Exploring the Impact of Human Feedback on Reinforcement Learning for Game Playing AI Agents

Final Project Report

Harish Kumar

September 25, 2023

Contents

1 Introduction

1.1 Abstract

Artificial Intelligence (AI) plays a crucial role in advancing technology and has greatly improved the efficiency of numerous industries. One such industry that has a strong connection with AI is game development, as the two fields have continuously influenced and benefited from each other. While AI has introduced various techniques to streamline game creation, integrating Reinforcement Learning (RL) for AI-controlled players or Non-Player Characters (NPCs) remains a complex challenge.

In this report I focus on the development of my project that aims to explore a novel and recent RL technique known as Reinforcement Learning with Human Feedback (RLHF), that could address some of the challenges hindering the widespread use of RL in game development. The objective is to evaluate this technique for crafting AI agents viably in the game development industry.

1.2 Chosen Template

This project was inspired by the `Kane and Abel: AIs that play games` template, which serves as a guiding framework for exploring the integration of AI agents into the realm of game-playing.

1.3 Background

The history of AI and video games demonstrates a deep interconnection, with both fields significantly influencing each other. Stuart J. Russel and Peter Norvig highlight the appeal of games, emphasizing their complexity and the challenging decision-making abilities they require in their book "Artificial Intelligence: A Modern Approach"[1]. AI's journey in game-playing began with Claude Shannon's chess-playing computer in 1949[2] and reached a groundbreaking milestone with DeepMind's AlphaGo algorithm in 2016[3]. Through advancements like powerful external GPUs and Neural Networks, AI has made remarkable progress in learning and playing games, with OpenAI's successful use of reinforcement learning (RL) algorithms to teach AI agents to play Atari games using only the game's frame buffer and rewards[4].

These advancements have not only influenced AI research but have also permeated the field of game development. Features such as Adaptive NPCs, AI players, and automated testing have been shaped by neural network-based game-playing AIs that can learn and adapt. Developers increasingly rely on AI to streamline the game development process, employing reinforcement learning for map testing to identify bugs more efficiently than human testers[5]. Generative AI techniques aid in creating diverse and immersive game worlds by helping generate them.

Despite these notable achievements, integrating trained AI agents as AI players into games remains a challenging research area. Performance optimization, ease of training, effective reward shaping, and maintaining game balance and control of AI behavior continue to present significant obstacles[5]. While neural network-based AI systems, like OpenAI's AI system 'Five' defeating top players in Dota 2[6], have demonstrated success, the practical implementation of trainable game-playing AI players within games remains a distant goal.

1.4 Aims and Objectives

This project centers around an investigation into the approach introduced in OpenAI's research paper titled `"Deep Reinforcement Learning from Human Preferences"` by Christiano et al.[7]. The objective is to explore the potential of this technique in overcoming some of the limitations

associated with RL agents, particularly in the areas of effective reward shaping and enhancing control over AI behavior. The paper proposes an innovative method of integrating human feedback into the RL process, allowing for the intuitive shaping of rewards and aligning AI agents with human preferences.

Building upon this framework, this project seeks to evaluate the practical implications of incorporating human feedback into RL. In contrast to the Atari games and various environments used in the original paper, this project uses the original Doom game as the chosen environment due to its complexity and closer resemblance to modern games. Through this investigation, I aim to gauge the effectiveness of this approach in various metrics, including performance, training time, and the ease of reward shaping when compared to regular RL. Ultimately, this research aims to underscore the potential applicability of integrating human feedback in the realm of game development.

1.5 Applications

In line with the objectives outlined in the ***Aims and Objectives*** section, this project centers on investigating the practicality of training AI game-playing agents using reinforcement learning with human feedback (RLHF) and showcasing its relevance in the realm of game development. The core idea behind this approach is to empower game developers with the ability to efficiently train AI players, tailoring their behaviors to align with specific preferences. For instance, developers can utilize RLHF to train AI agents in first-person shooter (FPS) games, allowing each agent to adopt distinct playstyles based on factors like the selected loadout. Additionally, RL offers the flexibility needed to fine-tune these models for optimal performance. This is particularly advantageous in the context of multiplayer games, where AI bots must provide human players with engaging and challenging experiences.

It is essential to recognize that while this project primarily examines the potential of Reinforcement Learning with Human Feedback (RLHF) in game development, the technique holds broader applications. Notably, it is already in use by the Large Language Model (LLM), ChatGPT[8], to enhance response quality and provide more accurate and informative information.

2 Literature Review

2.1 Search Strategy

To conduct a thorough literature review on game-playing agents, their development over time, and emerging techniques in this field, I implemented a systematic search strategy. The goal was to identify pertinent studies published in reputable peer-reviewed journals such as IEEE[9], ResearchGate[10], and Nature[11], as well as through online search engines like Google. I specifically searched for papers related to the following topics:

- Game Playing AIs
- The Application of Machine Learning and AI in Game Development
- The Advancements in Reinforcement Learning with Human Feedback

2.2 Research Papers Reviews

The research papers that I found to be the most relevant to the project are reviewed and analysed below.

2.2.1 Playing Atari with Deep Reinforcement Learning[4]

In this paper, a team of researchers from Google's DeepMind present their work on using AI to play games. Their goal is to develop a single RL (Reinforcement Learning) algorithm capable of learning and playing various Atari 2600 games, taking raw images as input and providing game actions. To achieve this, they propose using a Deep Q Network (DQN), which is a Deep Neural Network trained with the Q-Learning algorithm and stochastic gradient descent. The researchers evaluate their approach by comparing the scores achieved by their trained model against other popular RL methods in seven different Atari games. The results indicate that their approach outperforms existing solutions in six of the games, with performance even surpassing that of human players in three of them. This research marks significant progress in game-playing AI algorithms and identifies areas for future improvement.

However, it's worth noting that Deep Q-Learning, while providing state-of-the-art performance, still faces challenges in three games—Q*bert, Seaquest, and Space Invaders—due to their complexity and the need for long-term strategies. The authors acknowledge that further research is necessary to address these challenges. Additionally, the paper highlights the computational demands of training deep neural networks for reinforcement learning, which could limit the scalability of the approach to more complex tasks or larger game environments. Furthermore, an issue with the algorithm's convergence in certain scenarios, not mentioned in the paper, was later detected in practical usage. Nevertheless, these challenges were subsequently addressed by OpenAI's PPO algorithm[12].

2.2.2 Deep reinforcement learning from human preferences[7]

In this research paper, a collaborative effort between OpenAI and Google's DeepMind Technologies, the authors delve into the impact of human feedback on the reinforcement learning process across diverse environments, encompassing Atari games and 3D robotic scenarios. The initial phase involves the AI agent operating randomly, with two video clips of its behavior presented to human evaluators. These evaluators then select the clip that comes closer to achieving the task's objective. Through an iterative cycle of this feedback loop, the AI gradually forms a model of the desired task outcome. This incorporation of human feedback empowers the agents to acquire task-specific insights, resulting in more intuitive decision-making that aligns with human expectations.

The experimental results underscore significant achievements, with the AI systems surpassing human performance levels in various tests, even when lacking direct access to the environment's state information. For instance, in the Atari game SeaQuest, the AI system learned to prioritize oxygen preservation. Notably, the researchers successfully trained the agents to maintain parity with other cars in Enduro, rather than merely maximizing in-game scores by overtaking them. This accomplishment would have been significantly more challenging and complex to achieve through manual reward design methods rather than leveraging human feedback.

While highlighting the successes of their approach, the authors also acknowledge certain limitations, primarily the constraint of obtaining limited feedback from humans. Additionally, they propose promising avenues for future advancements, including the exploration of interactive feedback loops where the AI actively seeks input from humans and the application of this feedback in practical scenarios such as robotics. By recognizing these shortcomings and offering potential directions for further development, the research paves the way for enhanced learning techniques and the real-world deployment of AI systems.

2.2.3 The Role of Machine Learning in Game Development Domain - A Review of Current Trends and Future Directions[5]

In this paper, the authors delve into the substantial contributions of machine learning within the realm of game development, spanning various critical areas such as character behavior modeling, procedural content generation, player analytics, and game testing and debugging. They underscore the inherent advantages of integrating machine learning algorithms into these domains, which, in turn, facilitate the creation of more lifelike and dynamic game characters, the generation of diverse and captivating game content, and the acquisition of invaluable insights into player behaviors and preferences.

Furthermore, the paper scrutinizes the application of machine learning techniques in crafting AI players for games like Starcraft, wherein these AI entities partake in resource gathering, army construction, and strategic confrontations against human players. Despite the active research around the concept of "AI as a Player", it has not yet found widespread implementation in actual production environments. Within this context, the paper introduces intriguing techniques such as Neuroevolution and DeepStack, both of which are subjects of ongoing investigation for this specific purpose.

Additionally, the authors allude to Kim et al.'s proposal[13], advocating for the infusion of human perspectives into AI development. This approach allows for the tailored design of AI players, aligning with user expertise and ultimately elevating the overall gaming experience.

2.2.4 ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning[14]

This paper introduces ViZDoom, a versatile and adaptable platform tailored for visual reinforcement learning research, leveraging the popular video game Doom as its foundation. The paper seeks to address the demand for a standardized environment that empowers researchers to conceive and assess AI algorithms grounded in visual input, within intricate and ever-evolving virtual settings.

ViZDoom effectively serves as a bridge connecting the Doom game engine with AI frameworks, effectively enabling researchers to train and evaluate reinforcement learning agents that rely on visual cues extracted from the game environment. The platform encompasses an array of formidable scenarios and gameplay modes, affording researchers the opportunity to delve into diverse facets of visual perception, decision-making, and control, all within the immersive realm of first-person shooter games.

