A Simple Framework for Intrinsic Reward-Shaping for RL using LLM Feedback

Alex Zhang[†], **Ananya Parashar**[♢], **Dwaipayan Saha**[†]
† Department of Computer Science, Princeton University ♢Department of ORFE, Princeton University
{alzhang, parashar, dsaha}@princeton.edu

Abstract

The credit assignment problem is a long-standing issue in reinforcement learning (RL) applications. In complex settings with sparse reward signals and large state and action spaces, an agent naively exploring an environment without any prior knowledge requires an enormous amount of training data that is computationally intractable. One method for alleviating this issue is reward shaping, where intrinsic rewards are provided to an agent at intermediate states based on prior knowledge known by an expert. Large language models (LLMs) are known to contain a wealth of domain knowledge that can be used as a prior for many environments and can be harnessed for reward shaping. In this work, we exploit the code generation capabilities, reasoning, and domain knowledge of LLMs to iteratively generate and refine intrinsic reward functions on Pokemon Showdown, a multi-agent strategy game featuring complex interactions and stochasticity. We propose three simple methods of reward shaping using LLMs for iteratively generating reward functions during deep RL training. We first demonstrate our simple reward shaping framework on an application of Q-learning to the OpenAI MountainCar-v0 environment, then evaluate our reward shaping methods against each other as well as against other deep RL baselines on Pokemon Showdown, a stochastic 1-v-1 strategy game with readily available strategies and game information on the web. We demonstrate that under a fixed number of training iterations, reward shaping with LLMs can achieve higher sample efficiency by injecting its domain knowledge to reduce the agent's exploration of redundant states, with all three of our proposed methods achieving a $> 80\%$ win-rate over the base Deep Q Network model. Furthermore, the intrinsic reward functions generated by the LLMs are interpretable and easy to understand based on domain knowledge of the game. We publicly release our code[1].

1 Introduction

Modern reinforcement learning (RL) methods suffer from sample inefficiency issues during training, often requiring astronomical amounts of trajectory data to converge. One large factor in this issue is the credit assignment problem, where it is difficult to naively explore environments with combinatorially large state spaces and properly attribute rewards to "good" states. This issue is exacerbated in environments with sparse rewards, where a reward is generally only awarded for completing a single objective. Intuitively, most arbitrary game settings have a **single** "goal" condition (i.e. win the game), in which reward functions are generally sparse. A potential solution is to add an additional *intrinsic* reward at every step that is defined by some human expert with known prior about the environment to reduce the space of "redundant" states that the agent explores. However, prior works have shown that manual trial-and-error reward design and expert-designed reward functions often lead to sub-optimal learned behaviors [BKS+23]. For example, the famous "move 37" made by AlphaGo [SHM+] was

[1] https://github.com/alexzhang13/reward-shaping-rl

initially deemed unpredictable by any human expert, and any dense rewarding scheme defined by a human expert would have most likely never led to this move being made.

For complex domains such as Minecraft [FWJ+22], NetHack [KNM+20], and Pokemon [Sah23], it is difficult to provide "provable" guarantees for effective reward shaping in terms of sample efficiency. Furthermore, most domain knowledge for these environments is well-described in text through sources such as wikis, forums, and other player-curated content. In recent history, large language models (LLMs) have proven to be effective databases and generators of text content, with hundreds of new works showcasing their effectiveness in navigating text domains. Eureka [MLW+23] builds an explicit, highly-scalable framework for distributed training of RL agents using LLMs to generate rewards by generating reward functions in Python. However, their work focuses on low-level control tasks, and they explicitly use LLMs for their code generation abilities and self-reflection capabilities.

We build on the [MLW+23] work, but focus mainly on providing a simple framework for reward shaping by generating code that is applicable to any environment. In this work, we do the following:

1. **Create a simple, barebones framework for using LLMs to generate intrinsic reward functions** for RL agents and refine these reward functions after each episode. We develop this framework to be extremely easy to integrate into any RL application, even without distributed forms of training.
2. Propose **three simple methods for determining how reward-shaping feedback is used to train popular reinforcement learning algorithms** like tabular Q-learning, deep Q-learning [MKS+13], and proximal policy optimization (PPO) [SWD+17].
3. **Evaluate our method on simple gym-retro environments and Pokemon Showdown** (an unsolved game) against tabula rasa deep RL methods and heuristic methods to demonstrate that prior knowledge from LLMs are useful.

We focus our LLM-based reward shaping specifically on Pokemon Showdown, as there exists a plethora of online information about strategies and the game that are shown to be in the training data of many LLMs. The intuitive idea is that LLMs contain knowledge about well-known domains that are well-described in text but often difficult to apply in a rule-based framework **due to the inherent difficulty of grounding language to decisions**. In the case of reward shaping by generating reward functions using LLMs, the code-generated reward function acts as the implicit grounding between language and a decision-making agent. Thus, we are interested in understanding whether the reasoning capabilities and knowledge of an LLM can be leveraged to guide the learning of RL algorithms in a more sample efficient manner.

2 Background and Related Works

Building learnable agents using reinforcement learning has proven to be difficult in environments with intractably large or abstract state and action spaces [OB+19]. This issue is apparent in most deep RL algorithms that assume no prior knowledge about the environment, as they have to explore an environment while visiting and trying redundant or "bad" actions/states. However, with the abundance of human-knowledge encoded in language, a large question is whether these resources can be leveraged to accelerate the learning process of RL methods.

