COMS4047A

Reinforcement Learning

Assignment

WITS
UNIVERSITY

Taboka Chloe Dube - 2602515
Wendy Maboa - 2541693
Liam Brady - 2596852
Refiloe Mopeloa - 2333776

October 31, 2025

Introduction

Crafter is a procedurally generated 2D survival game designed as a benchmark for reinforcement learning research. It features a diverse set of tasks including resource gathering, tool crafting, creature combat, and achievement hunting, all while managing survival mechanics like hunger and health. This report will provide an overview of implementation of agents in Crafter using DQN and PPO algorithms.

In-class Algorithm: DQN

Out-of-class Algorithm: PPO

PPO (Proximal Policy Optimization) is a reinforcement learning algorithm that trains an agent by optimizing its decision-making policy. It works by collecting data through interactions with an environment and then using a clipped objective function to make stable updates to the policy. This approach is known for being more stable, efficient, and easier to implement than some other policy gradient methods (**schulman**).

Motivation

PPO (Proximal Policy Optimization) is a good choice for the Crafter environment because it provides a strong balance of stability, sample efficiency, and simplicity while effectively handling the environment's key challenges, such as sparse rewards, long-term reasoning, and procedural generation.

In addition these are the following theoretical benefits of PPO:

- **Stable Policy Updates:** The Crafter environment is complex and dynamic, where large, unconstrained policy updates could easily destabilize training and cause the agent to forget beneficial behaviors. PPO's clipping mechanism limits how much the policy can change at each step, ensuring stable and controlled learning.

- **Sample Efficiency:** Crafter involves many different achievements and complex interactions, meaning efficient use of experience is crucial. PPO is relatively sample-efficient because it can reuse collected data over multiple training epochs (mini-batches) without significant instability, unlike other on-policy methods that only use data once.

- **Facilitates Exploration:** The PPO objective often includes an entropy bonus term, which encourages the agent to explore different actions and strategies. This is vital in Crafter, which features wide, procedurally generated worlds and independent achievements that require broad exploration to discover all possibilities.

Hyperparamters used accross all agents

PPO Training: These hyperparameters were standardized for all agents so as to make sure the comparison is done based on the differing algorithms and not hyperparameter tuning.

- Learning rate: 3×10^{-4}
- Rollout steps (n_steps): 2,048
- Batch size: 64
- Epochs per update: 10
- Discount factor (γ): 0.99
- GAE lambda (λ): 0.95
- Clip range: 0.2
- Entropy coefficient: 0.01

Baseline Implementation

The baseline agent was implemented using the Stable-Baselines3 PPO algorithm. The initial hyperparameters were chosen to provide stable learning without aggressive updates, ensuring reproducibility. The agent at this point perceives the environment through single still images (frames) rather than continuous sequences, meaning it has only a limited view of the environment at each step.

Observations and Performance

The baseline PPO agent exhibited some exploratory behavior and moderate learning progress. However, its limited temporal awareness significantly constrained performance. The average episodic reward stabilized around **2.66**, with the maximum reward achieved being **6.1**. The following figures show the reward and survival distribution rates, as well as the achievement rates, over 1000 episodes:

