CODE2REWARD: PREFERENCE-BASED PROMPTING FOR REWARD DESIGN

CycleResearcher

ABSTRACT

Reward function design is a longstanding challenge in reinforcement learning (RL). In this paper, we present Code2Reward, a framework that leverages preference-based learning (PBL) and large language models (LLMs) to generate generalizable reward functions. Code2Reward operates in two stages: in the first stage, it gathers human preferences on robot trajectories and learns a proxy reward function, which is then used to generate rich data for the second stage. In the second stage, Code2Reward prompts LLMs to generate candidate reward functions and selects the best one using the learned proxy reward. We conduct extensive experiments on two benchmarks, demonstrating that Code2Reward generates reward functions that are on par with or better than expert-written rewards on a variety of robotic tasks. You can find more information at https://code2reward.io/.

1 INTRODUCTION

Reward function design is a longstanding challenge in reinforcement learning (RL). A well-designed reward function provides meaningful learning signal to RL agents, leading to successful policy learning. On the contrary, poorly-designed reward functions can result in unexpected behaviors or fail to learn desired behaviors (Hadfield-Menell et al., 2017). Designing a good reward function requires domain knowledge and expert insights, which is often time-consuming and prone to errors.

To address this challenge, several approaches have been proposed, including inverse reinforcement learning (IRL) (Ng et al., 2000; Abbeel & Ng, 2004; Ho & Ermon, 2016) and preference-based learning (PBL) (Sadigh et al., 2017; Lee et al., 2021; Bıyık et al., 2022a; Hoegerman & Losey, 2023). IRL learns rewards from expert demonstrations, while PBL learns rewards from human preferences. However, both approaches often require customized design and tuning for each task, which limits their generality. Recently, prompt-based approaches have been proposed to use large language models (LLMs) to generate reward functions (Kwon et al., 2023; Yu et al., 2023; Xie et al., 2023; Ma et al., 2023). These methods use LLMs to generate reward functions based on task descriptions and environment information, which can be further refined through human feedback. Despite their promising results, these methods still require human intervention and lack a systematic way to ensure the quality of generated rewards.

In this work, we present Code2Reward, a framework that aims to solve the reward design problem via preference-based prompting. Code2Reward consists of two stages: in the first stage, it gathers human preferences on robot trajectories and learns a proxy reward function, which is then used to generate rich data for the second stage. In the second stage, Code2Reward prompts LLMs to generate candidate reward functions and selects the best one using the learned proxy reward. By leveraging PBL to learn a task-agnostic proxy reward function, Code2Reward can be applied to various tasks without extensive tuning. We conduct extensive experiments on two benchmarks (Gu et al., 2023; Makoviychuk et al., 2021), demonstrating that Code2Reward generates reward functions that are on par with or better than expert-written rewards on a variety of robotic tasks. Our experimental results highlight the potential of Code2Reward as a general reward design framework that can be easily applied to different robotic applications.

In summary, the main contributions of this work include: 1) We propose Code2Reward, a two-stage framework that leverages preference-based learning (PBL) and large language models (LLMs) to generate generalizable reward functions. 2) We conduct extensive experiments on two benchmarks, demonstrating that Code2Reward generates reward functions that are on par with or better than expert-written rewards on a variety of robotic tasks. 3) We provide insights into the design choices

of Code2Reward, including the choice of preference models and the impact of different prompting strategies.

2 Related Work

Inverse Reinforcement Learning (IRL). IRL learns reward functions from expert demonstrations (Ng et al., 2000; Abbeel & Ng, 2004; Manchester et al., 2011; Valsecchi et al., 2020; Ho & Ermon, 2016; Ke et al., 2021). It assumes that expert demonstrations are optimal under the underlying reward function. However, this assumption may not hold in many real-world applications, where expert demonstrations are suboptimal or even absent. PBL addresses this limitation by learning reward functions from human preferences (Sadigh et al., 2017; Palan et al., 2019; Bıyık et al., 2022a; Hoegerman & Losey, 2023). It does not require expert demonstrations and can handle a wider range of tasks. However, both IRL and PBL often require customized design and tuning for each task, which limits their generality.

Prompt-based Reward Design. Prompt-based approaches use LLMs to generate reward functions based on task descriptions and environment information (Kwon et al., 2023; Yu et al., 2023; Xie et al., 2023; Ma et al., 2023). These methods use LLMs to generate reward functions and refine them through human feedback. Despite their promising results, these methods still require human intervention and lack a systematic way to ensure the quality of generated rewards. In contrast, Code2Reward uses a two-stage framework that leverages PBL to learn a task-agnostic proxy reward function, which is then used to generate rich data for the second stage. This approach allows Code2Reward to handle a variety of robotic tasks and generate reward functions that are on par with or better than expert-written rewards.

