problem statement Code generation, as a subdomain of NLP, has had a lot of fruitful research revolving around this field. However, there are still many weak points that models solving this task face.

State-of-the-art Large Language Models (LLMs) trained on code, especially smaller scale ones, can sometimes make basic mistakes that human programmers would likely not make and even compiler can detect. For example, it is common for LLM-generated code to use uninitialized variables, produce non-terminating loops or use incorrect types.

This seemingly easy problem is relevant because even if the underlying solution proposed by the model is logically sound, one is unable to evaluate the generation properly. In that case using execution based metrics is not an option, so we must rely solely on similarity metrics. Moreover fixing compilation errors doesn't provide the best user experience, which is the end goal.

So the issue of producing code with compiler errors undermines the potential and utility models hold in practical coding environments.

Current solutions Currently, this problem is addressed by scaling up - either enlarging the model size or increasing the volume of training data. This approach has proven to be very effective and has shown great results in many downstream tasks and as a side result it helps the issue of producing code that doesn't compile.

However, scaling is expensive as it requires a lot of resources for obtaining and cleaning data, as well as hardware resources for training the models. Current research is skewed towards larger models and it sidelines the potential smaller model architectures hold.

In light of that, this thesis proposes the use of a 1-billion parameter model for all experiments, in order to illustrate the challenges that smaller models face in one narrower context - compilation of outputs; and explore a possible solution besides obtaining code that compiles more often.

The central hypothesis of this project revolves around the effect of incorporating compiler feedback in the training process. Specifically, testing whether teaching models to produce compilable code will enhance model performance on downstream tasks.

Employed Solution The approach explored throughout this work is to utilize feedback during the training process. The usual way that feedback is looped back to the model is by using some Reinforcement Learning approaches. Although reinforcement learning has done a great job at aligning models with (usually human) feedback, which has as a result improved performance on downstream tasks, it's less stable and very sensitive to hyperparameter changes.

So, this thesis experiments with recently published reinforcement learning free methods for model alignment with feedback. Using Reinforcement Learning (RL) free methods is promising not only performance-wise, but also in terms of stability and scalability of the training process.

Usually all of those approaches that incorporate feedback rely on human annotated data that is fed to the model. However, this work uses compiler results as feedback, which is very cost-effective and directly addresses the core issue that is being solved, which is generating compilable code.

To summarize, this thesis tries to improve the performance of models on downstream code tasks, by combining state of the art research that is less resource intensive, specifically using compiler feedback and RL free approaches for feedback incorporation. This research aims to increase the impact task specific fine-tuning has by teaching model to generate compilable code.

Thesis structure The **opening chapter** lays out the background and motivation. It provides information about the current state of the art approach to program synthesis, details of the architecture and training process. The **second chapter** dives deeper into previous research on RL free methods for feedback incorporation and utilizing compiler feedback. The **third chapter** presents the practical aspects of the solution. It provides implementation and methodology details. The **fourth chapter** is dedicated to showcasing the research outcomes. It outlines the experiments layout (data, prompts and hyperparameters) and analyzes the findings. The **concluding chapter** encapsulates the project's significance and draws together the key findings and their implications, efficacy and impact. It also points out the contributions made by the thesis as well as potential directions for future research.

Chapter 1

Background

Over the past years code generation has evolved a lot, as subset of text generation. Early efforts focused on rule-based methods that defined explicit rules for code construction which soon got substituted by ML. Recently, deep learning and transformer architectures revolutionized the field, and now based on that advancement many sophisticated and capable code generation models are being trained. This chapter provides an overview of the current most common technique of training models for code generation,. It provides detail of model structure, specifically transformer based models and training process.

1.1 Neural Networks Background

Neural networks are the building block for deep learning models. A neural network is a computational model inspired by the human brain's structure and functioning. [4]

Perceptron is a basic ML computational unit which mimics simple decision making process. It takes as input a vector, applies weights and activation function in order to produce an output.

$$\hat{y} = \sum_{i=1}^{n} x_i w_i + b_i \tag{1.1}$$

Neural Network is structured as a collection of perceptions organised in layers. These layers work together to process complex patterns in the data.

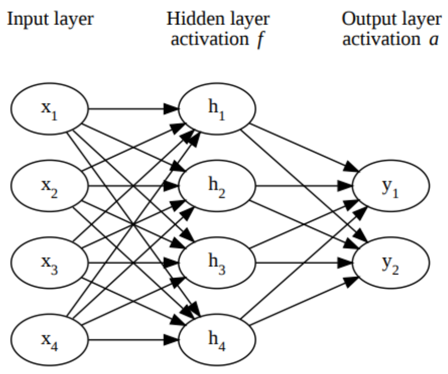

Figure 1.1 Neural Network structure. [5]

1.1.1 Neural Networks structure

There are three most important parts of the neural network, the input output and hidden layers as depicted in Figure 1.1. The input layer receives the raw data and passes it to the following layers. The hidden layers whose goal is to detect features or patterns, transform the data through intermediate computations. Finally, the output layer produces the final prediction. Neural networks learn to adjust the weights throughout the training process in order to produce expected outputs.

$$h_i = f\left(\sum_j x_j w_{j,i}^{(h)} + b_i^{(h)}\right) \tag{1.2}$$

$$y_i = a\left(\sum_j h_j w_{j,i}^{(y)} + b_i^{(y)}\right) \tag{1.3}$$

1.1.2 Neural Networks training process

Training the neural network is the process of optimizing the neural network's weights by minimizing the loss. The purpose of the loss function is to measure the error (e.g., Mean Squared Error, Cross-Entropy) and choosing the right loss depends on the task.

Training of a neural network is executed in three stages: initialization, feed-forward propagation and backpropagation.

- Initialization: Random values are assigned to the weights and biases.
- Feedforward propagation: The input is passed though the network in order to get the output without modifying the weights.