2.1 Language Guided Reinforcement Learning

Injecting language information to influence a reinforcement learning algorithm is not a well-understood field, but several prior approaches have been tried. For example, a class of RL methods called "**goal-conditioned reinforcement learning**" [LZZ22] have grown in popularity in the past few years. The general concept is to condition the behavior of the RL agent on both the state observations and a specific "goal", which can be made a language goal (e.g. find the door). In SayCAN[ABB+22], they demonstrate that given language captions of expert trajectories, they can scalably train robots to perform actions conditioned on language. The benefit of these methods is that they leverage the reasoning capabilities of LLMs to break down a complex instruction into sentences that their goal-conditioned agents have been trained on. However, these methods inherently rely on the existence of labelled expert trajectories, which is often not readily available.

Another approach to language-guided RL is to leverage language in **model-based reinforcement learning** methods, which rely on a model of the environment to learn an optimal policy. It is known and quite intuitive [SB18] that knowing the transition probabilities of an environment can provide provably faster convergence, so it has become an area of interest to leverage these methods by jointly learning a model of an environment while optimizing a policy. Because of the abundance of visual environments, methods like CLIP [RKH+21] provide a way to leverage language information into the learning of a world model [HS18] that can be used for model-based RL. For example, in LanGWM[PPZC23], they learn a vision-language embedded representation of the world model and build on top of [HLNB22] to achieve SOTA results on 3D environments.

Finally, a set of methods have recently arisen that **try and influence the training process of an RL algorithm by modifying the reward function**. In Motif [KDS+23], they train a reward model using an annotated dataset of language captions on Nethack [KNM+20] and use this "instrinsic" reward model to influence the learning of an RL agent. Moreover, in [YGF+23], they show that the reward model itself can be generated entirely in code by an LLM and they apply this reward shaping method with specific prompting techniques to robotics control tasks. Finally, in Eureka [MLW+23], the authors build a distributed training framework for reward shaping with LLMs (GPT-4) to solve a suite of low-level control tasks for simulated robotics motion. They also propose a set of methods that reflect on and "evolve" their reward functions as a kind of meta-learner. Our work is heavily inspired by this suite of works, and we aim to develop a simple framework on a more complex task where language information and reasoning capabilities can be leveraged better (specifically in Pokemon Showdown).

2.2 Reward Shaping

Many environments contain a single true sparse reward that signals that the agent has won or lost. In these settings, the credit assignment problem becomes exacerbated due to the combinatorially large state spaces that an agent has to traverse to reach any kind of reward. Reward shaping is a way of "injecting" pseudo-rewards at intermediate states based on a prior understanding of what is "good" to do in the environment. This thus accelerates the exploration process by encouraging the agent to avoid exploring states that are known to be "bad." This process is more formally described below in the Q-learning framework.

In traditional Q-learning, we define the Q-function which as the expected discounted sum of rewards by choosing action a from state s. That is for actions a and a' and states s and s' as well as reward r and discount factor γ, we have:

$$Q^*(s,a) = \mathbb{E}[r + \gamma \max_{a'} Q^*(s',a')] \tag{1}$$

Then the iterative update is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \tag{2}$$

for learning rate α. In this update, we may intuitively think of this as improving the Q-function by updating with the prior value of the Q-function added to the the temporal difference (TD) error scaled by the learning rate. This error is attained by summing the immediate reward gained (r) to the discounted optimal Q-value of the next state to get the TD target then subtracting the prior value of the Q-function.

In the setting of our paper, as we mentioned before, we do not modify the received reward r, rather we provide *additional shaped reward* for specific transitions[2]:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \underbrace{F(s,s')}_{\text{additional reward}} + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

The function F typically reflects domain knowledge which is manually encoded and $r + F(s,s')$ is the *shaped reward* for an action. Further, we denote $G^\pi = \sum_{i=0}^{\infty} \gamma^i (r_i + F(s_i, s_{i+1}))$ as the shaped reward for the entire episode. We may see that for the shaped reward:

- $F(s,s') > 0$ provides *positive* reward for transitioning from states s to s'

[2] `https://gibberblot.github.io/rl-notes/single-agent/reward-shaping.html`

- $F(s, s') < 0$ provides a small *negative* reward for transitioning from states s to s'

Thus this shaped reward either encourages or discourages such actions in future exploitation, respectively. Works such as [KDS+23] learn $F(s, s')$ through explicit supervised training. Our work is most similar to [YGF+23],[MLW+23], which iteratively generates $F_t(s, s')$ using an LLM.

2.3 LLM for Code Generation

Generating reward functions using LLMs is done through code generation because code is the "language" understood by the environment. Furthermore, code generation and code synthesis are active areas of research in the NLP community that are constantly improved on. There are several examples of explicitly fine-tuned models [RGG+23], [ALK+23], [NHX+23] for coding and also examples of base LLMs like GPT-4 being capable of coding zero-shot [ZW23].

Similar to [MLW+23], we focus on code generation of reward functions that is iterated on as the training process continues. The initial generated reward function will most likely be faulty or too simple, so we focus on refinement through generic iterative code refinement techniques used for LLM code generation. Furthermore, while [MLW+23] focuses on distributed RL training, we simplify the setting and the framework's computational overhead by focusing on a single RL agent.

3 Method

As briefly mentioned in our introduction, our proposed method is to use LLMs, primarily GPT-3.5 or GPT-4, in order to create reward functions. Inevitably, harnessing domain knowledge requires explicit prompting (e.g. identifying the domain to the LLM), but we do not focus on prompt-tuning in this work and instead focus on the reward function refining process. We do this through three different methods: sequential feedback, tree based feedback, and moving target feedback as depicted in Figure 1.