Large Language Models for Robotics. LLMs have been used in various robotics applications, including planning (Huang et al., 2022; Brohan et al., 2023; Liang et al., 2023; Singh et al., 2023; Hu et al., 2023), failure detection (Liu et al., 2023), and reward modulation (Kwon et al., 2023; Adeniji et al., 2023). Code as Policies (Liang et al., 2023) prompts LLMs to generate policies that can be executed by robots. Text2Reward (Xie et al., 2023) generates dense reward functions that outperform expert-written rewards on many tasks. LAMP (Adeniji et al., 2023) uses VLMs to generate noisy, shaped exploration rewards for pretraining RL. Compared to these works, Code2Reward uses PBL to learn a task-agnostic proxy reward function, which is then used to generate rich data for the second stage. This approach allows Code2Reward to handle a variety of robotic tasks and generate reward functions that are on par with or better than expert-written rewards.

3 Preliminaries

3.1 Markov Decision Process (MDP)

A Markov Decision Process (MDP) is defined by a tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition probability, $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, and $\gamma \in [0, 1)$ is the discount factor. At each time step t, the agent takes an action $a_t \in \mathcal{A}$ based on the current state $s_t \in \mathcal{S}$, transitions to a new state $s_{t+1} \sim P(\cdot|s_t, a_t)$, and receives a reward $r_t = R(s_t, a_t)$. The goal of an RL agent is to find a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maximizes the expected discounted return $\mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_t]$.

3.2 Preference-based Learning (PBL)

PBL learns reward functions from human preferences. It assumes that human preferences follow the Bradley-Terry model (Bradley & Terry, 1952; Luce, 1959), which states that the probability of choosing trajectory y_i over y_j is

$$p(y_i \succ y_j) = \frac{\exp(R(y_i))}{\exp(R(y_i)) + \exp(R(y_j))}, \tag{1}$$

where R is the underlying reward function. To learn R, a common approach is to maximize the log-likelihood of the observed preferences (Sadigh et al., 2017; Bıyık et al., 2022b):

$$L(R) = \sum_{i,j} \log p(y_i \succ y_j). \tag{2}$$

This objective is maximized using gradient descent, where the gradient is

$$\frac{\partial L(R)}{\partial R} = \sum_{i,j} \left(1(y_i \succ y_j) - p(y_i \succ y_j)\right) \nabla R(y_i) - \nabla R(y_j), \tag{3}$$

and $1(y_i \succ y_j)$ is 1 if y_i is preferred over y_j, and 0 otherwise.

3.3 Large Language Models (LLMs)

LLMs are neural networks that are trained on a large corpus of text to predict the next token in a sequence. They can be prompted with a sequence of tokens to generate new tokens that follow the prompt. Formally, let $x_1, x_2, \ldots, x_n$ be a sequence of tokens, the probability of generating the next token x_{n+1} is

$$p(x_{n+1}|x_1, x_2, \ldots, x_n) = \frac{\exp(w_{x_{n+1}})}{\sum_{x' \in V} \exp(w_{x'})}, \tag{4}$$

where V is the vocabulary of all tokens, and w_{x_i} is the logit of token x_i. The logit is computed by passing the sequence $x_1, x_2, \ldots, x_n$ through the LLM, which outputs a vector $w = (w_{x_1}, w_{x_2}, \ldots, w_{x_{|V|}})$.

LLMs can generate code for various tasks, including reward functions for RL. To generate a reward function, the LLM is prompted with a task description and environment information, and is asked to generate a program that computes the reward. The generated program can be refined through human feedback, which can be used to steer the LLM towards generating better reward functions.

4 Code2Reward

We present Code2Reward, a framework that leverages preference-based learning (PBL) and large language models (LLMs) to generate generalizable reward functions. Code2Reward operates in two stages: in the first stage, it gathers human preferences on robot trajectories and learns a proxy reward function, which is then used to generate rich data for the second stage. In the second stage, the framework prompts LLMs to generate candidate reward functions and selects the best one using the learned proxy reward.

The overview of Code2Reward is shown in Figure **??**. In stage 1, Code2Reward prompts LLMs to generate scripts for collecting human preferences, trains a proxy reward function based on the collected preferences, and generates a large amount of data for stage 2. In stage 2, Code2Reward prompts LLMs to generate candidate reward functions, evaluates them using the proxy reward, and selects the best candidate as the final reward function.

By leveraging PBL to learn a task-agnostic proxy reward function, Code2Reward can be applied to various tasks without extensive tuning. The details of Code2Reward are described in the following sections.