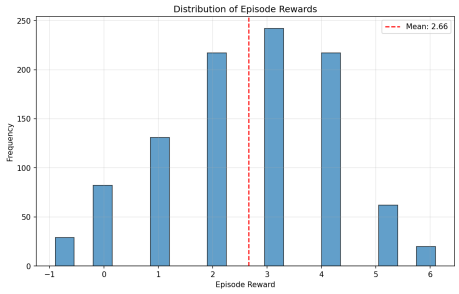

Figure 1: Reward distribution of PPO baseline

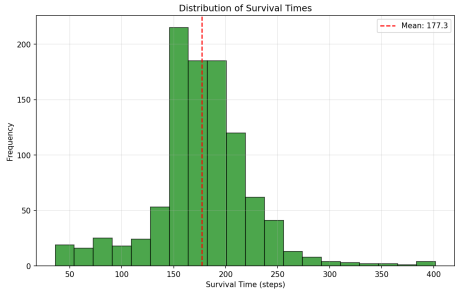

Figure 2: Survival distribution of PPO baseline

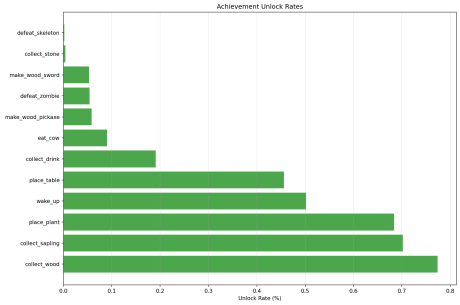

Figure 3: Achievement rate of PPO baseline

Identified areas of improvement

The following weaknesses were identified in the baseline model:

- **Lack of Memory:** The agent cannot recall previous states or actions, leading to suboptimal long-term decision-making.
- **Limited Exploration:** Without memory or intrinsic motivation, exploration remained shallow, resulting in repetitive behavior.

Improvement 1: Random Network Distillation

Background

Random Network Distillation (RND) is an exploration method that encourages reinforcement learning agents to explore novel states by providing intrinsic rewards based on prediction error. The method uses two neural networks: a target network that is randomly initialized and kept fixed, and a predictor network that is trained to predict the target network's outputs. The target network maps observations to feature vectors and remains frozen throughout training. The predictor network shares the same input-output architecture and is trained via gradient descent to minimize the mean squared error. The intrinsic reward is computed as the prediction error. This error is high for novel or infrequent states (where the predictor hasn't learned well) and low for frequently visited states (where the predictor has converged). (**burda2018exploration**)

Improvements

RND is introduced to address the limited exploration of the baseline implementation. Because Crafter is an environment with sparce rewards, traditional reinforcement learning methods may not work as they rely on dense rewards. We hypothesise that RND should improve the average reward of the agent due in part to its larger exploration window, allowing for more opportunities for rewards to be obtained.

Methodology

Architecture

Target Network (Fixed Random Features)
The target network is a randomly initialized convolutional neural network that remains fixed throughout training. It processes $64 \times 64 \times 3$ RGB observations through three convolutional layers:

- Conv1: 32 filters, 3×3 kernel, stride 2, padding 1 $\rightarrow$ output: $32 \times 32 \times 32$
- Conv2: 64 filters, 3×3 kernel, stride 2, padding 1 $\rightarrow$ output: $16 \times 16 \times 64$
- Conv3: 64 filters, 3×3 kernel, stride 1, padding 1 $\rightarrow$ output: $16 \times 16 \times 64$
- Flatten and fully connected layers: 16,384 $\rightarrow$ 512 features

All parameters are frozen after initialization to maintain consistent random target features.

Predictor Network (Learned Features)
The predictor network shares an identical architecture with the target network but is trained to predict the target network's output. The prediction error serves as the intrinsic reward signal, where high error indicates novel states.

Intrinsic Reward Computation
For each observed state s_t, the intrinsic reward is computed as:

$$r_{\text{intrinsic}}(s_t) = \|f_{\text{target}}(s_t) - f_{\text{predictor}}(s_t)\|^2 \quad (1)$$

where f_{target} and $f_{\text{predictor}}$ are the outputs of the target and predictor networks respectively. The intrinsic rewards are normalized using running mean and variance statistics to maintain stable learning dynamics.

Reward Integration
The total reward combines extrinsic (environment) and intrinsic rewards:

$$r_{\text{total}}(s_t) = r_{\text{extrinsic}}(s_t) + \lambda \cdot r_{\text{intrinsic}}(s_t) \quad (2)$$

where λ is the intrinsic reward coefficient (set to 1.0 by default). This is implemented via a custom Gymnasium wrapper (`RNDRewardWrapper`) that intercepts environment steps and augments rewards before they reach the PPO algorithm.