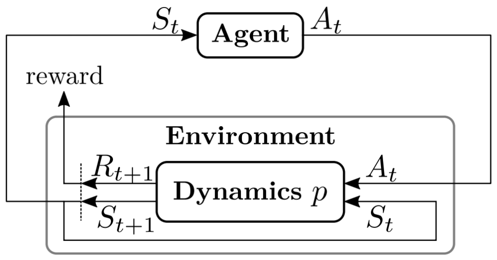

Figure 1.2 Reinforcement Learning environment dynamics. [7]

- Backpropagation: In this step the weights are adjusted based on the loss function, which is a measure of difference between the model's predicted outputs and the ground truth. The most common method for doing this is gradient descent which calculates the gradient of the loss with respect to the weights and subtracts it from the weights to minimize the loss.

1.2 Reinforcement Learning

Reinforcement Learning (RL) [6] is a framework for training models for solving certain tasks. There is an agent in an environment that tries to solve the task at hand. At each time-step the agent chooses an action and receives a rewards from the environment and natually wishes to maximize the reward.

Markov decision process (MDP) is a discrete-time stochastic control process that has the Markov property. The Markov property states that the next state depends on the previous state. It is discrete because there are concrete states that the agent is in where all properties are well defined. Since the agent decides on the next action based on a probability distribution, the whole process is stochastic and not deterministic.

An MDP is a quadruple $(\mathcal{S}, \mathcal{A}, p, \gamma)$

- $\mathcal{S}$ is a set of states
- $\mathcal{A}$ is a set of actions
- $p(S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a)$ is a probability that action $a \in \mathcal{A}$ will lead from state $s \in \mathcal{S}$ to $s' \in \mathcal{S}$, producing a reward $r \in \mathbb{R}$
- $\gamma \in [0, 1]$ is a **discount factor**

A policy π represents distribution of actions in a given state. It's equivalent to the probability of performing an action a in state s.

To evaluate a policy a state value function is used that is defined by 1.4.

$$v_\pi(s) \overset{\text{def}}{=} \mathbb{E}_\pi\left[G_t \mid S_t = s\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right] \tag{1.4}$$

A key concept in RL is exploration vs exploitation, which is a balance of exploring the environment to discover new knowledge and exploiting known information to maximize rewards.

1.3 Code Generation Models

Simple neural network architectures evolved into complex model architectures solving huge variety of tasks starting from simple classification problems to text generation. The most prominent model in text generation today is the transformer model which state of the art models like ChatGPT are based on. As code generation is in reality a subset of text generation same high level state of the art principals that apply for the latter apply for the former as well.

1.3.1 Structure of Transformer Models

Transformer-based models [8] consist of an encoder-decoder structure for sequence generation, depicted in Figure 1.2. Each Transformer block comprises multi-head self-attention mechanisms and position-wise feed-forward neural networks.

The architecture typically includes:

1. Input Embeddings: Neural networks cannot directly process textual input, so the input needs to be encoded. Hence, both natural language descriptions and code tokens are converted into dense vectors - embeddings. However, embeddings do not incorporate information about word position, which is why they are always coupled with positional encodings that retain the order of tokens.

2. Encoder: consists of stacked attention blocks, which calculate the attention scores for weighing the importance of elements in the input sequence. Each attention block contains multi-head self-attention and feed-forward layers. For code generation, the encoder reads the input (that consists of natural language and code) and constructs a contextual representation of the input. A key component of the encoder model is the self attention layer which weighs the importance of different tokens in the input sequence. It allows the model to focus to different parts of the input sequence simultaneously, enabling it to capture relationships between code tokens over long distances.

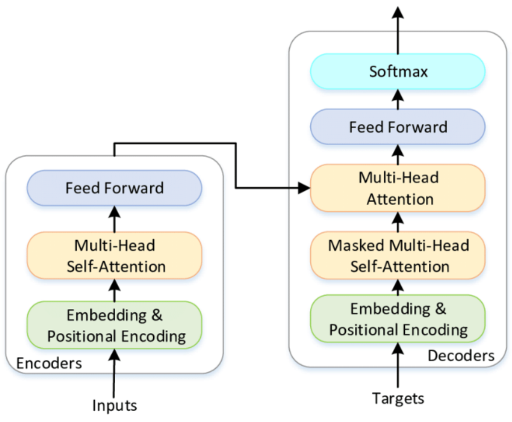
Softmax
Feed Forward
Multi-Head Attention
Masked Multi-Head Self-Attention
Embedding & Positional Encoding
Decoders
Targets
Feed Forward
Multi-Head Self-Attention
Embedding & Positional Encoding
Encoders
Inputs

Figure 1.3 Figure of transformer architecture. [9]

3. Decoder is similarly structured as the encoder. It generates the target sequence of code tokens, using the context provided by the encoder. The decoder employs masked self-attention, which ignores certain positions in the input sequence during the computation of attention scores. The goal of this is to ensure that it generates the code sequentially and it doesn't look into the future.

1.3.2 Code generation model training process

Code generation models are usually based on the transformer architecture. So training a code generation model starts with pre-training a transformer based model and then specializing it for the task of code completion.

The steps that are usually taken when training a model on code completion are the following:

- Pre-training: transformer based model is fed large-scale data in order to get a broad and deep understanding of code and text.
- Fine-tuning: continue training on a smaller dataset focused on one specific task with the goal of refining the model and enhancing its capabilities in a narrower context.
- Reinforcement Learning from Human Feedback (RLHF): align the model with human feedback with the goal of optimizing the model to produce more relevant and high-quality code. This ensures that the model performs well not only on similarity or execution based metrics but also meets expectations of end-users.