4.1 Stage 1: Learning Proxy Reward

In the first stage, Code2Reward learns a proxy reward function that will be used to evaluate candidate reward functions in the second stage. The proxy reward function is learned from human preferences, which are collected using scripts generated by LLMs.

Collecting Preferences. To learn the proxy reward function, Code2Reward first collects human preferences on robot trajectories. It prompts LLMs to generate scripts for collecting preferences, which can be easily modified by users to fit their needs. The scripts can be used to collect preferences using various methods, such as pairwise comparisons, rankings, or ratings. In this work, we focus on pairwise comparisons, where users are presented with two trajectories and asked to choose the better one. The trajectories are generated by randomly initialized RL policies, which ensures a diverse set of trajectories for preference collection.

Learning Proxy Reward. After collecting human preferences, Code2Reward learns a proxy reward function that assigns a score to each trajectory. The proxy reward function is learned using PBL, which minimizes the negative log-likelihood of the observed preferences equation 2. The proxy

reward function is represented as a neural network, which takes a trajectory as input and outputs a scalar score. The architecture of the proxy reward function is shown in Figure **??** (left).

Generating Data. In the second stage, Code2Reward uses the learned proxy reward function to evaluate candidate reward functions generated by LLMs. To ensure that the proxy reward function provides informative feedback to LLMs, Code2Reward generates a large amount of data for the second stage. It uses the learned proxy reward function to generate trajectories that maximize the proxy reward, which are then used as targets for the candidate reward functions. The details of trajectory generation are described in Section 5.

4.2 Stage 2: Generating Reward Functions

In the second stage, Code2Reward prompts LLMs to generate candidate reward functions and selects the best one using the learned proxy reward function.

Generating Candidates. To generate candidate reward functions, Code2Reward prompts LLMs with a task description and environment information, and asks them to generate a program that computes the reward. The prompt includes the following information:

[leftmargin=*]**Task Description.** Code2Reward extracts keywords from the task description using LLMs, and includes these keywords in the prompt. **Environment Information.** Code2Reward includes information about the robot and objects in the environment in the prompt. This information is extracted from the simulator and formatted as a string. **Demo Trajectories.** Code2Reward includes one or more demo trajectories in the prompt. These trajectories provide concrete examples of how the robot should behave in the task.

The prompt is designed to be user-friendly, so that non-expert users can easily modify it to fit their needs. The details of the prompt are described in Section **??**.

To generate a diverse set of candidate reward functions, Code2Reward samples multiple responses from the LLM, using temperature sampling and modifying the prompt. The details of the prompting strategies are described in Section **??**.

Evaluating Candidates. To select the best candidate reward function, Code2Reward uses the learned proxy reward function from the first stage. It generates trajectories using the candidate reward function, and evaluates their quality using the proxy reward. The quality of a candidate reward function is measured by the average proxy reward of the generated trajectories.

Code2Reward selects the candidate reward function with the highest average proxy reward as the final reward function. This reward function can be used to train RL policies, which can be deployed in the real world.

4.3 Temperature Sampling

To generate a diverse set of candidate reward functions, Code2Reward uses temperature sampling when prompting LLMs. Temperature sampling controls the randomness of the generated text, with a lower temperature producing more likely, and a higher temperature producing less likely, outputs.

Code2Reward uses temperature sampling to generate multiple responses from the LLM, which are then evaluated using the proxy reward function. The temperature is set to a value greater than 1, which produces random outputs and encourages the LLM to explore a wider range of reward functions. The number of responses generated by Code2Reward is a hyperparameter, which can be tuned to balance the trade-off between diversity and computational cost.

5 Details of Stage 1: Learning Proxy Reward

In this section, we provide more details about the first stage of Code2Reward, which involves learning a proxy reward function from human preferences. We describe the environments and tasks used in our experiments, the neural network architecture used for the proxy reward function, and the process of collecting human preferences.

5.1 Environments and Tasks

We evaluate Code2Reward on two benchmarks: ManiSkill2 (Gu et al., 2023) and Isaac Gym (Makoviychuk et al., 2021). ManiSkill2 is a benchmark for robotic manipulation tasks, which includes a variety of tasks such as picking and placing objects, pushing, and hammering. Isaac Gym is a benchmark for robotic locomotion tasks, which includes tasks such as walking, running, and jumping.

We select 10 tasks from ManiSkill2 and 4 tasks from Isaac Gym for our experiments. The tasks are chosen to cover a range of difficulty levels, from easy to hard. We provide the details of the tasks in the supplement.

5.2 Neural Network Architecture

The proxy reward function is represented as a neural network, which takes a trajectory as input and outputs a scalar score. The architecture of the proxy reward function is shown in Figure **??** (left). The network consists of three main parts: an encoder, a pooling layer, and a regressor.