The paper provides a comprehensive exploration of ViZDoom's architecture and features, accompanied by a series of illuminating experiments and case studies that highlights its immense potential in propelling visual reinforcement learning research to new heights.

2.3 Review Conclusion

The research papers reviewed above have played a vital role in shaping the path, objectives, and execution of this project, each contributing significantly to the field of game-playing AI. Specifically, OpenAI's paper, "Deep reinforcement learning from human preferences"[12], had a profound impact as it served as the driving force behind the adoption of Reinforcement Learning with Human Feedback (RLHF) in this project.

Moreover, the paper titled "The Role of Machine Learning in Game Development Domain - A

Review of Current Trends and Future Directions"[7] provided valuable insights into the landscape of AI and Machine Learning within game development. It effectively highlighted the challenges faced in different aspects of this field, inspiring my pursuit of RLHF as a novel solution to overcome the limitations associated with traditional RL when developing AI game players.

Building upon the principles introduced in "Playing Atari with Deep Reinforcement Learning"[12] I plan to apply similar techniques to train the game-playing agents within the dynamic Doom game environment, making use of the ViZDoom tool[14]. Ultimately, this project aims to synthesize and advance the concepts elucidated in these research papers, contributing to the evolution of sophisticated AI systems in the realm of game development.

3 Project Design

3.1 Hypotheses

Before delving into the project's design, it is crucial to formalize the hypotheses that will be evaluated. This process helps establish the project's requirements for achieving its goal. The project's design will revolve around confirming the following hypotheses:

1. Integrating human feedback will lead to the agent adopting behaviors aligned with human preferences, simplifying the process of reward shaping.
2. The time investment required from humans during the training process falls within an acceptable range.
3. The training process employing human feedback will yield faster convergence of the game-playing agent in comparison to agents trained without human feedback.

3.2 Development Methodology

To evaluate the formulated hypotheses, the implementation methodology will entail training two identical RL agents using distinct pipelines. The first pipeline will employ conventional RL techniques, with the agent learning to select specific actions for input frames from the game environment, relying on rewards issued by the environment. In contrast, the second pipeline will also employ RL techniques but will incorporate human feedback by training a reward predictor. In this setup, the agent will learn optimal actions for input frames based on rewards predicted by the human feedback-trained model.

To execute this experiment, five key components are essential: an RL agent, a game environment, a reward predictor, a training pipeline tailored for standard RL training, and a training pipeline equipped to collect human feedback.

3.2.1 Components

3.2.1.1 Reinforcement Learning (RL) Agent

The foundational component of the project is the RL agent itself, designed as a versatile class usable seamlessly across both pipelines without requiring modifications. This agent possesses the capability to learn from the rewards generated by its actions within the environment, allowing it to navigate the environment more proficiently when presented with a frame from the game environment as an input.

The RL agent will employ the Proximal Policy Optimization (PPO) algorithm[12], a highly esteemed RL algorithm renowned for its stability and effectiveness. PPO achieves this stability through a clever technique that limits the maximum change the model undergoes during training. It adheres to the actor-critic paradigm, with an actor network responsible for action selection based on the current policy and a critic network tasked with estimating state values. The actor strives to learn a policy that maximizes the expected cumulative reward, while the critic assesses and provides feedback on policy quality.

PPO was selected over the Deep Q Network (DQN) due to its superior stability and sample efficiency compared to DQN[4]. Given the project's context, where the agent learns from a reward function actively shaped by human feedback – potentially leading to evolving trajectories – algorithmic stability is of paramount significance.

3.2.1.2 Game Environment

The next critical component is the game environment in which the RL agent will operate and undergo training. In this project, the chosen game environment is Doom. The environment will be implemented as its own class, enabling the creation of a new Doom game environment with a loaded level. It will provide functionality for receiving game frames, conveying game state information (e.g., whether the episode is complete), resetting the environment, and executing actions that yield appropriate rewards.

To construct the Doom Environment, the project will make use of the ViZDoom[14] tool discussed in the ***Literature Review*** section. This tool offers an efficient means to create and interact with custom levels within the Doom game using Python. It furnishes the essential components required for training and operating an RL agent across various Doom game scenarios.

Additionally, the VizDoom library will be integrated into OpenAI's Gym environment. This choice was made to harness the utility functionalities provided by the Gym library. Gym is a well-established and mature library in the RL community, offering wrappers for various functionalities, including performance recording, video recording, and image data preprocessing (e.g., grayscale conversion, frame stacking, and image scaling). These capabilities will streamline the development process.

The initial emphasis will be placed on the "basic" level configuration provided by ViZDoom. In this "basic" setup, Doom Guy[15] navigates a confined space, maneuvering left and right, while confronting a stationary Cacodemon[16] that must be eliminated through gunfire. **Figure 1** visually illustrates this level. This fundamental configuration suffices for testing the formulated hypotheses. However, as development progresses and time allows, the project remains open to exploring more intricate levels offered by ViZDoom[14].

3.2.1.3 Reward Predictor

The reward predictor serves as the core component of the training pipeline that integrates human feedback. It functions as a neural network responsible for learning to predict rewards for a given state based on human preferences between two trajectories.

The reward predictor employs a relatively simple convolutional network, closely resembling the critic network of the PPO agent. However, the primary distinction lies in the training logic, which draws inspiration from the approach introduced in the Deep reinforcement learning from human

Figure 1: Screenshot of the basic level in ViZDoom.

preferences paper[7] by OpenAI. Instead of computing the loss using a standard loss function, which calculates mean squared error loss, the predictor uses a different method outlined in the paper.

Equation 1 outlines the technique for calculating the probability that one trajectory is preferred by a human over another. Here, $\hat{P}\left[\sigma^1 \succ \sigma^2\right]$ denotes the probability that trajectory σ^1 is chosen as the preferred option over trajectory σ^2, based on human preferences. This probability estimate relies on the reward function estimate $\hat{r}$, which is implemented as a neural network and draws inspiration from the principles of the Bradley-Terry model.

$$\hat{P}\left[\sigma^1 \succ \sigma^2\right] = \frac{\exp \sum \hat{r}\left(o_t^1, a_t^1\right)}{\exp \sum \hat{r}\left(o_t^1, a_t^1\right) + \exp \sum \hat{r}\left(o_t^2, a_t^2\right)} \tag{1}$$

Equation 2 represents the loss equation responsible for calculating the loss using the computed probabilities of human preference between trajectories. In this equation, μ determines the contribution of each component to the loss. For example, if trajectory 1 is preferred over trajectory 2, $\mu(1)$ will equal 1 and $\mu(2)$ will equal 0, and vice versa if trajectory 2 is preferred over trajectory 1. If both trajectories are equally preferred, both $\mu(1)$ and $\mu(2)$ will equal 0.5.

$$\text{loss}(\hat{r}) = -\sum_{(\sigma^1, \sigma^2, \mu) \in \mathcal{D}} \mu(1) \log \hat{P}\left[\sigma^1 \succ \sigma^2\right] + \mu(2) \log \hat{P}\left[\sigma^2 \succ \sigma^1\right] \tag{2}$$

3.2.1.4 Training Pipeline for Regular Reinforcement Learning

The training pipeline for regular reinforcement learning combines the PPO agent and the Doom game environment to construct a training loop that facilitates the learning process. This pipeline will manifest as a command-line interface (CLI) script, ensuring user-friendliness.

This pipeline utilizes the aforementioned components, including the PPO agent and the Doom game environment, to implement the RL training loop. Additionally, it incorporates the Tensorboard library to record and analyze diverse training and performance metrics of the RL agent. Tensorboard offers visualization tools and statistics that aid in monitoring the agent's progress and and identifying areas for improvement. The training pipeline adheres to a standard RL training procedure, encompassing the following stages:

1. **Interaction:** The RL agent actively interacts with the game environment. It observes the game state, selects actions based on its policy, and executes those actions in the environment.
2. **Feedback and Reward:** The environment provides feedback to the agent in the form of rewards. These rewards serve as a crucial element in the agent's learning process, indicating the quality of its actions.
3. **Memory Buffer:** Interactions between the agent and the environment are recorded in a memory buffer. This buffer stores observations, actions, and rewards, which will later be used to train the agent.
4. **Policy Update:** Using the Proximal Policy Optimization (PPO) algorithm, the agent updates its policy based on the received rewards. This update aims to enhance the agent's decision-making abilities over time.
5. **Training Loop:** These steps are repeated in a continuous loop, allowing the agent to continuously learn from its experiences and improve its performance.

Figure 2 visually illustrates the progression of these steps, showcasing the iterative nature inherent to the RL training process.

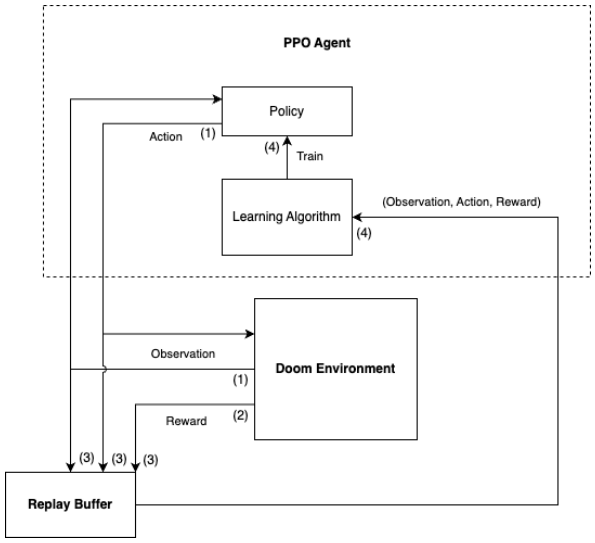


Figure 2: Structure of the regular RL training pipeline.