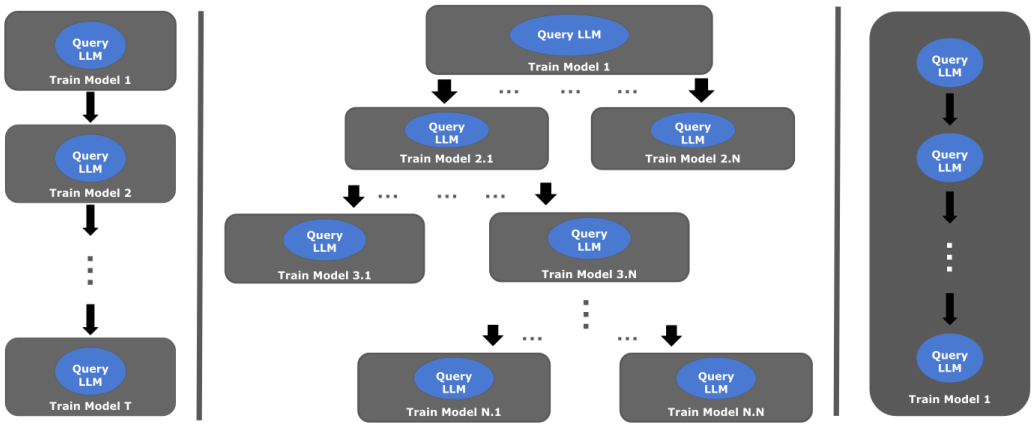

Figure 1: Left: Sequential Feedback, Center: Tree Based Feedback, Right: Moving Target Feedback

3.1 Method 1: Sequential Feedback

The simplest form of reward shaping is to start with a randomly initialized model and query the model to generate an intrinsic reward function $F_0(s, s')$ (in code) based on its prior knowledge about the environment. Then, after training the model for a fixed number of episodes, we sample a set of recent trajectories/transitions and query the LLM to generate a new intrinsic reward function $F_1(s, s')$ based on its prior knowledge about the environment and these transitions. We then train a model on this new reward function. Effectively, we sequentially generate a series of intrinsic reward functions $F_0(s, s'), F_1(s, s'), ..., F_M(s, s')$. Furthermore, we impose a Markovian assumption on how the reward function evolves, thus allowing us to develop a simple algorithm as in Algorithm 1.

Algorithm 1: Sequential Reward Function Generation

Data: Number of meta-optimization steps T, Training length L
Result: M_T
begin
  Query LLM to compute $F_0(s, s')$ for all $(s, s') \in \mathcal{C}$;
  Train model M_0 using shaped rewards using $F_0(s, s')$;
  while $t \leq T$ **do**
    Prompt LLM to get the generative distribution $\mathbf{P}(F_t(s, s')|F_{t-1}(s, s'))$;
    Sample $F_t(\cdot)$ from $\mathbf{P}(F_t(s, s')|F_{t-1}(s, s'))$ by querying the LLM;
    Train model M_t using $F_t(s, s')$ for L iterations;
  end
end

3.2 Method 2: Tree-based Feedback

The evolutionary search algorithm for LLM-based reward shaping proposed in [MLW+23] is designed for highly-distributed reinforcement learning training algorithms in order to iteratively produce and refine reward functions. We propose a similar, tree-based search algorithm on a simpler scale that builds on top of Sequential Feedback method but designed for the purpose of Pokemon Showdown.

Outlining the basic iterations of this optimization method, we first start off by sampling independently and identically distributed reward outputs from the LLM. Then, we simply take the best-performing reward out of these as context and use the LLM to generate K more of these independently and identically distributed reward outputs and keep iteratively performing this process for a certain number of iterations. The algorithm looks as in Algorithm 2.

Algorithm 2: Tree-based Reward Function Generation

Data: Number of meta-optimization steps T, Training length L
Result: M_T
begin
  Query LLM to compute $F_0(s, s')$ for all $(s, s') \in \mathcal{C}$;
  Train model M_0 using shaped rewards using $F_0(s, s')$;
  while $t \leq T$ **do**
    Prompt LLM to get the generative distribution $\mathbf{P}(F_t(s, s')|F^*_{t-1}(s, s'))$;
    Sample K reward functions $F^i_t(\cdot)$ for L iterations where $i \in [K]$ from $\mathbf{P}(F_t(s, s')|F^*_{t-1}(s, s'))$;
    Train a model M^i_t using $F^i_t(s, s')$ for each $i \in [K]$, generating K models $\{M^i_t\}_{i=1}^K$;
    Save reward-maximizing $M^*_t = \underset{j \in [K]}{\text{argmax}}\ M^j_t$ and the corresponding $F^*_t(\cdot) = F_t(\cdot)$;
  end
end

3.3 Method 3: Moving Target Feedback

Until now, our techniques involve training *new* models with updated reward functions. However, there is motivation behind iteratively updating the reward function via similar queries to the large language model yet still retaining the parameter weights from the previous stage of training. This instills a retention scheme in order to successively improve the objective, while still taking advantage of the domain knowledge of the large language model and also avoiding drastic difference amongst successive models. Overall, the moving target feedback scheme leads us to train the model episodically, but the *shaped reward function* gets updated over time. Thus, we may write this algorithm as represented in Algorithm 3, where we iterate until the algorithm converges.