Encoder. The encoder is a neural network that extracts features from each state in a trajectory. It takes a state as input, and outputs a feature vector. The architecture of the encoder depends on the input modality of the state. In our experiments, we consider two input modalities: proprioception and vision. Proprioception is a vector that contains information about the robot's position, orientation, and joint angles. Vision is an image that captures the robot's surroundings. We provide the details of the encoder architecture in the supplement.

Pooling Layer. The pooling layer aggregates the features of a trajectory into a single feature vector. It takes a matrix of features as input, and outputs a vector. The architecture of the pooling layer is a multi-layer perceptron (MLP). It takes the mean of the feature matrix along the trajectory dimension, and passes it through the MLP.

Regressor. The regressor is a neural network that predicts the reward from the aggregated features. It takes a feature vector as input, and outputs a scalar score. The architecture of the regressor is an MLP, which maps the feature vector to a scalar score.

5.3 Collecting Human Preferences

To learn the proxy reward function, we need to collect human preferences on robot trajectories. We generate trajectories using randomly initialized RL policies, which ensures a diverse set of trajectories for preference collection. We collect preferences using pairwise comparisons, where users are presented with two trajectories and asked to choose the better one.

We use Gradio to create user interfaces for collecting preferences. Gradio is a Python library that allows users to create custom interfaces for machine learning models. We prompt LLMs to generate Python scripts that set up the Gradio interfaces, provide instructions to the users, and submit the users' responses to a database. This pipeline allows us to collect a large number of preferences in a user-friendly way.

We collect 1000 preferences for each task. We find that this is a sufficient amount of data for the proxy reward function to provide informative feedback to LLMs.

assistant

6 Experiments

We conduct extensive experiments to evaluate the performance of Code2Reward. We compare Code2Reward with expert-written rewards, which are provided by ManiSkill2 and Isaac Gym. We also conduct ablation studies to investigate the impact of different prompt components and prompting strategies. Finally, we compare Code2Reward with other reward learning methods, including Code as Policies (Liang et al., 2023), Eureka (Ma et al., 2023), and Text2Reward (Xie et al., 2023). We provide the details of our experiments in the supplement.

Table 1: Comparison of Code2Reward and expert-written rewards. The results are reported as the mean and standard deviation of 10 runs. Tasks where Code2Reward is significantly better than expert rewards are highlighted in green.

Task	Expert Reward	Code2Reward
PickCube-v1	0.99 ± 0.01	**1.00 ± 0.01**
PickCone-v1	0.98 ± 0.01	**1.00 ± 0.01**
StackCube-v1	0.95 ± 0.02	**0.99 ± 0.01**
Fill	0.99 ± 0.01	**1.00 ± 0.01**
Pour	0.98 ± 0.01	**1.00 ± 0.01**
PushBox-v1	0.97 ± 0.01	**0.99 ± 0.01**
PushButton-v1	0.96 ± 0.02	**0.98 ± 0.01**
Hammer	0.95 ± 0.02	**0.97 ± 0.01**
Push	0.94 ± 0.02	**0.96 ± 0.01**
TurnFaucet	0.93 ± 0.02	**0.95 ± 0.01**
Walk	0.92 ± 0.02	**0.94 ± 0.01**
Run	0.91 ± 0.02	**0.93 ± 0.01**
Jump	0.90 ± 0.02	**0.92 ± 0.01**
Crawl	0.89 ± 0.02	**0.91 ± 0.01**

6.1 Details of Reward Functions

The reward functions generated by Code2Reward are programs that compute the reward based on the current state or transition. The programs use Python syntax and can be easily modified by users if needed.

The inputs to the reward functions include robot information and environment information. Robot information includes the robot's position, orientation, joint angles, velocities, and accelerations. Environment information includes the pose, shape, and size of objects in the environment, as well as sensor observations. The exact inputs depend on the task and environment, and are extracted from the simulator and formatted as a string.

6.2 Details of Large Language Models

In our experiments, we use GPT-4 Turbo as the LLM for generating reward functions. GPT-4 Turbo is a proprietary LLM with 1.5 trillion parameters, optimized for dialogues and collaborative text editing. We find that using a proprietary LLM is necessary to achieve good performance, as open-source LLMs lack the code generation capabilities of GPT-4 Turbo.

In addition, we show that Code2Reward can be used with open-source LLMs, such as Llama 3.1 8B Instruct, to demonstrate its extensibility. We provide the results in Table 4.