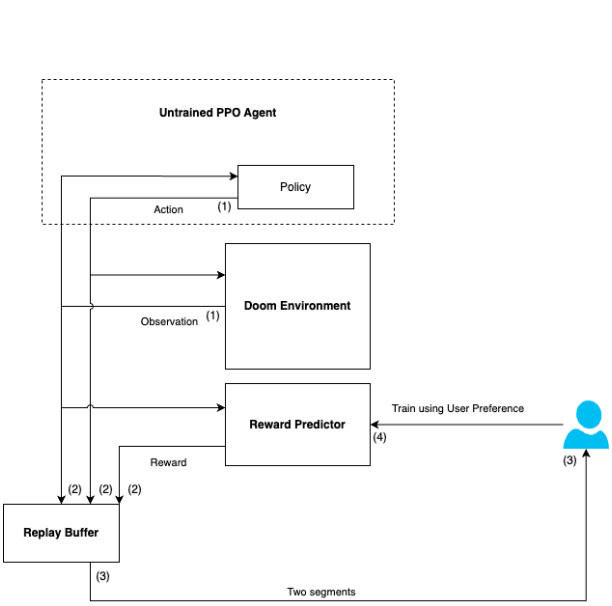
3.2.1.5 Training Pipeline for Reinforcement Learning with Human Feedback

This pipeline is designed to simultaneously train the PPO agent in the Doom game environment while training a reward predictor based on human feedback. To ensure efficient feedback collection from humans, this training pipeline will feature a graphical user interface (GUI).

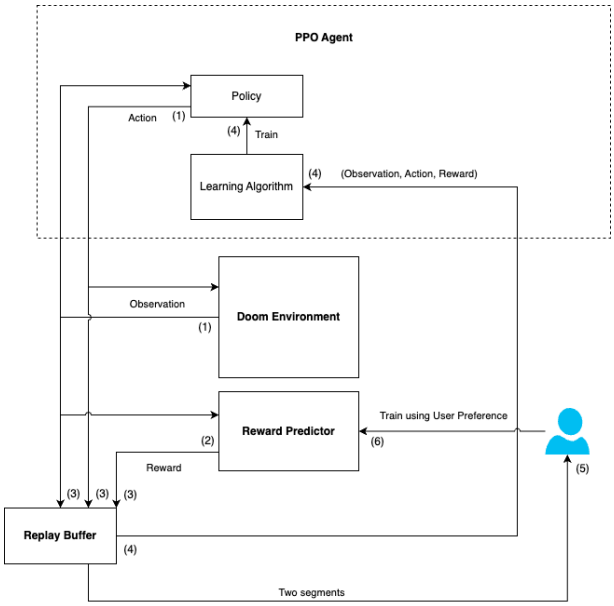
The implementation of this training pipeline takes inspiration from OpenAI's approach described in their paper[7], albeit with some simplifications. Notably, it does not delve into advanced techniques such as utilizing asynchronous training of the agent and the reward predictor, multiple reward predictors and choosing trajectory pairs based on uncertainity, as seen in the original paper. Instead, this pipeline will consist of two phases: the pre-training phase and the training phase.

Pre-Training Phase:

The pre-training phase sets the foundation for the subsequent training phase, providing the reward



(a) Structure of the pre-training phase of the RLHF training pipeline.



(b) Structure of the training phase of the RLHF training pipeline.

Figure 3: Structure of the RLHF training pipeline.

predictor with initial weights and enabling more effective convergence. The structure of the pre-training phase of the pipeline is described below and illustrated in **Figure 3a**.

1. **Interaction:** The pre-training phase begins with the untrained PPO agent taking random actions in the Doom game environment, akin to regular RL training.
2. **Interaction Storage:** Interactions between the agent and the environment are stored in the replay buffer in memory.
3. **Human Preference Collection:** After accumulating sufficient interactions, random pairs of interaction segments are sampled from the replay buffer and presented to a human evaluator. The evaluator selects their preference, indicating which behavior they believe best aligns with the desired behavior. A wireframe of this UI is depicted in **Figure 4**.
4. **Reward Predictor Training:** The selected interaction segments and preferences are employed to train the reward predictor, enhancing its ability to predict the human evaluator's preferences.
5. **Pre-Training Loop:** The above training process repeats until the reward predictor has been trained on a predetermined number of segment pairs and preferences (following the approach used by the authors of the paper[7], who used 500 preferences).

Training Phase:

The pre-training phase is followed by the training phase which will train both the reward predictor and the reinforcement learning agent simultaneously. The structure of training phase of the pipeline is described below and illustrated in **Figure 3b**.

1. **Interaction:** Similar to the pre-training phase, the training phase begins with the PPO agent taking random actions in the Doom game environment.
2. **Feedback and Reward:** Here, the reward predictor provides feedback to the agent in the

form of rewards based on the state of the observation buffer, reflecting the quality of its actions.

3. **Storing of Interactions:** Interactions between the agent and the environment, along with the feedback, are saved in the replay buffer in memory.
4. **Policy Update:** Using the collected experience in the replay buffer, the agent updates its policy using the Proximal Policy Optimization (PPO) algorithm to enhance its decision-making abilities.
5. **Human Preference Collection:** At regular intervals, segments of a specified size are created from this sequence within the replay buffer. These segments are then randomly paired and presented to a human evaluator, who selects their preference, indicating the behavior they believe best aligns with the desired outcome. Our approach here diverges slightly from that of the authors of the OpenAI paper[7], who sampled a single pair of segments based on the difference in their cumulative rewards compared to the ensemble of reward predictors and collected preferences for those pairs. In this project, random sampling of segment pairs is chosen to accommodate limited development time. To ensure the reward predictor's stability and prevent convergence towards incorrect behavior due to poor segment pairs, preferences are collected for random pairs from all the generated segments. A wireframe of this user interface is depicted in **Figure 4**.
6. **Reward Predictor Training:** The selected interaction segments and preferences are used to further train the reward predictor, improving its ability to predict the human evaluator's preferences.
7. **Training Loop:** The training process, wherein the PPO agent and the reward predictor are simultaneously trained, continues until the PPO agent exhibits satisfactory behavior.

Upon closer examination, it's apparent that this pipeline, despite integrating human feedback, shares a fundamental resemblance to conventional reinforcement learning. The primary distinction lies in the computation of rewards, which are now generated using a trainable reward function instead of directly extracted from the game environment. The introduction of the reward predictor introduces a dynamic element into the training process, potentially leading to instability. This consideration played a pivotal role in the selection of the PPO algorithm for this project.

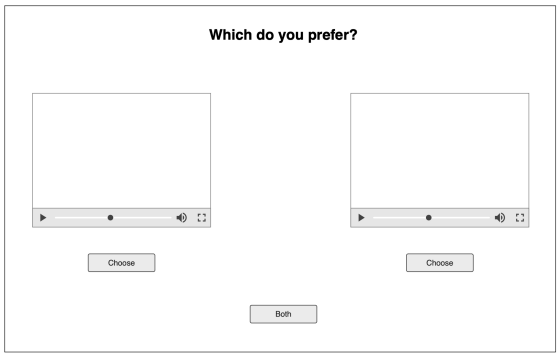


Figure 4: UI wireframe for RLHF training pipeline to collect human feedback.

3.3 Evaluation Methodology

Having designed the project's components, it is essential to establish the evaluation criteria for measuring success once the implementation phase is complete. The primary goal of the evaluation is to assess whether the project substantiates the proposed hypotheses. The subsequent subsections

delineate the testing methodologies for each of the three hypotheses. It's noteworthy that the project can be considered successful if at least two of these hypotheses are confirmed, as this would provide substantial evidence to present the methodology as a viable approach.

3.3.1 Evaluation Methodology for Hypothesis 1

The first hypothesis posits that incorporating human feedback will lead to the game-playing agent exhibiting behaviors preferred by humans, consequently facilitating reward shaping. To test this hypothesis, the AI agent will undergo training using the pipeline that incorporates human feedback. The anticipated behavior, in this scenario, is the successful elimination of the Cacodemon[16]. By monitoring whether the agent converges on this expected behavior during training, we can gauge the effectiveness of incorporating human feedback in shaping the agent's conduct. Successfully training the AI agent using human feedback would validate the viability of this technique for developing NPCs and game-playing AIs within the context of game development.

3.3.2 Evaluation Methodology for Hypothesis 2

The second hypothesis posits that the time involvement of humans in the training process remains within a reasonable range. This can be evaluated by assessing the time commitment of the human evaluator. In the "Deep Reinforcement Learning from Human Preferences" paper[7], the authors achieved satisfactory results with just one hour of human involvement. Considering the resources available for this research, a reasonable timeframe for testing this hypothesis would fall between one to four hours of human involvement. Although this target time range is broader than the duration reported in the paper, it still represents a significantly shorter timeframe compared to traditional game development methods like State Machines.

3.3.3 Evaluation Methodology for Hypothesis 3

The third and final hypothesis postulates that the training process using human feedback will lead to faster convergence of the game-playing agent in comparison to agents trained without human feedback. To evaluate this hypothesis, the time required by two distinct AI agents, trained using different pipelines, will be compared in their quest to attain a similar average score in the game, as internally calculated by ViZDoom[14]. If the AI agent trained using human feedback demonstrates a quicker convergence toward the expected behavior, it would substantiate this hypothesis.

4 Implementation

This section provides a detailed walkthrough of translating the theoretical concepts and designs discussed earlier into practical solutions. It covers the technologies used, architectural choices and integration of various components. It's important to emphasize that this section will primarily concentrate on the implementation of the more intricate and critical project elements. Furthermore, the code snippets described below have been slightly adapted for use in the project. A link to the repository of the project implementation can be found in the ***Code Repository*** section of the ***Appendix***.