As mentioned in the next section, we utilize a variety of common reinforcement learning training algorithms and variants primarily based off of Proximal Policy Optimization and Deep Q-Networks. The aforementioned algorithms are a way to exploit the domain knowledge of the large language

Algorithm 3: Moving Target Reward Generation

Data: Training length L per fixed reward
Result: M_T
begin
  Query LLM to compute $F_0(s, s')$ for all (s, s');
  Train model M using shaped rewards using $F_0(s, s')$ for L iterations;
  while *not converged* **do**
    Query LLM to generate the distribution $\mathbf{P}(F_t(s, s')|F_{t-1}(s, s'))$;
    Sample $F_t(\cdot)$ from $\mathbf{P}(F_t(s, s')|F_{t-1}(s, s'))$;
    Continue training the model M using $F_t(s, s')$ for L iterations;
  end
end

model, but also be able to iteratively update the knowledge using the sequential feedback received on the reward function from the previous iteration.

4 Environment

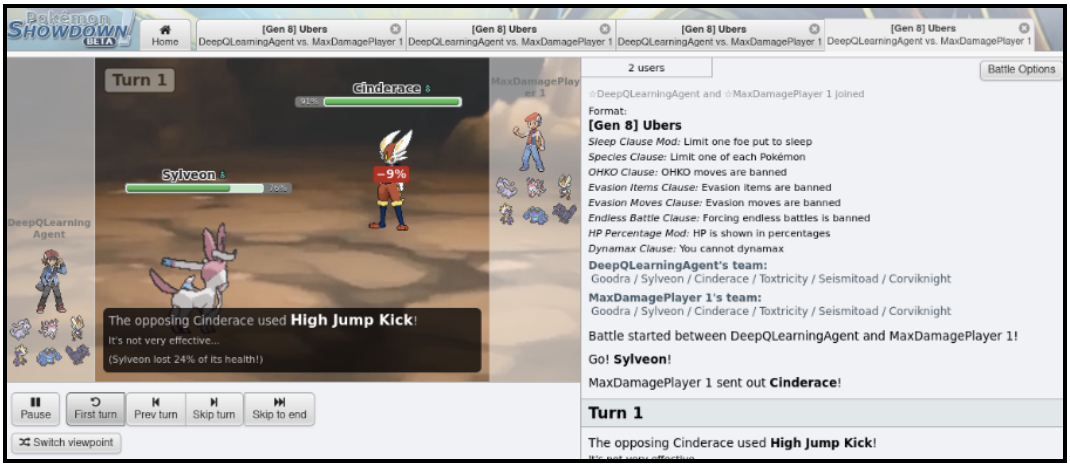

Figure 2: An example of a Deep Q-learning agent playing Pokemon Showdown against a bot.

Pokemon Showdown is a turn-based 1-v-1 battle game in which each player uses a team of up to six Pokemon and tries to defeat the entirety of the opponent's team of Pokemon. During each turn, both players simultaneously make a decision, and the game plays out based on these decisions. The game is highly complex and well-studied, featuring over 800 unique Pokemon, each with their own unique abilities, attacks, and interactions with one another. The game is stochastic and the action space is quite small (each Pokemon can only use up to four unique attacks in a single battle), making the game especially interesting for reinforcement learning. The game is also adversarial, as optimal actions are entirely determined by your opponent's decision, which are unknown to the player. Furthermore, there exists online forums such as Smogon[3] dedicated to building the optimal teams and strategies for competitive play, as well as official competitions (Pokemon VGC World Championships) that are still active to this day. This fact makes the game especially interesting for language-guided RL, since there is an abundance of online strategies, game data, and tutorials for playing the game.

There are several different game-modes, but we focus on **fixed-team, single-battles**, where both players use the same team and only a single Pokemon per-player can be active at once. For the purposes of this setting, the main points of interest are **type-matchups** and the **statistics** of each Pokemon. In Pokemon Showdown, each Pokemon has up to two *types* (e.g. water, grass, fire, ice, etc.), which determine how effective an opponents attack is on them (for example, water is strong

[3] `https://www.smogon.com/`

against fire). Furthermore, each Pokemon has access to up to four unique attacks with their own associated typing, and every turn each player can choose to use one of these attacks or use a turn switching out the currently active Pokemon with one of the others. Additionally, each Pokemon has **six main statistics**: health, attack, special attack, defense, special defense, and speed.

A Pokemon "faints" when its health reaches 0, at which point it becomes unusable in battle. Speed is used to determine which Pokemon attacks first in a turn (unless they are switched out), and the other statistics determine how powerful an attack is. Furthermore, there are certain attacks that raise these statistics in battle, which can be used strategically to set up win games. These interactions are hard to capture well through naive exploration, but are well understood by players online and can potentially be distilled into an RL agent through language.

We train our agent on poke-env (`https://github.com/hsahovic/poke-env`), which provides a gym-interface to Showdown. We also run and train all our experiments on a local server, which allows us to run multiple games in parallel.

5 Experiments

5.1 Baselines

We focus on applying reward shaping to two popular deep RL algorithms, PPO [SWD+17] and deep Q-learning [MKS+13].