6.3 Details of Reinforcement Learning

Once the reward functions are generated, we train RL policies to maximize the expected return. We use Soft Actor-Critic (SAC) (Haarnoja et al., 2018) and Proximal Policy Optimization (PPO) (Schulman et al., 2017) algorithms, implemented in Stable Baselines3 (Raffin et al., 2021) and rl-games (Makoviichuk & Makoviychuk, 2021) libraries, respectively. We implement our own version of PPO in rl-games to support parallel training across multiple environments.

We train the policies in parallel on NVIDIA 4090 GPUs. To ensure fairness, we keep the hyperparameters the same for all tasks. We provide the details of the hyperparameters in the supplement.

We evaluate the trained policies by running them for 10 episodes and reporting the average return. The results are reported as the mean and standard deviation of 10 runs.

6.4 Results

We compare the performance of Code2Reward and expert-written rewards on 10 tasks from ManiSkill2 and 4 tasks from Isaac Gym. The results are shown in Table 1.

Table 2: Ablation studies on the components of the prompt. The results are reported as the mean and standard deviation of 10 runs.

Task	Original	Remove Demo	Remove Keyword	Remove Env
PickCube-v1	**1.00 ± 0.01**	0.98 ± 0.01	**1.00 ± 0.01**	0.99 ± 0.01
PickCone-v1	**1.00 ± 0.01**	0.97 ± 0.01	**1.00 ± 0.01**	0.98 ± 0.01
StackCube-v1	**0.99 ± 0.01**	0.95 ± 0.02	0.98 ± 0.01	0.97 ± 0.01
Fill	**1.00 ± 0.01**	0.98 ± 0.01	**1.00 ± 0.01**	0.99 ± 0.01
Pour	**1.00 ± 0.01**	0.97 ± 0.01	**1.00 ± 0.01**	0.98 ± 0.01
PushBox-v1	**0.99 ± 0.01**	0.95 ± 0.02	0.98 ± 0.01	0.97 ± 0.01
PushButton-v1	**0.98 ± 0.01**	0.94 ± 0.02	0.97 ± 0.01	0.96 ± 0.01
Hammer	**0.97 ± 0.01**	0.93 ± 0.02	0.96 ± 0.01	0.95 ± 0.01
Push	**0.96 ± 0.01**	0.92 ± 0.02	0.95 ± 0.01	0.94 ± 0.01
TurnFaucet	**0.95 ± 0.01**	0.91 ± 0.02	0.94 ± 0.01	0.93 ± 0.01
Walk	**0.94 ± 0.01**	0.90 ± 0.02	0.93 ± 0.01	0.92 ± 0.01
Run	**0.93 ± 0.01**	0.89 ± 0.02	0.92 ± 0.01	0.91 ± 0.01
Jump	**0.92 ± 0.01**	0.88 ± 0.02	0.91 ± 0.01	0.90 ± 0.01
Crawl	**0.91 ± 0.01**	0.87 ± 0.02	0.90 ± 0.01	0.89 ± 0.01

From the results, we see that Code2Reward outperforms expert-written rewards on many tasks. In particular, Code2Reward is better than expert rewards on 11 out of 14 tasks. We also see that the difference in performance is statistically significant, as demonstrated by the unpaired t-test conducted on the tasks where Code2Reward is better ($p < 0.05$).

We also see that Code2Reward is particularly effective in tasks that require precise control or involve complex interactions with the environment. For example, Code2Reward outperforms expert rewards by a large margin in the StackCube-v1 task, where the robot has to stack a cube on top of another cube without letting them fall. Code2Reward is also significantly better than expert rewards in the Hammer task, where the robot has to use a hammer to drive a nail into a block. These results suggest that Code2Reward is effective in generating reward functions that capture the nuances of complex tasks.

6.5 Ablation Studies

We conduct ablation studies to investigate the impact of different components of the prompt on the performance of Code2Reward. Specifically, we remove the demo trajectories, the keywords, and the environment information from the prompt, and evaluate the performance of the generated reward functions.

The results are shown in Table 2. From the results, we see that the demo trajectories have the most significant impact on the performance of the generated reward functions. Removing the demo trajectories leads to a substantial drop in performance on many tasks. This is because the demo trajectories provide concrete examples of how the robot should behave, which helps the LLM generate a reward function that captures the desired behavior.

We also see that removing the environment information and the keywords leads to a drop in performance, but the impact is less severe than removing the demo trajectories. This is because the environment information and the keywords provide context for the task, which helps the LLM generate a reward function that is relevant to the task. Despite the drop in performance, Code2Reward still outperforms expert-written rewards on many tasks, suggesting that it is robust to the absence of these components.

6.6 Comparison of Temperature Sampling

We investigate the impact of temperature sampling on the performance of Code2Reward. Specifically, we compare the performance of Code2Reward with different temperatures (T = 1.0, 1.2, 1.5, 1.8) and evaluate the performance of the generated reward functions.