Training Procedure

- **RND Module Updates:** The predictor network is trained using mean squared error loss between its outputs and the fixed target network outputs. The optimizer uses Adam with learning rate 1×10^{-4}.
- **PPO Training:** The PPO agent is trained over 3×10^6 timesteps on the combined reward signal using standard hyperparameters and the following:
  - $intrinsic\_reward\_coef = 1.0$
  - $rnd\_learning\_rate = 1 \times 10^{-4}$
- **Evaluation Protocol:** Agent performance is evaluated every 10,000 training steps using deterministic policy rollouts over 5 episodes in a separate evaluation environment. Metrics including mean reward, standard deviation, minimum/maximum rewards, and episode lengths are logged to CSV files for analysis.

Exploration Mechanism
The RND method encourages exploration through the following mechanism:

1. Novel states produce high prediction errors, yielding high intrinsic rewards
2. The agent is incentivized to visit these high-reward states
3. As states become familiar, the predictor improves, reducing intrinsic rewards
4. The agent naturally shifts focus toward extrinsic task rewards

This approach addresses the sparse reward problem in Crafter by providing dense exploration bonuses while maintaining the original task structure.

Implementation Details
The implementation utilizes PyTorch for neural networks and Stable Baselines3 for the PPO algorithm. The RND networks use 3×3 convolutional kernels optimized for Crafter's 64×64 observation space. Running statistics for reward normalization use Welford's online algorithm to maintain numerical stability. All experiments use seed 42 for reproducibility, with the evaluation environment seeded at 142 to ensure different but deterministic trajectories.

Results

The agent was evaluated over 1000 episodes, giving an average reward of **3.3** and a maximum reward of **7.1**. This is an improvement of around **24%** and around **16%** over the average reward and maximum reward, respectively. This provides evidence that with RND, the average reward obtained improves from the baseline. However, this is a minor improvement and further improvements can be made by tuning the hyperparameters of the RND networks or running the training for more timesteps, which may give the agent more opportunities to explore novel states. Below are the figures showing the reward and survival distribution rates, as well as the achievement rates, over 1000 episodes:

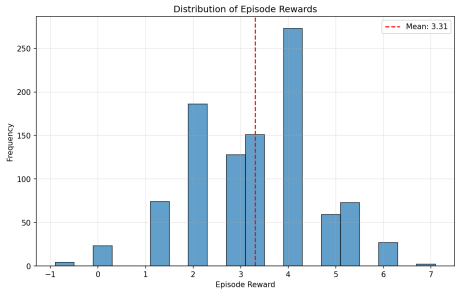

Figure 4: Reward distribution of PPO baseline

From this figure, the RND agent receives higher rewards more frequently than the baseline. This makes sense since by trying novel actions, there are more possibilities for rewards. However, a consequence of this is that the frequency of different rewards is not as uniform as the baseline.

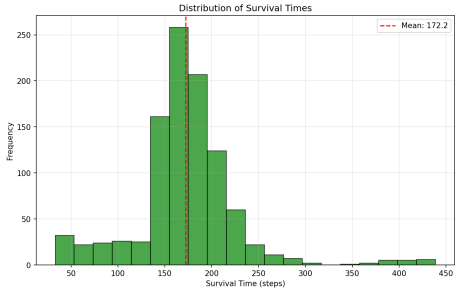

Figure 5: Survival distribution of PPO baseline

The RND figure in comparison to the baseline shows that the baseline tends to survive longer than the improvement. This can also be explained by the fact that the agent is more likely to explore - through more exploration comes more opportunities to enter states that lead the agent to die.