4.1 Technologies Used

The project was implemented entirely using the Python programming language, with PyTorch serving as the primary library for building neural networks. Additionally the GUI based components

were implemented using Interactive Python Notebooks. The Python development environment was created using Anaconda, and the entire project was developed on a Windows PC with the following specifications:

- **OS:** Windows 11
- **Memory**: 32 GB DDR4 RAM
- **Processor**: Ryzen 7 3800X
- **Graphics Card**: NVIDIA RTX 3080 10GB LHS

Additional information regarding setting up the project locally can be found in the ***Usage Guide*** section in the ***Appendix***.

4.2 Implementation Details of Components

4.2.1 Reinforcement Learning (RL) Agent

As the project's RL agent, the Proximal Policy Optimization (PPO) algorithm was chosen as the primary approach. This decision was made to gain precise control over the training process and ensure its adaptability across various pipelines. Heavy inspiration was drawn from Costa Huang's methodology[17], which was selected due to the performance of that implementation closely resembling the performance of the implementation in the `stable_baselines` library.

Two separate networks, one for the actor and another for the critic, were created, and the algorithmic logic was decoupled from the neural network architecture to enhance modularity. Since both these networks share identical convolutional layers, a common set of instances of convoloutional layers were used for both the networks to optimize memory usage, and reduce the size of saved models. Additionally the layers in the network use orthogonally initialized weights as suggested in the blog[17]. This is done using a custom utility function called `ppo_init()`. To improve flexibility, the agent classes were designed to use numpy arrays as inputs and outputs instead of tensors, thus reducing the dependence on PyTorch within the pipelines.

The following code snippet illustrates the creation of the networks as described above.

```
base_network = nn.Sequential(
    ppo_layer_init(nn.Conv2d(observation_space.shape[0], 32, 8, stride=4)),
    nn.ReLU(),
    ppo_layer_init(nn.Conv2d(32, 64, 4, stride=2)),
    nn.ReLU(),
    ppo_layer_init(nn.Conv2d(64, 64, 3, stride=1)),
    nn.ReLU(),
    nn.Flatten(),
    ppo_layer_init(nn.Linear(64 * 11 * 16, 512)),
    nn.ReLU(),
)
actor_network = ppo_layer_init(nn.Linear(512, action_space.n if
 ↪  isinstance(action_space, Discrete) else action_space.shape), std=0.01)
critic_network = ppo_layer_init(nn.Linear(512, 1), std=1)
```

When predicting the optimal action, these layers are utilized to compute the optimal action based on the given observation. This is achieved by using the actor network to create a probability distribution of the action with the highest reward probability. This logic can be found in the `forward()`

function of the `BasePpoAgent` class. The following code snippets illustrate this prediction process:

```
observations = torch.tensor(observations)

hidden = self.network(observations / 255.0)
logits = self.actor(hidden)
value = self.critic(hidden)

probs = Categorical(logits=logits)

actions = probs.sample()
```

The ability to learn is facilitated by the training process exposed using the `train()` function in the `BasePpoAgent` class. This training process takes into consideration core implementation details and Atari-specific implementation details mentioned in the blog[17]. It works as follows:

1. **Read Replay Buffer:** The replay buffer data, initially in numpy arrays, is transformed into PyTorch tensors for further computations.
2. **Anneal Learning Rate of Optimizer:** Optionally, the optimizer's learning rate can be reduced using the provided coefficient to enhance training stability.
3. **Advantages and Returns Computation:** Advantages reflect the benefits of actions during training. When Generalized Advantage Estimation (GAE) is active, advantages are iteratively calculated for each replay buffer step, considering future potential rewards and values. If GAE is not used, advantages are computed as the difference between estimated cumulative rewards and values at each time step.
4. **Mini-Batch Preparation:** The replay buffer steps are shuffled and organized into mini-batches to facilitate training.
5. **Loss Computation:** For each mini-batch, the agent computes new log probabilities, entropy, and updated value estimates based on mini-batch observations and actions. These values are then used to calculate two key losses: the policy loss and the value loss.
    - **Policy Loss:** The policy loss has two components: `pg_loss1` and `pg_loss2`. `pg_loss1` penalizes actions that diverge from the old policy but align with advantages, promoting exploration. On the other hand, `pg_loss2` ensures stability by constraining the ratio of new and old policy probabilities within a predefined range, often set to values like `0.1`. The final policy loss is determined as the maximum between these two options, striking a balance between exploration and exploitation, thereby enhancing policy stability during training.

      ```
      pg_loss1 = -mini_batch_advantages * ratio
      pg_loss2 = -mini_batch_advantages * torch.clamp(ratio, 1 - clip_coef, 1
       ↪  + clip_coef)
      policy_loss = torch.max(pg_loss1, pg_loss2).mean()
      ```

    - **Value Loss:** The value loss evaluates the critic network's predictions regarding expected cumulative rewards (returns) in a given state. It is computed as the mean squared error between predicted values and actual returns. However, for training stability, the value loss undergoes clipping, ensuring it doesn't exceed a predefined threshold, typically set

to values like `0.5`. This clipping mitigates abrupt changes to the network, promoting stability.

```
new_value = new_value.view(-1)
if clip_vloss:
    v_loss_unclipped = (new_value - b_returns[mb_inds]) ** 2
    v_clipped = b_values[mb_inds] + torch.clamp(
        new_value - b_values[mb_inds],
        -clip_coef,
        clip_coef,
    )
    v_loss_clipped = (v_clipped - b_returns[mb_inds]) ** 2
    v_loss_max = torch.max(v_loss_unclipped, v_loss_clipped)
    value_loss = 0.5 * v_loss_max.mean()
else:
    value_loss = 0.5 * ((new_value - b_returns[mb_inds]) ** 2).mean()
```

These losses are then combined to compute the total loss. The contributions of the entropy loss and value loss to the final loss are determined by coefficient hyper-parameters.

```
entropy_loss = entropy.mean()
loss = policy_loss - entropy_coef * entropy_loss + value_loss * value_coef
```

6. **Weight Adjustments:** The total loss is then back-propagated into the networks to adjust the weights.
7. **Multiple Epoch Iteration:** Steps 4 to 6 are reiterated for the specified number of training epochs.

4.2.2 Game Environment

As per the design, the game environment is encapsulated within the `VizDoomEnv` class, which functions as a OpenAI Gym wrapper for the ViZDoom instance. The code for the wrapper was not written from scratched but was instead sourced from the ViZDoom GitHub repository[18]. As expected, the `VizDoomEnv` class adheres to a OpenAI Gym interface, thereby enabling compatibility with the wrappers (pre-processors) offered by Gym like `ResizeObservation`, `GrayScaleObservation`, and `FrameStack`. These wrappers can therefore be used in the pre-processing pipeline for the observations in the training pipeline.

4.2.3 Reward Predictor

In alignment with the design principles, the reward predictor has been constructed as a class capable of forecasting rewards based on provided observations and learning from human preferences. Much like the RL agent class, the fundamental logic of the reward predictor has been abstracted from the network architecture to enhance modularity. The core functionality of the predictor is encapsulated within the `BaseHumanPreferenceRewardPredictor` class, while the network architecture is defined in the `DoomHumanPreferenceRewardPredictor` class. To maintain consistency with the RL agent, these reward predictor classes employ numpy arrays for inputs and outputs, thereby ensuring the separation of PyTorch from the training pipeline code.

The design of the reward predictor's neural network takes inspiration from insights presented by the authors in the *Experimental Details* section of the RLHF paper[7] by OpenAI. It incorporates elements such as leaky ReLU layers, dropout layers, and batch normalization to mitigate overfitting. The network's architecture is depicted in the code snippet below, with layer separation employed to dynamically calculate the input size for the fully connected layer.

```
observation_encoder = nn.Sequential(
    nn.Conv2d(observation_space.shape[0], 32, kernel_size=8, stride=4),
    nn.LeakyReLU(negative_slope=0.01, inplace=True),
    nn.BatchNorm2d(32),
    nn.Dropout2d(drop_out),

    nn.Conv2d(32, 64, kernel_size=4, stride=2),
    nn.LeakyReLU(negative_slope=0.01, inplace=True),
    nn.BatchNorm2d(64),
    nn.Dropout2d(drop_out),

    nn.Conv2d(64, 64, kernel_size=3, stride=1),
    nn.LeakyReLU(negative_slope=0.01, inplace=True),
    nn.BatchNorm2d(64),
    nn.Dropout2d(drop_out),

    nn.Conv2d(64, 64, kernel_size=3, stride=1),
    nn.LeakyReLU(negative_slope=0.01, inplace=True),
    nn.BatchNorm2d(64),
    nn.Dropout2d(drop_out)
)

with torch.no_grad():
    dummy_input = torch.randn(1, *observation_space.shape)
    conv_output = observation_encoder(dummy_input)
    conv_output_size = torch.prod(torch.tensor(conv_output.size()[1:]))

reward_predictor = nn.Sequential(
    nn.Flatten(),
    nn.Linear(conv_output_size, hidden_size),
    nn.LeakyReLU(negative_slope=0.01, inplace=True),
    nn.Linear(hidden_size, hidden_size),
    nn.LeakyReLU(negative_slope=0.01),
    nn.Linear(hidden_size, 1)
)
```

This network is employed to predict the reward for any given observation. When used in the training pipeline, the predicted rewards are then normalized to have a zero mean and a consistent standard deviation, preventing instability due to large value fluctuations. This normalization is achieved using a `RunningStat` class that efficiently calculates mean and standard deviation in real-time without having a substantial memory footprint. This corresponding logic can be found in the `forward()` function of the `BaseHumanPreferenceRewardPredictor` class. The following code

snippets illustrate this prediction process.

```
observation_encodings = self.observation_encoder_network(observations / 255.0)
reward_predictions = self.reward_predictor_network(observation_encodings)
reward_predictions = reward_predictions.squeeze()

for reward_prediction in reward_predictions:
    self.running_stat.push(reward_prediction)
reward_predictions = self._normalize_rewards(reward_predictions)
```

The reward predictor also exposes a `train()` function for training the network using two trajectories and human preferences as input. The training process for the network is explained below:

1. **Reward Prediction for Trajectories:** The network computes total rewards for each frame within the two trajectories.
2. **Accumulation of Cumulative Rewards:** Cumulative rewards for both trajectories are computed independently based on the predicted rewards.
3. **Calculating Probability of Prefering Trajectory:** Utilizing the probabilities equation **Equation 1** explained in the design section, the probability of preferring one trajectory over the other is determined. This computation takes into account the exponential of latent rewards for both trajectories and ensures they sum to 1.