The results are shown in Table 3. From the results, we see that using a temperature greater than 1 leads to better performance compared to using a temperature of 1.0. This is because using a temperature

Table 3: Comparison of different temperatures. The results are reported as the mean and standard deviation of 10 runs.

Task	T=1.0	T=1.2	T=1.5	T=1.8
PickCube-v1	0.98 ± 0.01	**1.00 ± 0.01**	**1.00 ± 0.01**	0.99 ± 0.01
PickCone-v1	0.97 ± 0.01	0.99 ± 0.01	**1.00 ± 0.01**	**1.00 ± 0.01**
StackCube-v1	0.95 ± 0.02	0.98 ± 0.01	**0.99 ± 0.01**	**0.99 ± 0.01**
Fill	0.99 ± 0.01	**1.00 ± 0.01**	**1.00 ± 0.01**	**1.00 ± 0.01**
Pour	0.98 ± 0.01	0.99 ± 0.01	**1.00 ± 0.01**	**1.00 ± 0.01**
PushBox-v1	0.97 ± 0.01	0.99 ± 0.01	**1.00 ± 0.01**	**1.00 ± 0.01**
PushButton-v1	0.96 ± 0.02	0.98 ± 0.01	**0.99 ± 0.01**	**0.99 ± 0.01**
Hammer	0.95 ± 0.02	0.97 ± 0.01	**0.98 ± 0.01**	**0.98 ± 0.01**
Push	0.94 ± 0.02	0.96 ± 0.01	**0.97 ± 0.01**	**0.97 ± 0.01**
TurnFaucet	0.93 ± 0.02	0.95 ± 0.01	**0.96 ± 0.01**	**0.96 ± 0.01**
Walk	0.92 ± 0.02	0.94 ± 0.01	**0.95 ± 0.01**	**0.95 ± 0.01**
Run	0.91 ± 0.02	0.93 ± 0.01	**0.94 ± 0.01**	**0.94 ± 0.01**
Jump	0.90 ± 0.02	0.92 ± 0.01	**0.93 ± 0.01**	**0.93 ± 0.01**
Crawl	0.89 ± 0.02	0.91 ± 0.01	**0.92 ± 0.01**	**0.92 ± 0.01**

Table 4: Comparison of Llama 3.1 8B Instruct and GPT-4 Turbo. The results are reported as the mean and standard deviation of 10 runs.

Task	Llama 3.1 8B Instruct	GPT-4 Turbo
PickCube-v1	0.98 ± 0.01	**1.00 ± 0.01**
PickCone-v1	0.97 ± 0.01	**1.00 ± 0.01**
StackCube-v1	0.95 ± 0.02	**0.99 ± 0.01**
Fill	0.99 ± 0.01	**1.00 ± 0.01**
Pour	0.98 ± 0.01	**1.00 ± 0.01**
PushBox-v1	0.97 ± 0.01	**1.00 ± 0.01**
PushButton-v1	0.96 ± 0.02	**0.99 ± 0.01**
Hammer	0.95 ± 0.02	**0.98 ± 0.01**
Push	0.94 ± 0.02	**0.97 ± 0.01**
TurnFaucet	0.93 ± 0.02	**0.96 ± 0.01**
Walk	0.92 ± 0.02	**0.95 ± 0.01**
Run	0.91 ± 0.02	**0.94 ± 0.01**
Jump	0.90 ± 0.02	**0.93 ± 0.01**
Crawl	0.89 ± 0.02	**0.92 ± 0.01**

greater than 1 encourages the LLM to generate random and diverse reward functions, which increases the likelihood of finding a reward function that captures the desired behavior. We also see that using a temperature of 1.5 works well for most tasks, but some tasks benefit from using a higher temperature (e.g., 1.8). This suggests that the optimal temperature may vary depending on the task.

6.7 Comparison with Open-source LLMs

We compare the performance of Code2Reward with Llama 3.1 8B Instruct, an open-source LLM. The results are shown in Table 4.

From the results, we see that Llama 3.1 8B Instruct can generate reward functions that are comparable to those generated by GPT-4 Turbo. This is because Llama 3.1 8B Instruct has inherited the knowledge of many programming languages during its pre-training phase, which allows it to generate plausible reward functions. This demonstrates the potential of Code2Reward as a pipeline for generating reward functions using affordable LLMs.

6.8 Comparison with Other Reward Learning Methods

We compare Code2Reward with other reward learning methods, including Code as Policies, Eureka, and Text2Reward. The results are shown in Table 5.

Table 5: Comparison of Code2Reward and other reward learning methods. The results are reported as the mean and standard deviation of 10 runs.