Pre-training the Model

Pre training a model consists of initializing a model and training it on a large and diverse dataset. The goal of this step is for the model to learn general data features and patterns. Models intended for code tasks are trained on a big corpus of natural language and code. The code part of the corpus should be collected from many repositories and cover various programming languages, libraries, and frameworks. The goal is enabling the model to develop a foundational understanding of syntax, structure and standard library usage for many programming languages.

Common tasks a model is trained on during pre-training are Masked Language Modeling (MLM) - model should predict randomly masked tokens from the input sequence; Next Sentence Prediction (NSP) predicting whether a given sentence B is the actual sentence that follows sentence A.

Supervised Fine-tuning (SFT)

Fine tuning step takes a pre trained model and trains it further on a smaller, high quality task-specific dataset. The goal narrows down to predicting the most probable token for the downstream task.

Fine-tuning is typically done in a supervised manner where the model is optimized over a dataset representative of the deployment scenario. For example for code completion it's similar to language modeling (next token prediction) where the model predicts the next sequence token, but in the domain of code.

Reinforcement Learning with Human Feedback (RLHF)

RLHF is a fine tuning phase that uses human annotators that provide direct feedback of the outputs the model has generated. This step has shown great success in aligning the model to human preference by addressing issues like code quality, readability and context relevance, which are harder to capture by automated metrics alone. This step consists of three intermediate steps:

- Collect Human Feedback: Human experts on the task evaluate outputs generated by the fine-tuned model based on some predetermined criteria.
- Reward Modeling: The collected human feedback is used to train a reward model that should quantify the quality and utility of the outputs the model produces in reference to the user query. This reward function $R(x, a)$ assigns higher scores to outputs preferred by humans.
- Policy Optimization: Then the model is trained using a reinforcement learning algorithm like Proximal Policy Optimization (PPO) [10]. The objective now is to maximize the expected reward.

Chapter 2

Related Research

This chapter introduces the previous research which this thesis is built on. The inspiration behind this thesis mostly lies in the work of Reinforcement Learning from Compiler Feedback (RLCF) [1] paper described below. There is one key difference between this work and their work which is that we utilize RL free methods for fine tuning the model like KTO and DPO, which are explained in more detail in this chapter.

2.1 Reinforcement Learning from Compiler Feedback

Using compiler feedback for improving model performance on downstream coding tasks is an idea that was introduced by RLCF paper [1]. They take inspiration from RLHF procedure that aligns model to human preferences and frame the problem as: model's output not aligned with the user's desired outcome of compilable code.

In order to alleviate that misalignment, they introduce an additional step, called coarse tuning whose goal is to align the model's output with compilable code more often.

The way they approach it is by formulating the problem of writing code as a Markov Decision Process (MDP). An MDP is a mathematical framework used to describe an environment in which an agent operates. This environment is characterized by states, rewards and transition probabilities. In this definition, the pre-trained model acts as a policy and we train a reward mode consisting of two parts: a compiler and a discriminator.

2.1.1 Reinforcement Learning from Compiler Feedback model

The problem we are trying to minimize is:

$$\mathbb{E}_{(x,y)\sim\mathcal{F}}\left[L_{\text{anc}}(\theta'|x,y,\theta)+\alpha\mathbb{E}_{y'\sim f_{\theta'}(\cdot|x)}\left[D(x,y,y')\right]\right] \tag{2.1}$$

Where, D is the reward, L_{anc} is anchor loss, which should ensure the parameters don't change too rapidly, so one option for this function is divergence between each token. F is the distribution from which x, the input (textual description of program or partially finished program) and y, the reference are sampled from: $(x,y)\sim F$ and y' is the generated solution that comes from the model.

The loss function is the second part of the above mentioned expectation, which we try to minimize.

Reward (D) The reward consists of two parts: compiler and discriminator. The first part of the reward is the compiler/static analysis, which simply compiles/analyzes both y and y' and returns a binary output 1 when it compiles and -1 when it doesn't compile. But also it provides some sort of localization by returning the tokens up until the error and a reward of -1.

The discriminator is called only if both y and y' compile, otherwise the reward is -1. The goal of the discriminator is to try and predict whether the code given to it was generated by a model. The code is fed into the discriminator as embeddings obtained from CodeBERT [11] model. The discriminator is represented by:

$$g_\omega(x,y_0,y_1)\equiv\tanh\left(\text{MLP}\left(\text{CodeBERT}(x\circ y_0)\right)-\text{MLP}\left(\text{CodeBERT}(x\circ y_1)\right)\right) \tag{2.2}$$

Additionally, the reward of each token is modified by the anchor loss.

$$j\in\{1,\ldots,|\mathbf{y}'|\},\ r_j\leftarrow-\beta\log\left(\frac{f_{\theta'^{(i)}}(y'_j|\mathbf{x},\mathbf{y}'_{<j})}{f_\theta(y'_j|\mathbf{x},\mathbf{y}'_{<j})}\right) \tag{2.3}$$

Training The reward model and the policy are plugged in Proximal Policy Optimization (PPO) [10], which is the algorithm used to improve the policy.

PPO requires a critic as well, which is initialized as the pre-trained LLM with the head replaced for an MLP head used for value estimation. The loss used for training the critic is Mean Square Error (MSE). For the loss of the policy and the critic we also need the advantages and estimated returns. To obtain these, they use GAE (Generalized Advantage Estimate).

2.2 Human aware loss functions

Human aware loss functions (HALO) is a family of loss functions, that incorporate human biases.