   ```
   prob_prefer_trajectory_1 = torch.exp(trajectory_1_latent_rewards_sum) /
    ↪  (torch.exp(trajectory_1_latent_rewards_sum) +
    ↪  torch.exp(trajectory_2_latent_rewards_sum))
   prob_prefer_trajectory_2 = torch.exp(trajectory_2_latent_rewards_sum) /
    ↪  (torch.exp(trajectory_1_latent_rewards_sum) +
    ↪  torch.exp(trajectory_2_latent_rewards_sum))
   ```

4. **Loss Computation for Probability Prediction:** To quantify the prediction accuracy, a loss is calculated by using the calculated probabilities in **Equation 2**. The loss function is then computed using the logarithms of the probabilities, with human preferences determining the contribution of the logarithmic probabilities of each trajectory.

   ```
   if preference == 0:
        mu1 = 1
        mu2 = 0
    elif preference == 0.5:
        mu1 = 0.5
        mu2 = 0.5
    else:
        mu1 = 0
        mu2 = 1

    mu1 = torch.tensor(mu1).to(self.device)
    mu2 = torch.tensor(mu2).to(self.device)
    loss = -mu1 * torch.log(prob_prefer_trajectory_1) - mu2 *
    ↪  torch.log(prob_prefer_trajectory_2)
   ```

The implementation of the loss calculation equation was moved to it's own custom loss class called `PreferenceLoss`.

5. **Weight Adjustments:** The calculated loss is subsequently employed for weight adjustments through backpropagation, effectively updating the neural network's parameters.
6. **Multiple Epoch Iteration:** Steps 1 to 5 are repeated for the specified number of training epochs.

4.2.4 Training Pipeline for Regular Reinforcement Learning (RL)

The RL training pipeline's implementation closely adheres to established design principles and is executed as a Python command-line script named `run_doom_rl_pipeline.py`. Its primary purpose is to train an instance of the `DoomPpoAgent` class within a selected ViZDoom level, utilizing the `VizDoomEnv` class. It follows conventional RL training methods, as illustrated in **Figure 2**. During this process, the agent engages with the environment continuously, accumulating experience that is subsequently used to train the agent to achieve specific behaviors aligned with the environment's reward system. This cyclic process repeats until the agent completes the desired number of steps.

In addition to the core pipeline features, several new functionalities were developed to enhance usability, including:

- Support for customizing hyperparameters through command-line arguments.
- Incorporation of progress bars using the `tqdm` library[19].
- Automatic saving of agent model weights when achieving a new highest average episodic return.
- Saving of training statistics to Tensorboard[20] for evaluation and analysis purposes.

More details regarding the usage of this pipeline script can be found in the ***Usage Guide*** section of the pipeline in the appendix.

4.2.5 Training Pipeline for Reinforcement Learning with Human Feedback (RLHF)

The RLHF pipeline, implemented as a Jupyter Notebook named `doom_ppo_agent_rlhf_training_pipeline.ipynb`, closely adheres to the established design. This implementation choice enables the straightforward rendering of graphical user interfaces (GUIs) using the `ipywidgets` library, as required by the design.

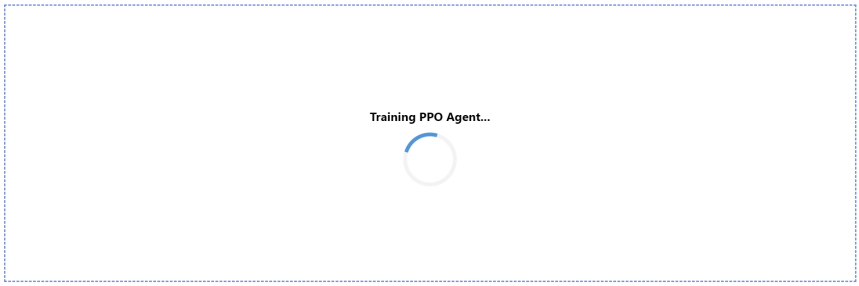
The primary objective of this implementation is to train an instance of the `DoomPpoAgent` class within a selected ViZDoom level, using the `VizDoomEnv` class. The training process aims to guide the agent in performing actions preferred by a human evaluator. This is achieved through the utilization of the `DoomHumanPreferenceRewardPredictor` class, which has been trained on human feedback to provide appropriate rewards for the agent's actions, rather than relying on the environment's reward system.

As outlined in the design section, the training process is divided into two main phases:

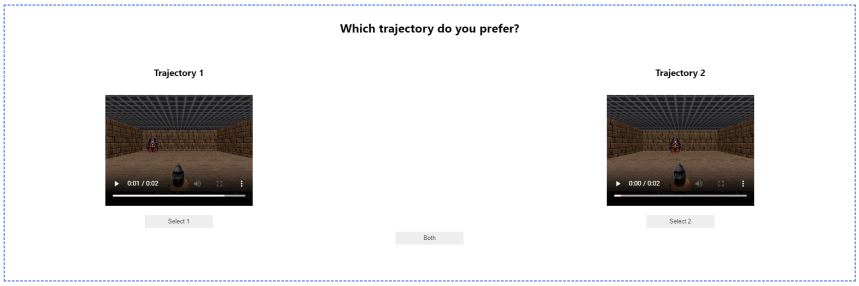
1. **Pre-Training Phase:** This initial phase focuses on providing context to the reward predictor, enabling it to properly reward the agent at the beginning of the training process. Human preferences are collected based on trajectories generated by an untrained RL agent exploring the environment. This phase ensures that the agent converges toward the desired behaviors during the subsequent training phase. **Figure 3a** illustrates the steps followed in this phase.

2. **Training Phase:** During the training phase, both the RL agent and the pre-trained reward predictor undergo simultaneous training. The RL agent's training loop follows the conventions of a typical RL pipeline. Periodically, at fixed intervals, human preferences are collected from randomly selected trajectories. These preferences are then used to further train the reward predictor, improving its ability to provide accurate rewards based on human feedback. **Figure 3b** illustrates the steps followed in this phase.

Figure 5a and **Figure 5b** depict the GUI as rendered within a Python notebook. These screenshots showcase the GUI's functionalities, including displaying training status, and facilitating the collection of preferences from the human trainer.



(a) Screenshot of the RLHF training pipeline GUI displaying training status.



(b) Screenshot of the RLHF training pipeline GUI collecting preferences.

Figure 5: Screenshot of the GUI rendered in the Python notebook.

Like the regular RL pipeline, in addition to the core pipeline features, several new functionalities were developed to enhance usability, including:

- Support for customizing hyperparameters through configuration dictionaries.
- Automatic saving of agent model weights when achieving after every training batch.
- Saving of training statistics to Tensorboard[20] for evaluation and analysis purposes.

More details regarding the usage of this pipeline notebook can be found in the Usage Guide section under the Appendix section.

5 Project Evaluation

To evaluate the project, I will follow the methodologies outlined in the ***Evaluation Methodology*** section. I'll start by establishing a performance baseline using the conventional RL pipeline, which will serve as a reference point for comparison. Then, I will examine the RLHF pipeline's performance compare it to the baseline. Ultimately, I will evaluate how well the RLHF pipeline aligns with the

defined hypotheses and whether it shows improvements over the traditional RL approach.

5.1 Establishing Baseline

As the initial step in the evaluation process, I established a baseline model to create a performance benchmark against which we can compare the RLHF pipeline. I chose to construct this baseline using the conventional RL pipeline and trained a model with it. This decision is based on the well-established effectiveness of Reinforcement Learning in training game-playing agents. Additionally, it serves as a proof of concept to ensure that the `DoomPpoAgent` class functions as expected.

I created the baseline model by executing the `run_doom_rl_pipeline.py` script with the hyperparameters listed in **Table 1**. These hyperparameters were adopted from the default settings recommended by Costa Huang in his PPO blog[17]. Since these default hyperparameters yielded effective results from the beginning, I opted to use them without modification for training the baseline model.

Argument	**Value**
–env-cfg	./envs/vizdoom/scenarios/basic.cfg
–learning-rate	0.0001
–total-timesteps	300000
–render-env	True
–enable-gpu	True
–track-stats	True
–num-envs	8
–num-steps	256
–anneal-lr	True
–enable-gae	True
–gamma	0.99
–gae-lambda	0.95
–num-minibatches	32
–training-epochs	10
–norm-adv	True
–clip-coef	0.1
–clip-vloss	True
–entropy-coef	0.01
–value-coef	0.5
–max-grad-norm	0.5
–target-kl	None

Table 1: Table of arguments used for RL training pipeline.

The baseline model, trained using the `run_doom_rl_pipeline.py` script with the specified parameters, demonstrated highly promising results. An impressive average episodic return of `85.716` was achieved by this model, with the highest recorded return being `95`. However, it's important to note that an average episodic return of `95` can only be attained when the Cacodemon[16] is spawned directly in front of the Doom Guy[15], allowing the demon to be shot in the first frame. In scenarios where the demon spawns elsewhere on the screen, the agent's score is lower due to a penalty of `-1` being applied for each frame of its existence.