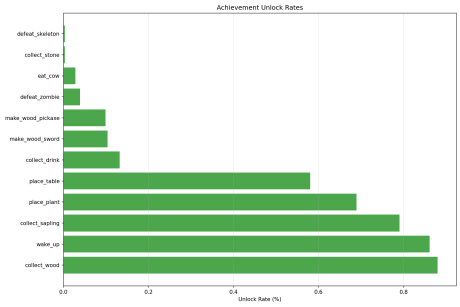

Figure 6: Achievement rate of PPO baseline

This figure shows that, in comparison to the baseline, the RND agent will have a less consistent spread of acheivement unlock rates since the agent is encouraged to choose novel states. While the baseline shows that over time the agent learns to repeat tasks that will provide it more rewards in the future, the RND has high achievement rates on tasks that won't offer much benefit to the agent, such as placing plants, or not eating and drinking as much.

Discussion

Strengths and weaknesses

The strengths of this implementation follow as a consequence of its approach. Because the agent is encouraged to explore more, the agent is more likely to unlock more acheivements, receive greater rewards, and learn more aboout the environment at a faster rate as compared to the baseline. However, its weaknesses are that it requires far more training time than the baseline since there is a large number of possible states to explore and similar to Monte Carlo exploration, requires a large numebr of episodes to get a good idea of the environment and its dynamics. Another weakness of the current RND implementation is the context window is very small, and thus even when the agent unlocks new achievements, these achievements and the rewards associated with them aren't carrying over in future.

Differences from Standard RND Implementation

This implementation differs from the standard Random Network Distillation approach proposed by **burda2018exploration** in several key aspects. The standard RND implementation uses two separate value functions—one for extrinsic returns and one for intrinsic returns—allowing different discount factors to be applied (typically $\gamma = 0.99$ for intrinsic and $\gamma = 0.999$ for extrinsic rewards), whereas this implementation uses a single combined value function with a uniform discount factor of 0.99. The CNN architecture has been simplified and adapted for Crafter's 64×64 observation space, using smaller 3×3 kernels throughout instead of the larger 8×8 and 4×4 kernels designed for 84×84 Atari frames, and omitting batch normalization and other regularization layers. Unlike standard RND which normalizes observations before feeding them to the RND networks and maintains separate statistics for intrinsic and extrinsic rewards, this implementation applies only basic running statistics normalization to intrinsic rewards using Welford's algorithm, with raw observations fed directly to the networks. The standard approach typically employs 128 or more parallel environments for efficient data collection and diverse training batches, updating the predictor network on

mini-batches from the rollout buffer with multiple gradient steps per rollout, whereas this implementation uses a single environment instance with per-step predictor updates, resulting in slower data collection and potentially noisier gradient estimates. Additionally, standard RND implementations include explicit reward clipping to prevent extreme values, separate advantage normalization for intrinsic and extrinsic components, and curiosity-driven exploration decay schedules, none of which are present in this implementation which uses a fixed intrinsic reward coefficient. Finally, the evaluation procedure differs: standard RND evaluates with intrinsic rewards completely disabled to measure true task performance without exploration bonuses, while this implementation uses deterministic policy evaluation without explicitly disabling the RND wrapper, making the separation between exploration and exploitation performance less clear.

Improvement 2: Long-Short Term Memory

Improvement 2 Implementation

To address the baseline and improvement 1 limitations in memory and exploration, we implemented a **Recurrent Proximal Policy Optimization (Recurrent PPO)** agent. This model extends PPO by incorporating a Long Short-Term Memory (LSTM) layer, enabling the policy to retain temporal context across time steps. This memory mechanism helps the agent handle the partially observable nature of the Crafter environment, where important state information may not be visible in a single frame. The following hyperparameters were added in addition to the previous implementations:

- **Value function coefficient (vf_coef):** 0.5
- **Maximum gradient norm (max_grad_norm):** 0.5
- **Reward shaping:** Enabled (`use_reward_shaping=True`)

Additonally, slight reward shaping was employed to accelerate convergence and guide exploration towards useful sub-goals. This ensured that intermediate achievements (such as crafting tools or collecting resources) contributed meaningful gradient signals during training.

```python
def step(self, action):
    obs, reward, terminated, truncated, info = self.env.step(action)

    current_achievements = obs['achievements']
    new_achievements = current_achievements - self.previous_achievements

    # Bonuses
    achievement_bonus = np.sum(new_achievements) * 5.0
    survival_bonus = 0.01
    exploration_bonus = 0.001
```

Figure 7: Reward Shaping implemented in RecurrentPPO agent

Observations and Performance

The introduction of temporal recurrence through an LSTM notably improved the agent's ability to integrate information across time steps. Compared to the baseline PPO model, which achieved a mean episodic reward of approximately 3.26 (maximum 7.1), the Recurrent PPO (R-PPO) demonstrated consistent performance gains.