Let π_θ be the model being trained. The model maps input $x \in \mathcal{X}$ to a probability distribution over possible outputs $y \in Y$

$$\pi_\theta : \mathcal{X} \to \mathcal{P}(\mathcal{Y}) \tag{2.4}$$

π_{ref} is the reference model; r_θ the implied reward;

In order for a loss ($f : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$) to qualify as a HALO it needs to satisfy the following:

1. Reference point distribution

$$Q(X^{'}, Y^{'}|x, y) \text{ over } \mathcal{X} \times \mathcal{Y} \tag{2.5}$$

2. Value function $v_f : \mathbb{R} \to \mathbb{R}$ that is non-decreasing everywhere and concave in $(0, \infty)$ such that f is linear in:

$$v_f(r_\theta(x, y) - \mathbb{E}_Q[r_\theta(x^{'}, y^{'}))]]) \tag{2.6}$$

2.3 Kahneman-Tversky Optimization

Kahneman-Tversky Optimization (KTO) [2] is a HALO that directly maximizes the utility of generations instead of maximizing the log-likelihood of preferences, as other methods. KTO does not need preferences, only a binary signal of whether on output is desirable or undesirable for a given input. Good results of offline PPO with dummy +1/-1 rewards suggests that—with the right inductive biases—a binary signal of good/bad generations may be sufficient to reach DPO-level performance.

The inspiration for KTO comes from Tversky and Kahneman's (1992) model [12], specifically the Prospect Theory, that looks into why humans make decisions that don't maximize expected output and looks into human biases. For example it is known that humans are loss averse, meaning that they are more sensitive to losses than gains.

The loss function for KTO is:

$$L_{KTO}(\pi_\theta, \pi_{ref}) = \mathbb{E}_{x,y\sim D}[\lambda_y - v(x, y)] \tag{2.7}$$

where

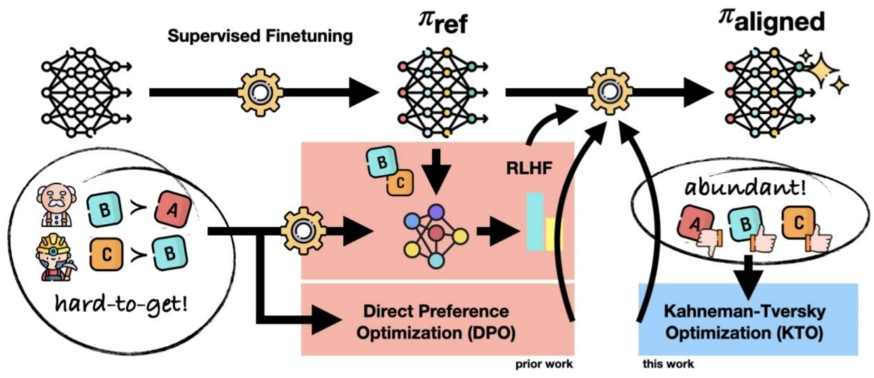

Figure 2.1 KTO method figure. [2]

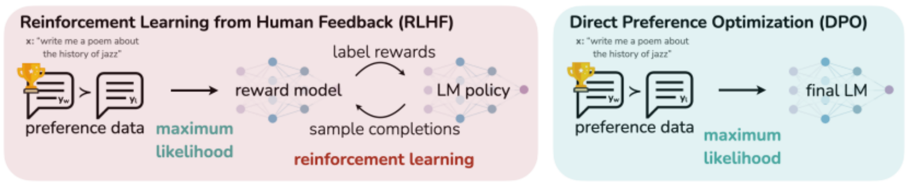
Reinforcement Learning from Human Feedback (RLHF)
x: "write me a poem about the history of jazz"
preference data
maximum likelihood
label rewards
reward model
LM policy
sample completions
reinforcement learning
Direct Preference Optimization (DPO)
x: "write me a poem about the history of jazz"
preference data
maximum likelihood
final LM

Figure 2.2 DPO method figure. [3]

$$r_\theta(x,y) = log\frac{\pi_\theta(y|x)}{\pi_{ref}(y|x)} \tag{2.8}$$

$$z_0 = \mathbb{E}_{x'\sim\mathcal{D}}[\text{KL}(\pi_\theta(y'|x')\|\pi_{ref}(y'|x'))] \tag{2.9}$$

$$v(x,y) = \begin{cases} \lambda_D\sigma(\beta(r_\theta(x,y) - z_0)) & \text{if } y \sim y_{desirable}|x \\ \lambda_U\sigma(\beta(z_0 - r_\theta(x,y))) & \text{if } y \sim y_{undesirable}|x \end{cases} \tag{2.10}$$

2.4 Direct Preference Optimization

Instead of training a reward from preferences and then optimizing the policy, as RL methods would do, Direct Preference Optimization (DPO) [3] re-expresses the loss function to directly incorporate human preferences such that it can be used to train the policy directly.

This loss function is constructed such that it aligns the model's output probabilities with user preferences, so it directly optimizes the language model over preference data. If users prefer output y_1 over y_2 for input x, the loss function increases the likelihood of y_1 and decreases that of y_2 under the model's policy.

The way it does that is by leveraging an analytical mapping from reward functions to optimal policies and it expresses the reward function though the optimal and reference policy, which in turn means there's no need to train a reward model. It leverages a particular choice of reward model parameterization that enables extraction of its optimal policy in closed form, without an RL training loop.

$$\begin{aligned}&\mathcal{L}_{\mathrm{DPO}}(\pi_\theta;\pi_{\mathrm{ref}}) = \\ &= -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log\sigma\left(\beta\log\left(\frac{\pi_\theta(y_w \mid x)}{\pi_{\mathrm{ref}}(y_w \mid x)}\right) - \beta\log\left(\frac{\pi_\theta(y_l \mid x)}{\pi_{\mathrm{ref}}(y_l \mid x)}\right)\right)\right]\end{aligned} \tag{2.11}$$

Where D is the discriminaator; π_θ and π_{ref} are the optimal and reference policy respectively. y_w is the preferred label and y_l is the dis-preferred label.

The first part of the substitution aims to increase the likelihood of y_w and the second part to decrease the likelihood of y_l.

This approach aims to maximize the likelihood of generating preferred outcomes y_w over less preferred outcomes y_l.