Remarkably, this high average episodic return was attained by the RL agent in fewer than `130000` steps within the environment. The results from `5` test runs using the trained model with the `play_doom_with_trained_agent.py` script are presented in **Table 2**.

The effectiveness of this pipeline and the model it has trained can be further analyzed in the reward and loss graphs presented in **Figure 6**. In these graphs, the episodic return steadily increases while the episodic length decreases, indicating improved agent performance. Additionally, the value loss (the loss in value prediction) metric demonstrates the effectiveness of the training process as it gradually reduces without getting trapped in a local minimum. These results further corroborate the efficacy of the PPO algorithm for training game-playing agents.

The baseline model has been included with the source code, and additional instructions about testing it locally can be found in the ***Testing Trained Models*** sub-section within the ***Usage Guide*** section of the ***Appendix***. Detailed instructions for running the entire pipeline locally can be found in the ***Running Reinforcement Learning*** sub-section under the ***Usage Guide*** section of the ***Appendix***.

Run	Number of Episodes	Avg Rewards
1	50	86.3
2	50	82.72
3	50	86.82
4	50	86.32
5	50	86.42

Table 2: Test results of the baseline model from five runs of 50 episodes.

5.2 Testing the Reinforcement Learning with Human Feedback Pipeline

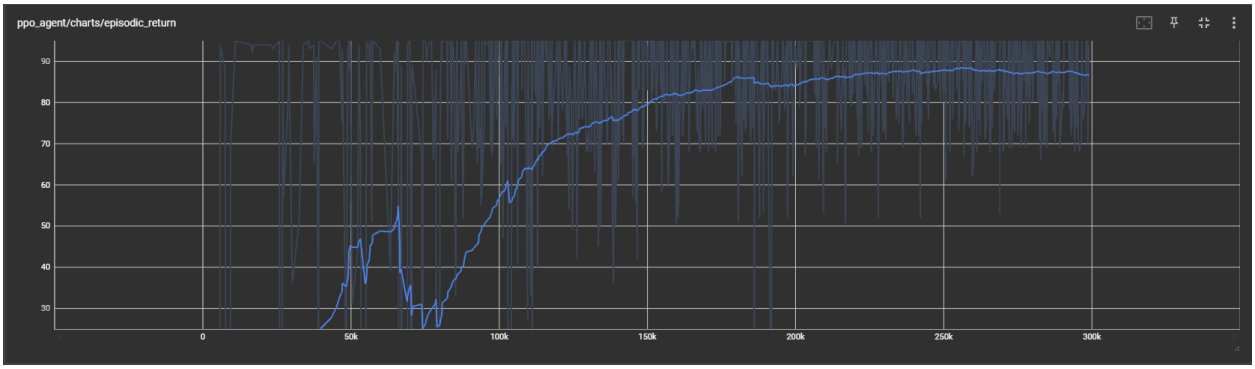
Having established the baseline, the established evaluation methodologies can be applied on the RLHF pipeline. To evaluate using the established methodolgies, I trained the agent using the RLHF pipeline to teach it how to track and shoot the Cacodemon[16], mimicking the reward system inherent to the VizDoom environment. When choosing between presented trajectories, I followed a set of rules to determine which trajectory favored the goal of training the agent to shoot the Cacodemon[16]. These rules are listed in decreasing order of importance:

After establishing the baseline, I applied the predefined evaluation methodologies to evaluate the RLHF pipeline. To conduct this evaluation, I trained the agent using the RLHF pipeline to learn how to track and shoot the Cacodemon[16], replicating the reward system inherent in the VizDoom environment, making it easier to compare it with the baseline model.

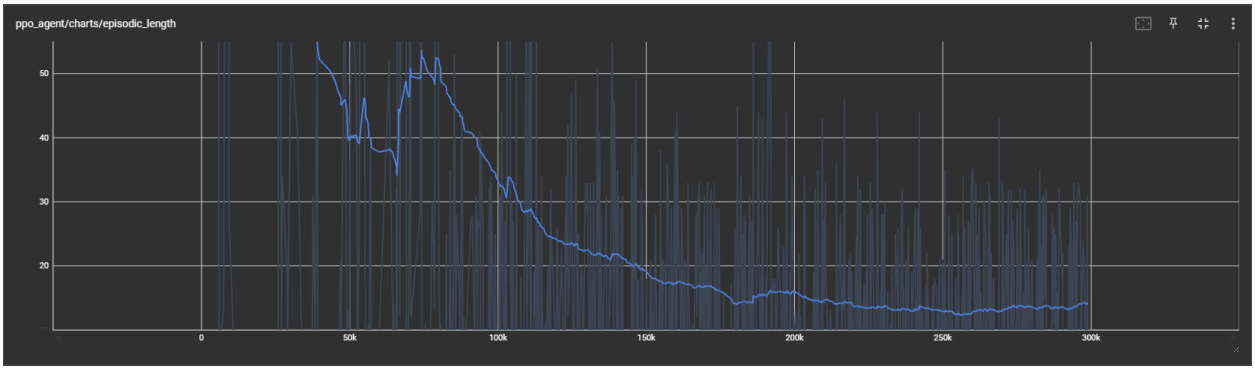
During the training process, when I was presented with two trajectories to choose from, I adhered to a set of rules to determine which trajectory best served the objective of training the agent to target and eliminate the Cacodemon[16]. These rules were ranked in decreasing order of importance:

1. Doom Guy[15] has killed the Cacodemon[16] the most number of times.
2. Doom Guy[15] is closer to the Cacodemon[16] in most frames of the video.
3. Doom Guy[15] is moving towards the Cacodemon[16].

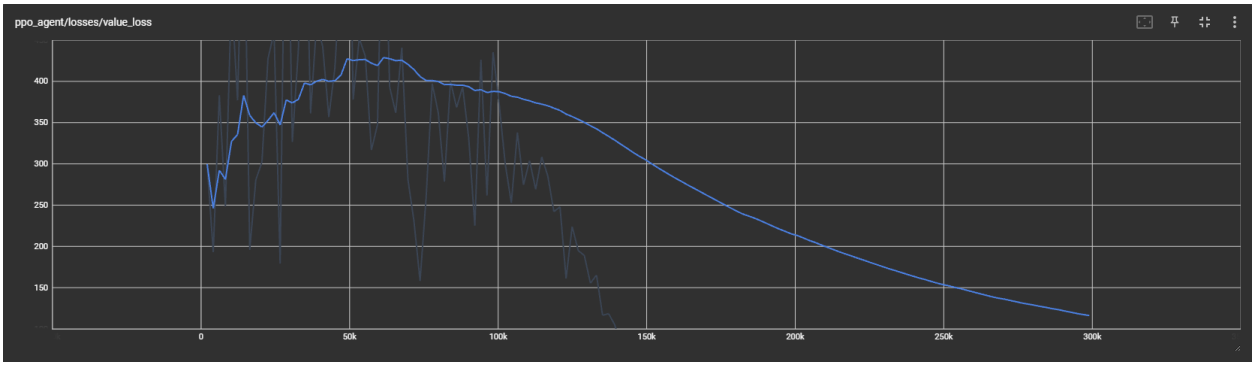
In cases where both trajectories exhibited similar qualities, I selected the `Both` option.



(a) Episodic return



(b) Episodic length



(c) Value loss

Figure 6: Training metrics for the RL pipeline run. In these graphs, the X-Axis represents the number of training steps and the Y-Axis represents the different values. (The darker lines represent smoothed data points and the low opacity lines represent the real data points).

The pipeline involved pre-training the reward predictor using 250 preferences based on trajectories from an untrained PPO agent, with reward predictor training at regular intervals of every 7th agent training process during the training phase. The hyperparameters used for testing this pipeline configuration are presented in **Table 3**.

Config Key	**Dictionary Name**	**Value**
env_cfg	pipeline_args	./envs/vizdoom/scenarios/basic.cfg
total_timesteps	pipeline_args	300000
render_env	pipeline_args	True
enable_gpu	pipeline_args	True
track_stats	pipeline_args	True
num_envs	pipeline_args	8
env_replay_buffer_size	pipeline_args	256
enable_pre_training	pipeline_args	True
num_pre_train_requests	pipeline_args	250
num_trajectory_frames	pipeline_args	64
human_feedback_interval	pipeline_args	7
learning_rate	agent_args	0.0001
anneal_lr	agent_args	300000
enable_gae	agent_args	True
gamma	agent_args	0.99
gae_lambda	agent_args	0.95
num_minibatches	agent_args	32
num_training_epochs	agent_args	10
norm_adv	agent_args	True
clip_coef	agent_args	0.1
clip_vloss	agent_args	True
entropy_coef	agent_args	0.01
value_coef	agent_args	0.5
max_grad_norm	agent_args	0.5
target_kl	agent_args	None
learning_rate	reward_predictor_args	1e-4
hidden_layer_size	reward_predictor_args	64
num_training_epochs	reward_predictor_args	1
model_path	reward_predictor_args	None

Table 3: Table of arguments supported by the run_doom_rl_pipeline.py script.