5.1.1 Proximal Policy Optimization

Algorithm 4: PPO-Clip

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, \ldots$ **do**
Collect set of trajectories $D_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
Compute rewards-to-go $\hat{R}_l$. Compute advantage estimates, $\hat{A}_l$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}. Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|D_k|L} \sum_{\tau \in D_k} \sum_{l=0}^{L} \min\left(\frac{\pi_\theta(a_l|s_l)}{\pi_{\theta_k}(a_l|s_l)} A^{\pi_{\theta_k}}(s_l, a_l),\ g(\epsilon, A^{\pi_{\theta_k}}(s_l, a_l)) \right),$$

typically via stochastic gradient ascent with Adam. Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|D_k|L} \sum_{\tau \in D_k} \sum_{l=0}^{L} \left(V_\phi(s_l) - \hat{R}_l \right)^2,$$

where we redefine the rewards-to-go function using shaped rewards:

$$\hat{R}_l = \sum_{i=l}^{L} R(s_i, a_i, s_{i+1}) + F_t(s_i, s_{i+1})$$

typically via some gradient descent algorithm.
end

Proximal Policy Optimization [SWD+17] is descendant of a Trust Region Policy Optimization [SLM+17], which uses a suite of first order approximations instead of complex second order techniques. We use PPO-clip, the updates for which are characterized as,

$$\theta_{k+1} = \arg\max_{\theta} \mathbb{E}_{s,a \sim \pi_{\theta_k}} \left[L(s, a, \theta_k, \theta) \right] \tag{3}$$

where the loss is defined as

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1 - \epsilon, 1 + \epsilon \right) A^{\pi_{\theta_k}}(s, a) \right) \tag{4}$$

Here ϵ controls how far new policy can move from the previous step to **avoid** having updates that are too large. Further, $A^{\pi_{\theta_k}}$ is defined as the estimated advantage at update step k which are computed from the current value function estimate and environment rewards. We inject our reward shaping technique into the aforementioned rewards to instead use the *shaped reward*. Our implementation is built upon the reference provided by OpenAI [SWD+17], which also provides the pseudocode in Algorithm 4 that we modify to incorporate the additional shaped rewards $F(s, s')$ [4]:

5.1.2 Deep Q-Networks

Algorithm 5: Deep Q-learning with Experience Replay and Reward Shaping

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for $episode = 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s' = s, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s')$
    Store transition $(\phi_t, a_t, r_t + F_t(s, s'), \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j + F_j(s, s'), \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j + F_j(s, s') & \text{for terminal } \phi_{j+1} \\ r_j + F_j(s, s') + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end
end

We introduced the traditional Q-learning method in the prior section as a model-free reinforcement learning algorithm, specifically looking at the intuition behind the update in Equation 2 as well as adding the shaped reward here. We also defined the optimal Q-function Q^* (defined using the Bellman optimality equation in Equation 1). During the implementation of the Q-learning algorithm, we represent this function as a Q-table which stores the expected reward of each state-action pair and iteratively updates as the agents continues exploring. However, when the number of actions and states grows, storing each state-action pair is expensive memory-wise and it also becomes more unlikely that we explore every state to create the Q-table in a reasonable amount of time. Thus for large environments, we use Deep Q-Learning ([MKS+13]), replacing the Q-table representation of the Q-function with a neural network with parameters θ to approximate the Q-function, that is $Q(s, a; \theta) \approx Q^*(s, a)$. We compute this by minimizing our loss, the squared sum of temporal difference errors, at each step i:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim \rho(.)} \left[(y_i - Q(s, a; \theta_i))^2 \right]$$

Here, y_i is the TD target, formally written as:

$$y_i = r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})$$

Moreover, ρ is the information learned from the environment of the distribution of {s, a, r, s'} where r is the reward gained by taking action a from state s.

The concept of experience replay, as introduced in [MKS+13], involves pooling every agent's experiences at each time-step denoted by $e_t = (s_t, a_t, r_t, s_{t+1})$ across episodes. These experiences are then stored in a circular buffer called the replay buffer. This allows us to sample random mini-batches from the replay buffer to train the network, rather than solely relying on the latest experience. This elucidates one of the advantages of this method – variance reduction as a result of random sampling. Moreover, we enhance data efficiency by as potentially experiences could be reused in multiple weight updates.

[4] Note that within the algorithm in the PPO-objective we use $g(\cdot)$ which is a simplified version of Equation 4, as suggested in [SWD+17].

Now, we use our reward-shaping technique to modify the rewards defined for Deep Q-Learning. We use pseudo-code for Deep Q-Learning with experience replay from [MKS+13] and modify it to include our reward shaping by adding the function $F_i(s, s')$ to our original reward r_i (where $s' = s_{t+1}$ and $s = s_t$) as displayed in Algorithm 5.

5.2 Simple Example: MountainCar-v0

Our most simple example comes from OpenAI's Gym Environment called "Mountain Car" as in [Far23]. In this Markov Decision Process, there are two hills, with the mountain car placed stochastically at the bottom of the valley between them. The objective of the mountain car is to reach the flag on the top of the right hill (x position of 0.5) in the least amount of time possible by controlling its acceleration. The mountain car begins with zero velocity at some point between the mountains (between -0.6 and -0.4 where the x axis is defined along -1.2 to 0.6). As such, the observation space for the mountain car is two dimensional consisting of the position of the mountain car along the x-axis and velocity of the mountain car at each time step. Additionally, there are three possibilities for the mountain car's acceleration: accelerate right, accelerate left, and don't accelerate. Moreover, rewards are negative for each time step, incentivizing the mountain car to reach the flag faster. We terminate if we obtain this goal or if we reach the maximum length of the episode of 200.

We make queries to GPT-3.5 Turbo in order to produce reward functions while training. The intrinsic reward function GPT-3.5 Turbo produced is given in Figure 3 and results are given in Figure 4, where we see that the implementation of Q-learning with the additional LLM feedback given by generated reward function code outperforms the simple Q-learning with sparse reward.