Chapter 3

Proposed Approach

The approach that was introduced in Reinforcement Learning from Compiler Feedback paper removes a biased and expensive part of training - human labeling. However, it still depends on RL which is unstable. So in this thesis the idea introduced by RLCF paper is expanded by using RL free HALOs for coarse tuning, as expained in the previous chapter instead of the RL PPO.

The experiments were executed with two HALOs: KTO and DPO. The coarse tuning will have a bit different definition based on the model (KTO or DPO). But in general the idea is to fine tune a pre-trained model with the feedback of whether or not generated examples compile. The goal of coarse tuning is to train a model that will produce compilable results more often, hence providing better performance on downstream tasks. For KTO there is simply a binary signal that is propagated back to the model. Whereas for DPO, it's a bit more fine grained. If the binary signal responds with compiled, there is a discriminator that is trained to distinguish between model and human generated code. And if it doesn't compile it provides a localized error description.

3.1 Data

Creating the dataset for both KTO and DPO follows the same few steps.

1. Seed: Initial dataset - CodeSearchNet (Appendix A).
2. Base: Dataset consisting of predictions collected from the pre-trained model.
3. Compiler enriched: Contains compiler feedback. This dataset is different for both training models (KTO/DPO), which is explained in the following sections.

3.1.1 Seed Dataset

CodeSearchNet The dataset based on which all models are trained is Code Search Net. This dataset was introduced by GitHub in collaboration with the research community to facilitate the development of models capable of searching and understanding code. It is specifically designed to support code search tasks, making it highly relevant for our research. It is a dataset that consists of (documentation, function) pairs. The dataset schema is as follows:

- `id`: Arbitrary number
- `repository_name`: Name of the GitHub repository
- `func_path_in_repository`: Path to the file which holds the function in the repository
- `func_name`: Name of the function in the file
- `whole_func_string`: Code + documentation of the function
- `language`: Programming language in which the function is written
- `func_code_string`: Function code
- `func_code_tokens`: Tokens yielded by Treesitter
- `func_documentation_string`: Function documentation
- `func_documentation_string_tokens`: Tokens yielded by Treesitter
- `split_name`: Name of the split to which the example belongs (one of train, test or valid)
- `func_code_url`: URL to the function code on GitHub

The dataset comprises over 6 million functions from open-source repositories across six programming languages: Python, JavaScript, Java, PHP, Ruby, and Go.

The dataset is filtered based on a few heuristics. More details can be found in Appendix A. This dataset is used for supervised fine tuning of the base models without any alterations (after filtering).

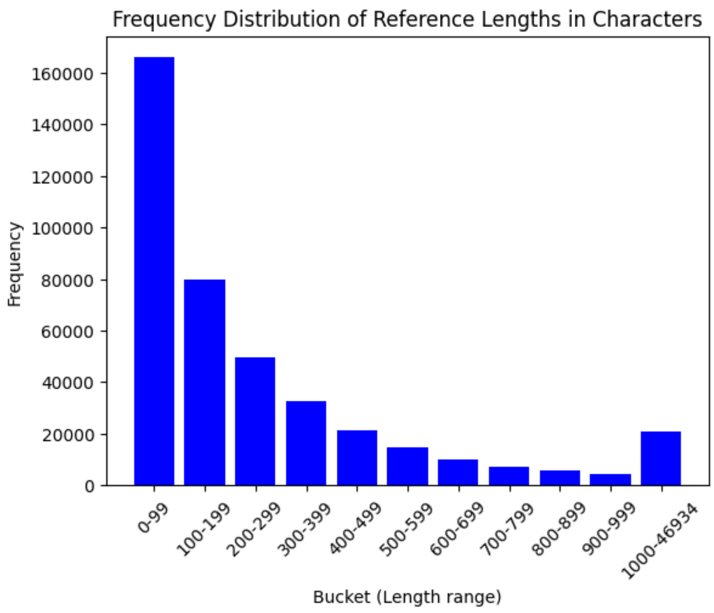

Figure 3.1 Distribution of length of datapoints for Python of original CSN dataset.

3.1.2 Base Dataset

The base dataset contains all columns from CodeSearchNet, as well as predictions collected using the pre-trained model (deepseek-coder1.3b) [13], given a certain prompt. The prompt used for most experiments is empty, as it's proven to be the best one.

In Figure 3.2 is the original distribution of reference lengths in characters. Displayed in Figure 3.3 you can find the distribution of tokenized references of the filtered CSN dataset. The amount of references that have more than 1K tokens is very low. Which is why the base dataset is collected with a fixed max tokens value of 1000.

3.1.3 Compiler Enriched Dataset

The final dataset that is used for training KTO and DPO which require some form of feedback or preference is the compiler enriched dataset.

The compilation process for Java involves invoking the 'javac' compiler and capturing the output. The references that come from the seed dataset CodeSearchNet are plain function that are not placed in a class. Javac cannot compile that as is, so the function is put in a class before compiling. The dataset also doesn't include imports, so most references don't compile which is why import errors are ignored.

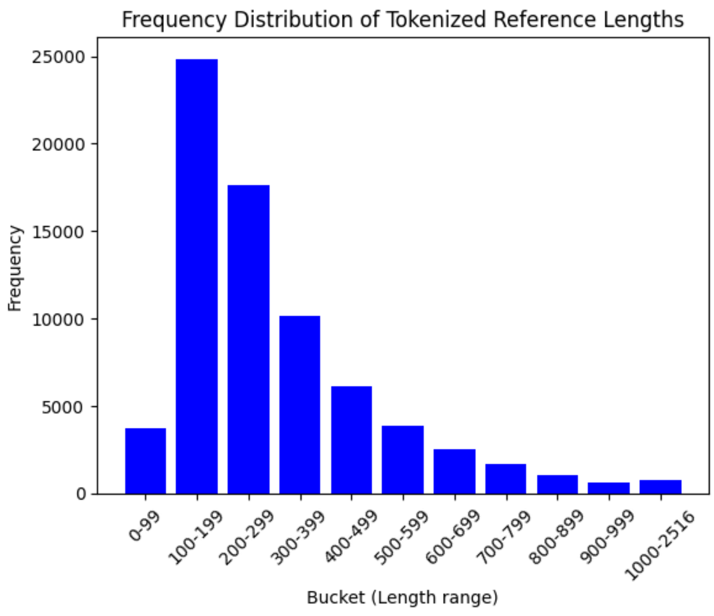

Figure 3.2 Distribution of length of tokenized references for Python after filtering.