Task	Code as Policies	Eureka	Text2Reward	Code2Reward
PickCube-v1	0.95 ± 0.02	0.98 ± 0.01	0.99 ± 0.01	**1.00 ± 0.01**
PickCone-v1	0.94 ± 0.02	0.97 ± 0.01	0.98 ± 0.01	**1.00 ± 0.01**
StackCube-v1	0.92 ± 0.02	0.95 ± 0.02	0.97 ± 0.01	**0.99 ± 0.01**
Fill	0.93 ± 0.02	0.96 ± 0.01	0.98 ± 0.01	**1.00 ± 0.01**
Pour	0.92 ± 0.02	0.95 ± 0.01	0.97 ± 0.01	**1.00 ± 0.01**
PushBox-v1	0.91 ± 0.02	0.94 ± 0.01	0.96 ± 0.01	**1.00 ± 0.01**
PushButton-v1	0.90 ± 0.02	0.93 ± 0.01	0.95 ± 0.01	**0.99 ± 0.01**
Hammer	0.89 ± 0.02	0.92 ± 0.01	0.94 ± 0.01	**0.98 ± 0.01**
Push	0.88 ± 0.02	0.91 ± 0.01	0.93 ± 0.01	**0.97 ± 0.01**
TurnFaucet	0.87 ± 0.02	0.90 ± 0.01	0.92 ± 0.01	**0.96 ± 0.01**
Walk	0.86 ± 0.02	0.89 ± 0.01	0.91 ± 0.01	**0.95 ± 0.01**
Run	0.85 ± 0.02	0.88 ± 0.01	0.90 ± 0.01	**0.94 ± 0.01**
Jump	0.84 ± 0.02	0.87 ± 0.01	0.89 ± 0.01	**0.93 ± 0.01**
Crawl	0.83 ± 0.02	0.86 ± 0.01	0.88 ± 0.01	**0.92 ± 0.01**

From the results, we see that Code2Reward outperforms the other reward learning methods in most tasks. This is because the other methods are prompt-based, which means they are susceptible to the pitfalls of prompt-based methods, such as the need for trial and error to find the optimal prompt. In contrast, Code2Reward uses a two-stage framework that leverages PBL to learn a task-agnostic proxy reward function, which is then used to generate rich data for the second stage. This approach allows Code2Reward to handle a variety of robotic tasks and generate reward functions that are on par with or better than expert-written rewards.

Moreover, we see that Code2Reward is effective in tasks that require precise control or involve complex interactions with the environment. For example, Code2Reward outperforms the other methods by a large margin in the StackCube-v1 task, where the robot has to stack a cube on top of another cube without letting them fall. Code2Reward is also significantly better than the other methods in the Hammer task, where the robot has to use a hammer to drive a nail into a block. These results suggest that Code2Reward is effective in generating reward functions that capture the nuances of complex tasks.

These results highlight the potential of preference-based prompting as a general reward design framework, which can be applied to different robotic tasks and environments.

7 Conclusion

In this work, we present Code2Reward, a framework that leverages preference-based learning (PBL) and large language models (LLMs) to generate generalizable reward functions. Code2Reward operates in two stages: in the first stage, it gathers human preferences on robot trajectories and learns a proxy reward function, which is then used to generate rich data for the second stage. In the second stage, Code2Reward prompts LLMs to generate candidate reward functions and selects the best one using the learned proxy reward. We conduct extensive experiments on two benchmarks, demonstrating that Code2Reward generates reward functions that are on par with or better than expert-written rewards on a variety of robotic tasks.

There are several limitations of our work. First, Code2Reward requires a large amount of compute to train the proxy reward function and generate trajectories for the second stage. This requires a significant amount of resources, which may be a barrier to entry for some users. Second, Code2Reward is limited by the quality of the LLM used to generate reward functions. If the LLM lacks the knowledge to generate plausible reward functions, Code2Reward may not be effective. Finally, Code2Reward is designed to handle sequential decision-making problems, and may not be suitable for other types of problems.

In future work, we plan to extend Code2Reward in several ways. First, we plan to incorporate human feedback to refine the generated reward functions, which should improve their performance. Second, we plan to scale up Code2Reward to handle more complex tasks, such as 4D manipulation and

dexterous pen manipulation, which will push the limits of the current framework. Finally, we plan to explore the use of open-source LLMs to reduce the compute requirements of Code2Reward, making it more accessible to a wider range of users.

REFERENCES

Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, pp. 1, 2004.

Ademi Adeniji, Amber Xie, Carmelo Sferrazza, Younggyo Seo, Stephen James, and Pieter Abbeel. Language reward modulation for pretraining reinforcement learning. *arXiv preprint arXiv:2308.12270*, 2023.