Figure 3: Intrinsic reward function by `GPT3.5 Turbo`:

```
def reward(x_position, velocity, action):
    if x_position >= 0.5 and action == 2:
        # At right-most hill, accelerate right
        return 0.1  # Return reward
    elif x_position <= 0 and action == 0:
        # In valley, accelerate left
        return 0.1
# Return a reward
    elif x_position <= -0.5 and action == 2:
        # At left-most hill, accelerate right
        return 0.1  # Return reward
    else:
        return 0
```

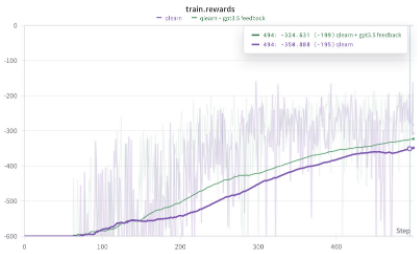

Figure 4: Performance of Q-learning with sparse reward vs. Q-learning with additional LLM feedback as generated reward function code.

5.3 Pokemon Showdown Battles

5.3.1 Input Representation

We create a vector representation of the Pokemon Showdown environment that is used as input to each model. In this setting, we ignore the effects of abilities and items since every training episode uses the same team, so effectively the abilities and items are unique to the Pokemon that hold them. Thus, to represent the current state, we provide the available moves $m_1, ..., m_4$ indexed by a unique identity, and attributes about each of the agent's Pokemon $\mathcal{P}_1^p, ..., \mathcal{P}_6^p$ and the opponent's Pokemon $\mathcal{P}_1^o, ..., \mathcal{P}_6^o$. Thus, the current observation is

$$O_t = (m_1, m_2, m_3, m_4, \mathcal{P}_1^p, \mathcal{P}_2^p, \mathcal{P}_3^p, \mathcal{P}_4^p, \mathcal{P}_5^p, \mathcal{P}_6^p, \mathcal{P}_1^o, \mathcal{P}_2^o, \mathcal{P}_3^o, \mathcal{P}_4^o, \mathcal{P}_5^o, \mathcal{P}_6^o) \in \mathbb{R}^{58}$$

where each Pokemon $\mathcal{P}_i$ is

$$\mathcal{P}_i = (\texttt{is\_active}, \texttt{is\_fainted}, \texttt{status}, \texttt{HP}, \texttt{attack}, \texttt{defense}, \texttt{sp.att}, \texttt{sp.def}, \texttt{speed})$$

5.3.2 Cross-evaluation Performance

We evaluate the performance of each baseline method, as well as the three methods described in Section 3. Each model was trained for the same total number of episodes (10000), and their performance against eachother is shown in Table 1. For reward shaping method 1 (M1), we train each

sequential for 1000 episodes and query for an intrinsic reward function 10 times. For reward shaping method 2 (M2), we train each episode for 500 episodes, then query for 5 reward functions. We then pick the best intrinsic reward, and condition on the previous reward function. We repeat this process 4 times. Finally, for reward shaping method 3, we update the reward function every 1000 episodes without resetting the model weights.

Table 1: We had each of our methods cross-evaluated against eachother over a total of 50 games. The win-rate of each method (row wins vs. column) is listed below.

Models	DQN	DQN+M1	DQN+M2	DQN+M3	PPO	PPO+M1	PPO+M2	PPO+M3
DQN	-	0.16	0.08	0.14	0.46	0.6	0.14	0.18
DQN+M1	0.84	-	0.9	0.88	0.6	0.56	0.06	0.12
DQN+M2	0.92	0.1	-	0.22	0.14	0.6	0.36	0.24
DQN+M3	0.86	0.12	0.78	-	0.54	0.64	0.54	0.52
PPO	0.54	0.44	0.4	0.32	-	0.26	0.16	0.18
PPO+M1	0.4	0.44	0.4	0.36	0.74	-	0.3	0.22
PPO+M2	0.86	0.94	0.64	0.46	0.84	0.7	-	0.38
PPO+M3	0.82	0.88	0.76	0.48	0.82	0.78	0.62	-

There are some interesting results to note here, namely that reward shaping for these models performs quite well **compared to the base model on a limited number of steps**. These results are not conclusive to show that reward shaping with LLMs offers a clear boost in sample efficiency, as for PPO and DQN, we did not take advantage tricks like asynchronous updates and proper reward clipping often used for these methods. However, they offer insight the usefulness of using domain knowledge from an LLM to influence the shaping of a reward function. For example, consider the final shaped reward function generated by the end of applying Method 2 to DQN in Figure 5. By prompting the model to utilize its understanding of Pokemon, it is able to understand that an important subgoal in this game is to defeat the enemy's individual Pokemon and deplete their health and train the model to maximize these subgoals.