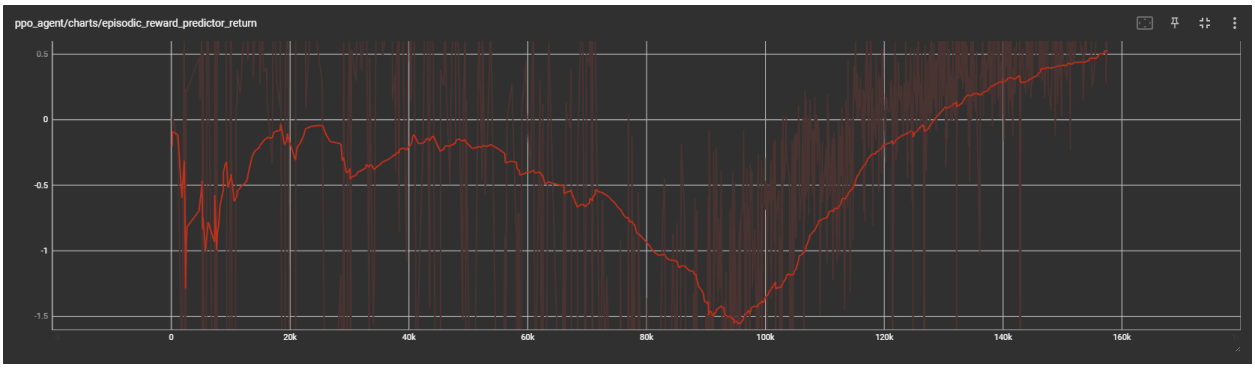
The model trained using this pipeline configuration demonstrated promising results. The trained model achieved an average episodic return of `70.34` at `130000` steps. The entire training process consumed approximately `2.5 hours` of my time, with the pre-training phase accounting for `45 minutes` and the remaining time dedicated to the training phase, where the pipeline requested preferences more sparsely.

Table 4 presents the results from `5` different test runs of the trained model using the `play_doom_with_trained_agent.py` script.

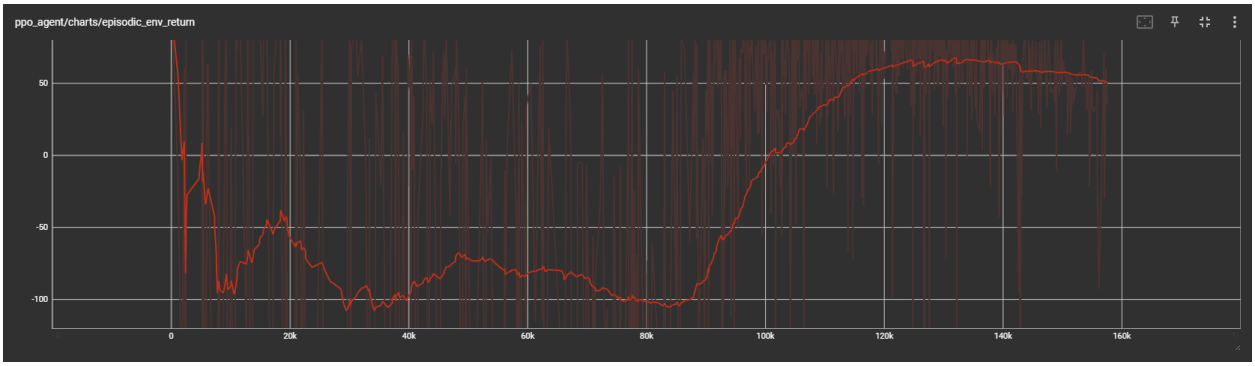
Run	Number of Episodes	Avg Rewards
1	50	64.8
2	50	73.1
3	50	71.78
4	50	74.12
5	50	67.9

Table 4: Test results of the model trained using the RLHF pipeline from five test runs of 50 episodes each.

The training process metrics are presented more clearly in the graphs in **Figure 7**. In addition to the average episodic returns from the reward predictor, **Figure 7** includes the episodic returns from the VizDoom environment. These environmental rewards exhibit the same desired behavior in the agent that I have been training the RLHF pipeline to teach the agent. The graphs show that initially, the average environmental episodic return and the average reward predictor episodic returns are opposite, but as training progresses, they begin to display the same trend, indicating that the reward predictor successfully learned to reward the agent for tracking and shooting the Cacodemon[16].



(a) Episodic Rewards from the Reward Predictor



(b) Episodic Rewards from the Environment

Figure 7: Training metrics for the RLHF pipeline run. In these graphs, the X-Axis represents the number of training steps and the Y-Axis represents the different values. (The darker lines represent smoothed data points and the low opacity lines represent the real data points).

It's worth highlighting that the lower average episodic return isn't a strong indicator of the agent's

ability to track and shoot the demon due to the specific reward system employed by VizDoom. The environment assigns a penalty of `-1` for each frame the agent exists and for every non-terminating action it takes. However, in practice, the agent performs exceptionally well, as showcased in the video submitted along with this report.

This trained model has also been included along with the source code. To test the model locally, please refer to the steps outlined in the ***Testing Trained Models*** sub-section within the ***Usage Guide*** section of the ***Appendix***. Detailed instructions for running the entire pipeline locally can be found in the ***Running Reinforcement Learning with Human Feedback Pipeline*** sub-section under the ***Usage Guide*** section of the ***Appendix***.

5.3 Evaluating the Hypotheses

The results obtained from testing the RLHF pipeline confirm the validation of two out of the three hypotheses proposed in the ***Hypotheses*** section, thereby demonstrating the viability of RLHF as a technique for developing AI game players. The following subsections delve into the evaluation of these hypotheses:

5.3.1 Evaluating Hypothesis 1

The first hypothesis posited that the agent trained using the RLHF pipeline would successfully learn the desired behavior from my preferences. In this case, the desired behavior was to track and shoot the Cacodemon[16]. Remarkably, the agent not only accomplished this behavior but also did so efficiently, using at most 2 bullets and as few frames as possible, resulting in an average reward of `70.34` from the ViZDoom environment.

5.3.2 Evaluating Hypothesis 2

The second hypothesis stated that the time needed for human feedback should not surpass `3 hours`. The results confirm this hypothesis, as the agent successfully learned the desired behavior in only `2.5 hours`. Out of this duration, `1.75 hours` were spent on my part to provide feedback, while the rest was devoted to reinforcement learning.

5.3.3 Evaluating Hypothesis 3

The third hypothesis, which suggested that the RLHF pipeline would expedite the learning process compared to the RL pipeline, turned out to be false. Both pipelines required the same number of steps, specifically `130000` steps, to teach the agent the desired behavior. While the RLHF pipeline didn't prove to be faster than the RL pipeline, it also didn't demonstrate any delays in learning.

5.4 Areas for Improvement

While the project has achieved success by validating two of the proposed hypotheses and coming close to the third, there remains significant room for enhancement and opportunities for future work.

Two vital aspects warrant improvement: the speed of learning and the quantity of human feedback collected. Although I followed the principles outlined in the `"Deep Reinforcement Learning from Human Feedback"` paper[7], my implementation of the training methodology differs considerably. The paper suggests advanced techniques such as employing an ensemble of reward predictors

and selecting trajectories based on ambiguity between predictions on trajectory pairs from various reward predictors in the ensemble. In contrast, the methodology I developed uses a single reward predictor and randomly pairs trajectories for collecting user preferences, collecting multiple preferences simultaneously to maintain stability. While my techniques have proven effective, the methods recommended in the paper are expected to yield significant performance improvements due to their asynchronous nature and a more substantial reduction in the number of human feedback instances required for successful agent training.

Additionally, one area for future work involves the integration of Long Short-Term Memory (LSTM) layers into the agent and reward predictors. LSTM layers are components within neural networks designed to handle sequences of data effectively, making them capable of capturing long-term dependencies and patterns in sequential data. While LSTM may not provide substantial advantages in simpler games like Doom, it could enable the agent and reward predictor to learn more advanced sequential strategies in complex games or more intricate levels within Doom.

6 Conclusion

The project's evaluations undeniably highlight its success in confirming two of the proposed hypotheses, solidifying its effectiveness. As outlined in the beginning of this report, the methodology used for training agents shows significant promise in the domain of AI player development for games. It presents a compelling and efficient alternative to conventional approaches like state machines.

However, it's essential to acknowledge that while this project serves as a compelling proof of concept, there is substantial room for refinement to enhance its suitability for game development environments and its long-term viability in the game development industry. The achievements attained thus far suggest a bright future for RLHF techniques, not only within Doom but also in more intricate game environments. The potential of this methodology to streamline AI player development is evident, providing a more adaptable and dynamic approach compared to traditional methods.

7 References

1. Russell, S., & Norvig, P. (2010). *Artificial intelligence: A modern approach* (3rd ed.). Pearson.
2. Shannon, C. E. (1950). XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, *41*(314), 256–275. https://doi.org/10.1080/14786445008521796
3. Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Driessche, G. van den, Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489. https://doi.org/10.1038/nature16961
4. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1312.5602
5. Edwards, G., Subianto, N., Englund, D., Goh, J. W., Coughran, N., Milton, Z., Mirnateghi, N., & Ali Shah, S. A. (2021). The role of machine learning in game development domain - a review of current trends and future directions. *2021 Digital Image Computing: Techniques and Applications (DICTA)*. https://doi.org/10.1109/dicta52665.2021.9647261
6. OpenAI, Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., Pinto, H. P. d. O., Raiman, J., Salimans, T., … Zhang, S. (2019). *Dota 2 with large scale deep reinforcement learning (v.1)*. https://doi.org/10.48550/ARXIV.1912.06680
7. Christiano, P., Leike, J., Brown, T. B., Martic, M., Legg, S., & Amodei, D. (2017). *Deep reinforcement learning from human preferences*. https://doi.org/10.48550/arxiv.1706.03741
8. OpenAI. (2022). *Introducing ChatGPT*. OpenAI. https://openai.com/blog/chatgpt
9. *IEEE xplore digital library*. (2017). IEEE; https://ieeexplore.ieee.org/Xplore/home.jsp.
10. *ResearchGate | share and discover research*. (2015). ResearchGate; https://www.researchgate.net.
11. *Nature*. (2019). Nature; https://www.nature.com/.
12. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv Preprint arXiv:1707.06347*.
13. Kim, M.-J., Kim, K.-J., Kim, S., & Dey, A. K. (2018). Performance evaluation gaps in a real-time strategy game between human and artificial intelligence players. *IEEE Access*, *6*, 13575–13586. https://doi.org/10.1109/access.2018.2800016
14. Kempka, M., Wydmuch, M., Runc, G., Toczek, J., & Jaśkowski, W. (2016). ViZDoom: A doom-based AI research platform for visual reinforcement learning. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1605.02097
15. *Doom slayer*. (n.d.). https://doom.fandom.com/wiki/Doom_Slayer.
16. *Cacodemon*. (n.d.). https://doom.fandom.com/wiki/Cacodemon.
17. Huang, S., Dossa, R. F. J., Raffin, A., Kanervisto, A., & Wang, W. (2022). *The 37 implementation details of proximal policy optimization* (S. Huang, R. F. J. Dossa, A. Raffin, A. Kanervisto, & W. Wang, Eds.). The ICLR Blog Track; https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/.
18. Farama-Foundation. (n.d.n.d.). *ViZDoom/gymnasium_wrapper/base_gymnasium_env.py*. https://github.com/Farama-Foundation/ViZDoom/blob/master/gymnasium/_wrapper/base/_gymnasium/_env.py
19. Costa-Luis, C. da. (n.d.n.d.). *Tqdm documentation*. https://tqdm.github.io
20. TensorFlow. (2019). *TensorBoard | TensorFlow*. TensorFlow. https://www.tensorflow.org/