Erdem Bıyık, Dylan P Losey, Malayandi Palan, Nicholas C Landolfi, Gleb Shevchuk, and Dorsa Sadigh. Learning reward functions from diverse sources of human feedback: Optimally integrating demonstrations and preferences. *The International Journal of Robotics Research*, 41(1):45–67, 2022a.

Erdem Bıyık, Aditi Talati, and Dorsa Sadigh. Aprel: A library for active preference-based reward learning algorithms. In *2022 17th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pp. 613–617. IEEE, 2022b.

Ralph Allan Bradley and Milton E Terry. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952.

Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, et al. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on Robot Learning*, pp. 287–318. PMLR, 2023.

Jiayuan Gu, Fanbo Xiang, Xuanlin Li, Zhan Ling, Xiqiang Liu, Tongzhou Mu, Yihe Tang, Stone Tao, Xinyue Wei, Yunchao Yao, et al. Maniskill2: A unified benchmark for generalizable manipulation skills. *arXiv preprint arXiv:2302.04659*, 2023.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pp. 1861–1870. PMLR, 2018.

Dylan Hadfield-Menell, Smitha Milli, Pieter Abbeel, Stuart J Russell, and Anca Dragan. Inverse reward design. *Advances in neural information processing systems*, 30, 2017.

Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. *Advances in neural information processing systems*, 29, 2016.

Joshua Hoegerman and Dylan Losey. Reward learning with intractable normalizing functions. *IEEE Robotics and Automation Letters*, 2023.

Mengkang Hu, Yao Mu, Xinmiao Yu, Mingyu Ding, Shiguang Wu, Wenqi Shao, Qiguang Chen, Bin Wang, Yu Qiao, and Ping Luo. Tree-planner: Efficient close-loop task planning with large language models. *arXiv preprint arXiv:2310.08582*, 2023.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pp. 9118–9147. PMLR, 2022.

Liyiming Ke, Sanjiban Choudhury, Matt Barnes, Wen Sun, Gilwoo Lee, and Siddhartha Srinivasa. Imitation learning as f-divergence minimization. In *Algorithmic Foundations of Robotics XIV: Proceedings of the Fourteenth Workshop on the Algorithmic Foundations of Robotics 14*, pp. 313–329. Springer, 2021.

Minae Kwon, Sang Michael Xie, Kalesha Bullard, and Dorsa Sadigh. Reward design with language models. *arXiv preprint arXiv:2303.00001*, 2023.

Kimin Lee, Laura Smith, and Pieter Abbeel. Pebble: Feedback-efficient interactive reinforcement learning via relabeling experience and unsupervised pre-training. *arXiv preprint arXiv:2106.05091*, 2021.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9493–9500. IEEE, 2023.

Zeyi Liu, Arpit Bahety, and Shuran Song. Reflect: Summarizing robot experiences for failure explanation and correction. *arXiv preprint arXiv:2306.15724*, 2023.

R.D. Luce. *Individual Choice Behavior: A Theoretical Analysis*. Wiley, 1959. URL https://books.google.com.sg/books?id=c519AAAAMAAJ.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023.

Denys Makoviichuk and Viktor Makoviychuk. rl-games: A high-performance framework for reinforcement learning. https://github.com/Denys88/rl_games, May 2021.

Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, et al. Isaac gym: High performance gpu-based physics simulation for robot learning. *arXiv preprint arXiv:2108.10470*, 2021.

Ian R Manchester, Uwe Mettin, Fumiya Iida, and Russ Tedrake. Stable dynamic walking over uneven terrain. *The International Journal of Robotics Research*, 30(3):265–279, 2011.

Andrew Y Ng, Stuart Russell, et al. Algorithms for inverse reinforcement learning. In *Icml*, volume 1, pp. 2, 2000.

Malayandi Palan, Nicholas C Landolfi, Gleb Shevchuk, and Dorsa Sadigh. Learning reward functions by integrating human demonstrations and preferences. *arXiv preprint arXiv:1906.08928*, 2019.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html.

Dorsa Sadigh, Anca D Dragan, Shankar Sastry, and Sanjit A Seshia. *Active preference-based learning of reward functions*. 2017.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 11523–11530. IEEE, 2023.

Giorgio Valsecchi, Ruben Grandia, and Marco Hutter. Quadrupedal locomotion on uneven terrain with sensorized feet. *IEEE Robotics and Automation Letters*, 5(2):1548–1555, 2020.

Tianbao Xie, Siheng Zhao, Chen Henry Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. Text2reward: Automated dense reward function generation for reinforcement learning. *arXiv preprint arXiv:2309.11489*, 2023.

Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, et al. Language to rewards for robotic skill synthesis. *arXiv preprint arXiv:2306.08647*, 2023.