Figure 5: Intrinsic reward function generated by Method 2. Intuitively, the LLM is rewarding the agent for defeating an enemy Pokemon and dealing as much damage as possible in the current transition.

```
def reward(prev_battle_state, next_battle_state):
    prev_fainted = [mon for mon in prev_battle_state.opponent_team.values() if mon.fainted]
    next_fainted = [mon for mon in next_battle_state.opponent_team.values() if mon.fainted]

    prev_total_hp = sum([mon.current_hp for mon in prev_battle_state.opponent_team.values()])
    next_total_hp = sum([mon.current_hp for mon in next_battle_state.opponent_team.values()])

    reward_defeat = len(next_fainted) - len(prev_fainted)
    reward_damage = (prev_total_hp - next_total_hp) / 1000.0

    return reward_defeat + reward_damage
```

5.3.3 Comparison against Heuristic Algorithms

As mentioned earlier, Pokemon Showdown is heavily based on the used strategy, so better performance against a certain strategy A that consistently beats a strategy B does not transitively imply better performance against B. Furthermore, since the table in Table 1 was cross-evaluated, it is unclear how good these methods are against other as general strategies. There are several basic heuristic algorithms in this setting that are easy to understand for a human, and are also easy to replicate with rule-based mechanisms. We evaluate our methods against these heuristic algorithms. These heuristic algorithms are specifically:

1. `Random`: The agent chooses a completely random action, which includes wasting a turn switching out your active Pokemon.
2. `Random Attack`: The agent always chooses to attack by selecting a random move from the currently active Pokemon.

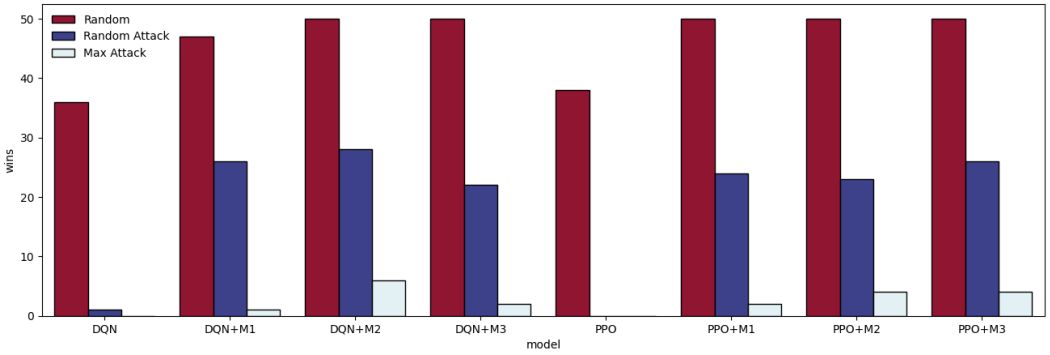

Figure 6: Comparison of our proposed methods and baselines against known Pokemon Showdown heuristic methods. All methods perform well against a random baseline, but struggle against the damage-calculating heuristic.

3. `Max Attack`: The agent computes the attack that deals the maximum amount of damage to the opponent's active Pokemon based on damage multipliers, typing, and attack base power, and will always choose this move.

Notably, all of these strategies are easy to play against adversarially, but we do not expect the agent to learn how to combat the hardest strategy since it has no access to damage calculations or type interactions, and therefore must rely on the LLM to properly generate a reward function that reflects these interactions.

From Figure 6, we see that generally our agents perform on par with an agent that attacks a lot. Within 10000 episodes of a sparse reward and reward shaping, it learns a "local" minima of choosing attacking/damaging moves, but none of the methods fare well against a damage-calculating heuristic.

6 Future Work and Limitations

The goal of this project was to demonstrate the usability of LLM-based reward shaping as a form of language-guided RL in simple use-cases. There are several important limitations to address in this work that are largely addressed by scaling both the environment and the training. We detail them below.

6.1 Unwanted priors introduced by our embedding representation

The input representation given to our models is a relatively low-dimensional numerical embedding representation that captures information about the game in a fixed manner. This representation makes RL training computationally cheap, allowing the experiments to run on a single RTX3080Ti GPU. However, the most generic form of input is to compress the text and numbers provided by the Pokemon Showdown environment using a language model encoder and then to train a large model over this high-dimensional input space. The learning process would converge significantly slower in this way, but it also removes any biases provided by hand-selecting input features that would influence the effect of reward-shaping using an LLM. Given more computational resources, this approach would better highlight the effects of LLM-based reward shaping.

6.2 Expanding the difficulty of the full environment

The goal of this work is to demonstrate the application of LLM-based reward shaping on an arbitrary but complex game environment. The primary benefits of using an LLM to generate an intrinsic reward function come from the prior knowledge about game environments that LLMs contain and their ability to easily iterate and inference at scale (relative to a human expert). It would be ideal to evaluate this approach on a fully randomized version of the game where the agent potentially sees a team, Pokemon, or strategy that it has never seen in its training distribution. However, observing these "generalization" benefits requires a large amount of pre-training regardless of whether or

not an "optimal" intrinsic-reward function is created, so it was not within the scope of this work. Nevertheless, there is room for future work to expand this training to the full Pokemon Showdown game.

6.3 Playing against real opponents

A large appeal of Pokemon Showdown for reward shaping is the adversarial nature of the game and the aspect of "reading your opponent," making it impossible to develop an "optimal" bot that consistently beats any player. However, because of the combinatorially large number of teams and strategies used in online games, it was infeasible for this work to try to play against real opponents. In the real setting, the Pokemon names, abilities, items, effort values (EVs), etc. all matter, and the interactions between these statistics have to be learned by a sufficiently large RL model. We did attempt to deploy our bots on the online ladder to see if any meaningful results would arise, but we found that the "wins" were generally attributed to opponents disconnecting or purposely losing, making the results unmeaningful.

6.4 More extensive baseline training

Our current baseline methods are vanilla implementations of PPO and DQN. However, most deep RL algorithms are hand-tuned for a specific domain, which we did not do in our experiments. Furthermore, using other state of the art training mechanisms will be helpful in extrapolating the true benefit of using reward shaping as mechanism to learn better policies whilst being significantly more sample efficient and reaching convergence quicker.