Qualitative Observations

Behaviorally, the Recurrent PPO agent demonstrated:

- More coherent decision sequences across time, owing to the LSTM's memory retention.
- Improved persistence in long-horizon tasks such as crafting and navigation.
- Less tendency to repeat suboptimal exploration patterns seen in the baseline agent.

In addition, reward shaping contributed to smoother early learning and faster convergence during training, as the agent received structured feedback for intermediate achievements.

Discussion

The performance gain from incorporating temporal memory validates the hypothesis that Crafter's partial observability penalizes stateless architectures. By maintaining a hidden state, the Recurrent PPO agent effectively constructs an implicit representation of unobserved parts of the environment. This allows for more context-aware actions and ultimately leads to higher cumulative rewards.

While the improvement is evident, the results also reveal that the agent has not fully stabilized—indicated by the wide reward range. Further tuning of learning rate, clipping threshold, and recurrent hidden size could reduce variance and enhance consistency across episodes.

The following figures show the different reward and survival distribution rates over 100 episodes:

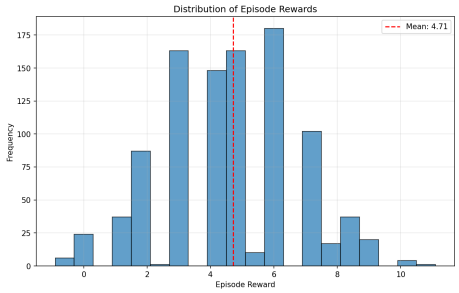

Figure 8: Reward distribution of PPO baseline

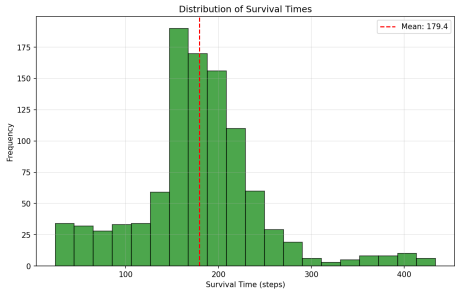

Figure 9: Survival distribution of PPO baseline

Model	Reward	Survival	Achievements	Geom Mean
Baseline	2.69±1.62	178.0±50.4	13/22	8.10%
Improvement 1	3.32±1.36	170.3±50.7	12/22	11.26%
Improvement 2 (LSTM)	4.68±2.08	177.6±65.8	16/22	9.92%

Figure 10: Comparison Metrics accross all PPO agents

An interesting observation from the experiments was that the Recurrent PPO (R-PPO) agent achieved a higher average reward (4.42) than the baseline PPO (3.26), despite exhibiting a lower overall survival rate. This contradiction can be explained by the agent's behavioral bias toward short-term, high-value actions.

The inclusion of memory through the LSTM layer allowed the agent to recall and exploit previously observed opportunities, such as nearby resources or enemies, resulting in higher reward density per timestep. However, this same decisiveness increased exposure to risk, reducing overall survival time.

Additionally, the use of reward shaping likely amplified this effect by reinforcing immediate subgoal completion (e.g., crafting, combat) rather than conservative, long-term survival strategies. Consequently, the R-PPO learned to act more efficiently but less cautiously, prioritizing cumulative reward over lifespan duration.

This highlights a fundamental reinforcement learning trade-off between *reward maximization* and *survival optimization*, emphasizing that longer episodes do not necessarily equate to better task performance.