tensorboard

8 Appendix

8.1 Code Repository

The code developed for this project has been made accessible to the graders by hosting it in a public GitHub repository. The repository can be accessed via this ***link***. It's important to note that the link points to the specific version of the repository that was tagged during the submission of this report. This was done to ensure the preservation of the exact files and code version used, enabling precise referencing and the ability to reproduce the implemented solution accurately.

8.2 Usage Guide

8.2.1 Setting up the Project Locally

The dependenices of the project have been setup as an Anaconda environment to make it easier for recreating the environement. At the moment, the project was only tested on Windows x64, but it will most likely work on Linux x86-64bit as well. It is recommended to anyone trying it locally to run it on an Windows x64 machine.

To begin, one has to first install the following applications on the machine:

- Anaconda
- Python 3.9.7

Once installed the environment can be setup using either one of the provided environment files, namely `x86_64-cpu-environment.yml` and `x86_64-cuda-environment.yml` depending upon the availabilty of NVIDIA Graphics Card. It must be noted that in order to use the GPU environment, the installed NVIDIA graphics card should support CUDA. To install the dependencies one of the following command should be executed from the root folder of the project folder in the terminal:

```
# To create CPU environment
conda env create --file x86_64-cpu-environment.yml

# To create CUDA environment
conda env create --file x86_64-cuda-environment.yml
```

Once the environment is setup, the environment can be activated by running the following command in the terminal.

```
conda activate doom-rlhf
```

8.2.2 Testing Trained Models

8.2.2.1 Prerequsites

The `play_doom_with_trained_agent.py` script requires the Anaconda Environment to be setup and activated. More information regarding the setup process can be found in the local setup section above.

8.2.2.2 How to Run

The `play_doom_with_trained_agent.py` script has been developed as a command line script and can simply be executed using the following command

```
python play_doom_with_trained_agent.py --env-cfg envs/vizdoom/scenarios/basic.cfg
 ↪  --agent-model baseline_doom_ppo_agent_model --episodes 50 --enable-gpu
```

Among the different arguments used in the above command, `--env-cfg` and `--agent-model` are the required arguments while the number of episodes is set to `50` by default, and the GPU is disabled by default. These parameters will be listed along with their description using the `--help` argument when executing the script and for reference they have also been listed in the table in **Table 5**.

Argument	Description
—env-cfg	Path to ViZDoom level config
—agent-model	Path to trained model
—episodes	Number of episodes to run the agent for
—enable-gpu	Flag for enabling GPU usage for the neural networks

Table 5: Table of arguments supported by the play_doom_with_trained_agent.py script.

8.2.2.3 Available Models

The following models can be used with the `play_doom_with_trained_agent.py` script:

- **RL Pipeline Trained Doom PPO Agent Model (Baseline Model):** `./final_models/baseline_doom_ppo_agent_model`
- **RLHF Pipeline Trained Doom PPO Agent Model:** `./final_models/ rlhf_pipeline_trained_doom_`

8.2.3 Running Reinforcement Learning Pipeline

8.2.3.1 Prerequsites

The `run_doom_rl_pipeline.py` script requires the Anaconda Environment to be setup and activated. More information regarding the setup process can be found in the local setup section above.

8.2.3.2 How to Run

Being implemented as a command line interface (CLI), the `run_doom_rl_pipeline.py` script supports the customization of hyper-parameters for the training process using command line arguments. The arguments supported by the script along with their default values can be viewed by passing the `--help` argument when running the script using python. These arguments are also listed and described in the table presented in **Table 6**.

Argument	Description
—env-cfg	Path to ViZDoom level config
—learning-rate	Learning rate of the RL agent
—total-timesteps	Total timesteps the agent should run for
—render-env	Flag for rendering one among the multiple environments

Argument	Description
—enable-gpu	Flag for enabling GPU usage for the neural networks
—track-stats	Track stats using Tensorboard
—num-envs	Number of environments to run simultaneously
—num-steps	Number of steps per environment for a training batch
—anneal-lr	Flag for annealing the learning rate of the RL agent
—enable-gae	Flag for enabling GAE calculation during RL agent training
—gamma	Coefficent that determines the influence of future rewards
—gae-lambda	The lambda value for GAE calculation during RL agent training
—num-minibatches	Number of mini-batches to use when training RL agent
—training-epochs	Number of epochs to train the RL agent with a training batch
—norm-adv	Flag to normalize calculated advantages during RL agent training
—clip-coef	The clipping coefficient of losses when training RL agent
—clip-vloss	Flag for clipping value loss when training RL agent
—entropy-coef	The coefficient for the entropy loss when training RL agent
—value-coef	The coefficient for the value loss when training RL agent
—max-grad-norm	Maximum normalized gradient value when training RL agent
—target-kl	Maximum KL divergence limit before stopping RL agent training

Table 6: Table of arguments supported by the run_doom_rl_pipeline.py script.

The script can be executed using python using the following command:

```
python run_doom_rl_pipeline.py --env-cfg envs/vizdoom/scenarios/basic.cfg
```

It must be noted that `--env-cfg` is a mandatory argument and the script requires it for functioning. The above example uses the VizDoom level config file located at `envs/vizdoom/scenarios/basic.cfg` since this is the level being used throughout the project. However, the other `'.cfg'` files located in the `envs/vizdoom/scenarios` directory will work fiine as well.

8.2.4 Running Reinforcement Learning with Human Feedback Pipeline

8.2.4.1 Prerequsites

The `run_doom_rl_pipeline.py` script requires the Anaconda Environment to be setup and activated. More information regarding the setup process can be found in the local setup section above. Once the environment is activated a Jupyter Notebook server can be started in the root directory of the project, which can then be used to run the pipeline.

8.2.4.2 How to Run

Being implemented as a Jupyter Notebook, the RLHF pipeline does not support arguments like the RL pipeline script and will require change to the params in the notebook itself for customization. However this process has been made easier by initializing the params for the pipeline using dictionaries in a separate code cell. There are three config dictionaries each corresponding to a different component of the pipeline. These configuration keys are listed in **Table 7**. Once the configuration of the pipeline is done, it can be executed by simply running all the cells of the notebook in order.

Config Key	Dictionary Name	Description
env_cfg	pipeline_args	Path to ViZDoom level config
total_timesteps	pipeline_args	Total timesteps the agent should run for
render_env	pipeline_args	Flag for rendering one among the multiple environments
enable_gpu	pipeline_args	Flag for enabling GPU usage for the neural networks
track_stats	pipeline_args	Track stats using Tensorboard
num_envs	pipeline_args	Number of environments to run simultaneously
env_replay_buffer_size	pipeline_args	Number of steps per environment for a training batch
enable_pre_training	pipeline_args	Flag for enabling the pre-training phase
num_pre_train_requests	pipeline_args	Number of preferences to collect during pre-training
num_trajectory_frames	pipeline_args	Number of frames in trajectory video
human_feedback_interval	pipeline_args	Number of agent training before collecting human feedback in training phase
learning_rate	agent_args	Learning rate of the RL agent
anneal_lr	agent_args	Flag for annealing the learning rate of the RL agent
enable_gae	agent_args	Flag for enabling GAE calculation during RL agent training
gamma	agent_args	Coefficent that determines the influence of future rewards
gae_lambda	agent_args	The lambda value for GAE calculation during RL agent training
num_minibatches	agent_args	Number of mini-batches to use when training RL agent
num_training_epochs	agent_args	Number of epochs to train the RL agent with a training batch
norm_adv	agent_args	Flag to normalize calculated advantages during RL agent training
clip_coef	agent_args	The clipping coefficient of losses when training RL agent
clip_vloss	agent_args	Flag for clipping value loss when training RL agent
entropy_coef	agent_args	The coefficient for the entropy loss when training RL agent

Config Key	Dictionary Name	Description
value_coef	agent_args	The coefficient for the value loss when training RL agent
max_grad_norm	agent_args	Maximum normalized gradient value when training RL agent
target_kl	agent_args	Maximum KL divergence limit before stopping RL agent training
learning_rate	reward_predictor_args	Learning rate for the Reward Predictor
hidden_layer_size	reward_predictor_args	Size of hidden layers in the reward predictor
num_training_epochs	reward_predictor_args	Number of epochs to train the reward predictor for with a training batch
model_path	reward_predictor_args	Path to the trained reward predictor model if a trained model should be used

Table 7: Table of arguments supported by the run_doom_rl_pipeline.py script.