For python code compilation leverages the built-in ‘compile‘ function with the ‘’exec’‘ mode to evaluate whether the code can be executed without syntax errors.

Although there are slight differences in the way the data is represented, the main component added in this step to the dataset is the signal of whether or not the generated code from the previous step compiles or not.

3.2 Model

3.2.1 KTO with compiler feedback

KTO requires data with binary signal of whether that is a good or bad example. Usually this requires a human signal classifying the examples as bad or good. In this case instead of using a human in the loop, we use compiler in the loop. So the good/bad signal is actually compiles/doesn’t compile signal.

Data The predictions from the base dataset get classified based on whether or not they compile. The final dataset consists of the prompt and completion, as well as a label of good or bad. The predictions that don’t compile got classified as bad and the ground truth completions are classified as good.

Example of a datapoint which consists of three parts prompt, completion and a flag of whether or not it was rejected.

Prompt

```
def isprime(number):
    """
    Check if a number is a prime number
    :type number: integer
    :param number: The number to check
    """
```

Completion

```
    if number == 1:
        return False
    for i in range(2, int(number ** 0.5) + 1):
        if number % i == 0:
            return False
    return True
```

Label

```
True
```

3.2.2 DPO with compiler feedback

DPO requires data that expresses preference of whether it is a good or bad example. Typically this would be human preference, i.e. labeled data where humans express preference of one over the other example. We replace the human in the loop component with a compiler in the loop approach, using the binary signal of the compiler (code did or did not compile) as representation of good or bad examples.

Data The predictions are classified based on whether they compile or not. The final dataset than contains pairs of reference solution (as a positive example) and a prediction that does not compile (as a negative one). This means that any prediction that compiles cannot not used in the dataset, as it is not a negative example. Therefore, we introduce small modifications to those predictions that compile, such as removing a special symbol, or misspelling a keyword, so that it does not compile.

Example of a datapoint which consists of three parts prompt, rejected completion and the chosen completion out of the two. The chosen completion is always the reference.

Prompt

```
def isprime(number):
    """
    Check if a number is a prime number
    :type number: integer
    :param number: The number to check
    """
```

Chosen

```
    if number == 1:
        return False
    for i in range(2, int(number ** 0.5) + 1):
        if number % i == 0:
            return False
    return True
```

Rejected

```
    if number % 2 == 1:
        return False
    return True
```

3.3 Evaluation

Evaluation is done on two benchmarks - MBXP and HumanEval.

3.3.1 MBXP

MBXP [14] is a benchmark used for the evaluation of code completion. It consists of roughly 1000 examples for every language. One datapoint in the benchmark is a triple of:

- Function definition and documentation
- Reference solution

- A set of tests the generated solution is run on

It is designed to be solvable by entry-level programmers, covering programming fundamentals, standard library functionality.

3.3.2 HumanEval

HumanEval [15] is another benchmark used for evaluating performance of models in the domain of programming and code generation. It consists of 160 problems. The structure of this benchmark is the same as above. It consists of a problem, solution and a set of tests.

3.3.3 Metrics

The metrics used for evaluation are `pass@k`, `compile@k` and `exec@k`. All metrics require k generations from the model on the same input. Then, each metric is calculated over those k generations.

`pass@k` represents the percentage of the datapoints that have at least one of the k generations (on the same input) that passes the tests from the benchmark.

`exec@k` is a softer variant of the metrics, where it calculates the percentage of datapoints where at least one generation can be executed without any errors.

`comp@k` calculates the percentage of datapoints where at least one generation can be compiled without any errors.

Mxeval is a library used for computing execution based metrics. Mxeval provides support for calculating `pass@k` on both benchmarks - MBXP and HumanEval, given predictions. However, as mxeval doesn't include `comp@k` and `exec@k`, which is why we extended mexeval to support those metrics as well

3.4 Resources

TRL [16] is a library built on top of the transformers library that provides the implementation for KTO and DPO, as well as SFT training. It's a full stack tool to fine-tune and align transformer language and diffusion models using methods like KTO, DPO, SFT.

Model DeepSeek Coder [13] is a family of code language models, each trained from scratch on a dataset encompassing of 2 trillion tokens. The training corpus comprises 87% code and 13% natural language datapoints in both English and Chinese. The model series offers various sizes, ranging from 1 billion to 33 billion parameters. Models are trained on project-level code corpus by employing a window size of 16K and a extra fill-in-the-blank task, to support project-level code completion and infilling.

The model used throughout this thesis for all experiment is deepseek-coder1.3b. This model has not been trained on CSN.

Cadence Fine tuning a model of size bigger than 1B parameters becomes increasingly challenging in terms of hardware resources. For that I used cadence plugin [17] which allows you to run a project on AWS machines without having to deal with syncing the project at every change and setting up the project environment.

HuggingFace All models' weights are uploaded to HuggingFace. All predictions collected for the benchmarks are also uploaded as HuggingFace datasets.

Weights & Biases Wandb is used to track models' losses during training and models' performance on the mentioned benchmarks during evaluation.

Chapter 4

Results and Discussion

This chapter presents the conducted experiments and their outcomes. By assessing the base model's performance across different prompts, as shown in the table 4.1, we can see the empty prompt performs best. That is why for the majority of fine tuning experiment the empty prompt was used.