6.5 Conditioning on learning history for reward shaping and to context window limitations

In all of the proposed methods, there exists the notion of sampling the reward function at step t notated as $F_t(s, s')$, from the generative probability distribution $\mathbf{P}(F_t(s, s')|F_{t-1}(s, s'))$. Specifically, at each step, we train a model using the shaped reward function and evaluate the true associated reward. More formally, one can think of the true reward R_t at each step t as a $(\mathbb{R}, \mathcal{B}(\mathbb{R}))$ valued random variable and the reward trajectory as the following stochastic process $\{R_t\}_{0 \leq t \leq T}$. Further, we condition upon $F_{t-1}(s, s')$, which means that, formally, we condition upon the sigma algebra generated by the events up to time step $t-1$, i.e. $\sigma(R_0, R_1, \ldots, R_{t-1})$. In simpler terms, we are encoding the information gained from training the model with the sampled reward function at steps $0, \ldots, t-1$ as well as the corresponding performance in terms of true reward, which is generated by training with the shaped reward.

However, there is a caveat. It is known that even the most advanced language models such as GPT-4 have a limited context window. As a result, it is not often possible to condition on the entire history. As a remedy, we propose truncating to most recent history that fits within the allowed context window. The motivation behind this is that the model is iteratively making small improvement steps and the most recent history is the most relevant, instead of implementing remedies such as random sampling along the trajectory.

6.6 LLM reward function code generation errors

In order to ensure the correct execution of our tasks by the LLM, our prompting to the LLM must be *precise*. We must specify multiple constraints to the LLM, outside of the immediate prompt of reward shaping, which may seem obvious. That is, we must specify to the LLM to not generate any other text other than the function (i.e. not respond to the prompt, rather simply provide code), to not execute the code that the LLM outputs, etc. Such comprehensive and precise instruction is essential to ensuring that the code generation is as intended and produces the desired behaviors – if this is not given, there are multiple compiler errors.

6.7 Reproducibility

We noticed some inconsistencies in how the LLM generates the reward function. Method 2 is the most consistent due to generating multiple reward functions, but often our reward functions generated by the LLM were faulty (compiler errors, syntax errors, incorrect API calls, etc.) that required extra

manual engineering (e.g. restart from that checkpoint) to get working. We are looking into how coding LLM approaches like [LCC+22] to provide a more robust framework for reward generation.

7 Conclusion

We have presented a simple framework for reward shaping with LLMs that works for non-parallelized algorithms. We demonstrate that this method can achieve sample efficiency gains over naive exploration by leveraging domain knowledge in LLMs to shape the reward function. We show that this method can work on Pokemon Showdown, which contains an abundance of available domain knowledge and strategies online that were most likely present in the LLM's training corpus.

8 Acknowledgements

We would like to thank Professor Sanjeev Arora for his kind feedback and suggestions, alongside an engaging semester learning deep learning theory. Furthermore, we would like to thank Yikai Wu for his insightful conversations.

References

[ABB+22] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as i can and not as i say: Grounding language in robotic affordances. In *arXiv preprint arXiv:2204.01691*, 2022.

[ALK+23] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*, 2023.

[BKS+23] Serena Booth, W. Bradley Knox, Julie Shah, Scott Niekum, Peter Stone, and Alessandro Allievi. The perils of trial-and-error reward design: Misdesign through overfitting and invalid task specifications. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(5):5920–5929, Jun. 2023.

[Far23] Farama Foundation. Gymnasium documentation. `https://gymnasium.farama.org`, 2023. Accessed: [12/04/2023].

[FWJ+22] Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 18343–18362. Curran Associates, Inc., 2022.

[HLNB22] Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models, 2022.

[HS18] David Ha and Jürgen Schmidhuber. World models. 2018.

[KDS+23] Martin Klissarov, Pierluca D'Oro, Shagun Sodhani, Roberta Raileanu, Pierre-Luc Bacon, Pascal Vincent, Amy Zhang, and Mikael Henaff. Motif: Intrinsic motivation from artificial intelligence feedback, 2023.

[KNM+20] Heinrich Küttler, Nantas Nardelli, Alexander H. Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. The nethack learning environment, 2020.

[LCC+22] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022.

[LZZ22] Minghuan Liu, Menghui Zhu, and Weinan Zhang. Goal-conditioned reinforcement learning: Problems and solutions. In Lud De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 5502–5511. International Joint Conferences on Artificial Intelligence Organization, 7 2022. Survey Track.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[MLW+23] 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, 2023.

[NHX+23] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages, 2023.

[OB+19] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019.

[PPZC23] Rudra P. K. Poudel, Harit Pandya, Chao Zhang, and Roberto Cipolla. Langwm: Language grounded world model, 2023.

[RGG+23] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2023.

[RKH+21] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021.

[Sah23] Haris Sahovic. poke-env: A python interface for training reinforcement learning bots to battle on pokemon showdown. `https://github.com/hsahovic/poke-env`, 2023.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[SHM+] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. 529(7587):484–489.

[SLM+17] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[YGF+23] Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, Brian Ichter, Ted Xiao, Peng Xu, Andy Zeng, Tingnan Zhang, Nicolas Heess, Dorsa Sadigh, Jie Tan, Yuval Tassa, and Fei Xia. Language to rewards for robotic skill synthesis, 2023.

[ZW23] Li Zhong and Zilong Wang. Can chatgpt replace stackoverflow? a study on robustness and reliability of large language model code generation, 2023.