Language	Prompt	pass@1 mbpp	pass@1 humaneval
Python	empty	0.195	0.128
Python	oneshot	0.165	0.146
Python	markdown	0.113	0.097
Java	empty	0.203	0.137
Java	oneshot	0.151	0.043
Java	markdown	0.180	0.118

Table 4.1 Performance of base model on MBPP and HumanEval

The core hypothesis of the thesis revolves around the influence compiler feedback has on model performance. Moreover, it hypothesizes that model performance on downstream tasks, specifically code generation improves. To evaluate this hypothesis, we select a pre-trained LLM, deepseek coder 1.3B, and coarse tune it with compiler feedback with an RL free HALO method. Then that model is optionally fine tuned for the task of function completion. The experiments are run for two languages - Java and Python, and subsequently assess whether these refined models show better results than the original base models.

The data size on which most models were trained on was usually in the range of 5000-20000 datapoints.

Coarse Tuning Results

Purely coarse tuning using KTO or DPO on provides either minor improvements or even slight worsening of the performance on MBPP and HumanEval benchmarks for Python, although not so for Java. Results are shown in Table 4.2 for Python and Table 4.3 for Java. The KTO and DPO models were trained on 5000 examples. Except for the KTO model trained for Java as shown in Table 4.3, which was trained on 7500 examples.

Model	pass@1 mbpp	pass@1 humaneval
deepseek1B	0.194	0.128
deepseek1B + KTO	0.217	0.152
deepseek1B + DPO	0.191	0.128

Table 4.2 Performance of fine tuned model for Python on MBPP and HumanEval

Model	pass@1 mbjp	pass@1 humaneval
deepseek1B	0.203	0.137
deepseek1B + KTO	0.365	0.174
deepseek1B + DPO	0.241	0.112

Table 4.3 Performance of fine tuned model for Java on MBJP and HumanEval

Ablation study for KTO and DPO effect on final performance

A small ablation study is conducted that shows that applying SFT on top of the coarse tuned model seems to provide better results than the purely supervised fine tuned model. It aims to evaluate the impact of different training methods on the performance of the "deepseek1B" model. Specifically, we examine the effects of Supervised Fine-Tuning (SFT), Knowledge Transfer Optimization (KTO), and Data Preference Optimization (DPO), as well as SFT over DPO and KTO fine tuned models. The metrics that are used for evaluation include "pass@1," "exec@1," and "compile@1" for two different benchmarks: on MBXP and HumanEval. Each configuration of the model provides insights into the individual and combined contributions of these techniques.

For both Java and Python pass@1 increases over both the base model and SFT model. More details can be found in Table 4.4 and Table 4.5

The supervised fine tuning was all done with the same configuration of hyperparameters and the size of the data was 5000.

Whereas KTO and DPO were trained on 7500 examples. The KTO and KTO + SFT, as well as DPO and DPO + SFT were trained with the same KTO and DPO configuration of hyperparameters. DPO was trained for 5 epochs, with learning rate 1e-6, and KTO was trained for 3 epoch with learning rate of 5e-5.

Dataset	mbxp			humaneval		
Method	pass	exec	comp	pass	exec	comp
deepseek1B	0.194	0.557	0.822	0.128	0.658	0.914
deepseek1B + SFT	0.235	0.524	0.816	0.235	0.524	0.816
deepseek1B + KTO + SFT	0.281	0.543	0.830	0.201	0.623	0.927
deepseek1B + DPO + SFT	0.247	0.508	0.806	0.207	0.610	0.945
deepseek1B + KTO	0.150	0.290	0.450	0.146	0.372	0.5
deepseek1B + DPO	0.184	0.578	0.805	0.195	0.713	0.933

Table 4.4 Performance of all method combinations for Python. All metrics are `@1`, i.e. `pass@1`, `exec@1`, and `comp@1`.

Dataset	mbxp		humaneval	
Method	pass	comp	pass	comp
deepseek1B	0.203	0.690	0.137	0.596
deepseek1B + SFT	0.239	0.659	0.106	0.640
deepseek1B + KTO + SFT	0.281	0.731	0.137	0.652
deepseek1B + DPO + SFT	0.268	0.734	0.118	0.621
deepseek1B + KTO	0.218	0.735	0.106	0.590
deepseek1B + DPO	0.224	0.711	0.111	0.652

Table 4.5 Performance of all method combinations for Java. All metrics are `@1`, i.e. `pass@1`, `exec@1`, and `comp@1`.

SFT on bigger datasets for Python

In order not to attribute success to the size of the data, we ran experiments for SFT on bigger datasets, of sizes 10000 and 20000. The same hyperparameters were used for all experiments, and from the metric results it seems that the models might be slightly over-fitting.

The results can be observed in Table 4.6

Model	Dataset Size	pass@1 mbpp	pass@1 humaneval
deepseek1B	5K	0.235	0.235
deepseek1B	10K	0.235	0.220
deepseek1B	20K	0.215	0.146

Table 4.6 Performance of SFT 5k 10k 20k model on MBPP and HumanEval for Pyhton

Experiments conclusion From the above data it's visible that coarse tuning on compiler feedback is significantly useful. Looking at Table 4.2 there is a 10% increase in pass@1 after training the model with KTO on only 7500 examples.

Additionally, from the ablation study we can see that only fine tuning in a supervised way has lower results than doing a coarse tuning step before that either with KTO or DPO.

Note that this was a small sized experimentation, as the dataset size is limited and the model used had 1B parameters. So scaling things up looks promising.

4.1 Project

The project can be found on GitHub. And instruction for how to use it are included in the README.md. The models and datasets can be found on HuggingFace.

Conclusion

Conclusion This thesis has successfully managed to experiment with incorporating compiler feedback into RL free HALO methods, specifically KTO and DPO. The results have shown that using compiler feedback indeed improves the model performance on downstream tasks. But also helps model learn how to provide compilable code more often.

This thesis touches upon a bigger topic, which is improving the performance of smaller scale language models. That topic still has a lot of research that needs to be done, however that is a much more complex problem, whose resolution lies beyond the scope of this thesis.

Practical use Practical use case of this work is evident - it improves code generation capabilities of models. That is a big topic in software development industry today, so making that process a bit more efficient by producing compilable code is very important. But also doing so in an inexpensive and more stable way by using compiler feedback and RL free HALOs.

Future research First, this reseach was smaller scale, so future research could start by scalling dataset size and model size in order to make the improvement more noticeable. Additionally, one could build on top of this research by integrating run-time error analysis and other static code checks, such as those provided by integrated development environments (IDEs), into the model training process.

Bibliography

[1] A. Jain et al. “Coarse-Tuning Models of Code with Reinforcement Learning Feedback”. 2023. URL: `https://arxiv.org/abs/2305.18341`.

[2] K. Ethayarajh et al. “KTO: Model Alignment as Prospect Theoretic Optimization”. In: *41 st International Conference on Machine Learning* (2024). URL: `https://arxiv.org/abs/2402.01306`.

[3] R. Rafailov et al. “Direct Preference Optimization: Your Language Model is Secretly a Reward Model”. In: *37th Conference on Neural Information Processing Systems* (2023). URL: `https://arxiv.org/abs/2305.18290`.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* `http://www.deeplearningbook.org`. MIT Press, 2016.

[5] Milan Straka. *Multiclass Logistic Regression, Multilayer Perceptron.* NN picture. 2022. URL: `https://ufal.mff.cuni.cz/~straka/courses/npfl129/2223/slides.pdf/npfl129-2223-04.pdf`.

[6] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[7] Milan Straka. *Introduction to Reinforcement Learning.* MDP picture. 2022. URL: `https://ufal.mff.cuni.cz/~straka/courses/npfl122/2223/slides.pdf/npfl122-2223-01.pdf`.

[8] Ashish Vaswani et al. *Attention Is All You Need.* 2023. arXiv: `1706.03762 [cs.CL]`. URL: `https://arxiv.org/abs/1706.03762`.

[9] Azmat Anwar et al. *Constructing Uyghur Named Entity Recognition System using Neural Machine Translation Tag Projection.* Transformer picture. 2020. URL: `https://www.researchgate.net/publication/344911225_Constructing_Uyghur_Named_Entity_Recognition_System_using_Neural_Machine_Translation_Tag_Projection`.

[10] John Schulman et al. *Proximal Policy Optimization Algorithms.* 2017. arXiv: `1707.06347 [cs.LG]`. URL: `https://arxiv.org/abs/1707.06347`.

[11] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. arXiv: 2002.08155 [cs.CL]. URL: https://arxiv.org/abs/2002.08155.

[12] DANIEL KAHNEMAN AMOS TVERSKY. *Advances in Prospect Theory: Cumulative Representation of Uncertainty*. 1992. URL: https://link.springer.com/article/10.1007/BF00122574.

[13] Daya Guo et al. *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence*. 2024. arXiv: 2401.14196 [cs.SE]. URL: https://arxiv.org/abs/2401.14196.

[14] Ben Athiwaratkun et al. *Multi-lingual Evaluation of Code Generation Models*. 2023. arXiv: 2210.14868 [cs.LG]. URL: https://arxiv.org/abs/2210.14868.

[15] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG]. URL: https://arxiv.org/abs/2107.03374.

[16] Leandro von Werra et al. *TRL: Transformer Reinforcement Learning*. Version 0.2.1. URL: https://github.com/huggingface/trl.

[17] A. Trofimov et al. “JetTrain: IDE-Native Machine Learning Experiments”. In: *ACM ISBN* (2024). URL: https://arxiv.org/abs/2402.10857.

Appendix A

Code Search Net Filtering

From CSN dataset I've used several heuristics to collect a subset, approximately 1/4 of the original size of the 2 datasets:

- Filter by documentation size (number of characters): [60, 2000]
- Filter by code size (number of characters): [60, 4000]
- Filter by space count: [6, inf]
- Filter out datapoints whose documentation contains: "todo", "fixme", "tbd"
- Filter out documentation that contains the name of the method itself
- Filter documentation based on keywords it should contains, for explain the parameters, and functionality of the code
- Filter code by number of lines [0, 70)
- Filter out code that has exact match lines (code repetitiveness)
- Filter code by number of parameters a function accepts - [0, 8)
- Filter code by number of nested blocks - [0, 6)
- Filter out documentation - code ratio (0.1, 1]

After post-processing the number of datapoints was cut down to 72K.

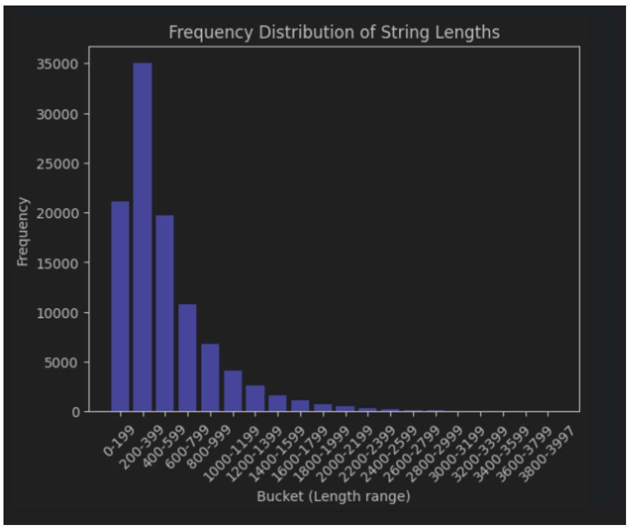

Figure A.1 Distribution of length of datapoints after postprocessing.