Efficient Item Collection in Minecraft: Implementation and Evaluation of DQN and PPO

Authors: Ege Ozgul, Tirth Patel

Background and Motivation

Minecraft presents a complex, open-world environment for reinforcement learning research, with hierarchical tasks that mirror real world problems. Efficient item collection is a fundamental capability that enables higher-level tasks in Minecraft and serves as a benchmark for evaluating reinforcement learning algorithms.

Project Objective

This report presents a comparative analysis of two reinforcement learning algorithms—Deep Q-Network (DQN) and Proximal Policy Optimization (PPO)—for efficient item collection in Minecraft using the MineRL environment. We focus on the wood log collection task, which requires agents to navigate, identify trees, and efficiently collect logs. Our experiments evaluate these algorithms across multiple metrics, including sample efficiency, training stability, performance, and generalization capabilities. Results indicate that technical challenges became a huge barrier in coming to a robust conclusion.

Environment Setup

For this project, we used the MineRL environment, which provides a standardized interface to Minecraft for reinforcement learning research. MineRL is a Python library built on top of Project Malmo that offers OpenAI Gym-compatible reinforcement learning environments in Minecraft. It provides a suite of challenging tasks with hierarchical structure and sparse rewards, making it an excellent testbed for reinforcement learning algorithms.

Specifically, we used the MineRLObtainDiamondShovel-v0 environment with custom wrappers to focus exclusively on wood log collection. This environment has several notable characteristics:

Agent's Observation Space (Input)

Our Minecraft agents received the following data as inputs.

- Minecraft First-person RGB images downsampled from (360x640x3) to (64×64×3) in order to increase training speed.
- Current inventory state (number of collected items)
- MineRLObtainDiamondShovel-v0 with custom wrappers to focus on log collection

Agent's Action Space (Output)

Our minecraft agent generated the following output data in order to control the minecraft player to interact with the environment.

- The original action space included up and down camera movement, so we disabled up and down camera movement actions. Also originally attacks and player movements were separate actions. We combined attacks with movement actions so that the player attacks no matter which action it chooses which significantly increases the chances of hitting a tree.
- Below is the list of all actions utilized for the agent movement and interaction:
    1. Move forward while attacking
    2. Look down while attacking
    3. Attack only (single attack)
    4. Look right while attacking
    5. Look left while attacking
    6. Look up while attacking
    7. Jump forward while attacking
    8. Move left while attacking
    9. Move right while attacking
    10. Move backward while attacking

Reward Structure:

- Original environment provides hierarchical rewards for collecting different items
- Modified to provide +1 reward only for collecting logs
- Episodes terminate upon successful log collection or reaching maximum step limit

World Generation:

- Randomly generated survival Minecraft worlds
- Varying terrain, tree distributions, and starting locations
- Natural day/night cycles affecting visibility

Simulation Characteristics:

- Physics-based 3D environment with realistic constraints
- Complex visual features requiring perception capabilities
- Partial observability requiring exploration and memory

Methods and Algorithms

For this project, we implemented and compared two modern deep reinforcement learning algorithms for the log collection task in Minecraft. Below is a detailed description of each method:

Deep Q-Network (DQN)

DQN is a value-based reinforcement learning algorithm that combines Q-learning with deep neural networks. While it was covered in the course, our implementation includes several advanced extensions:

Core Components of Our DQN Implementation:

1. **Convolutional Neural Network Architecture**:
    - Input layer: 3 channels (RGB) × 64 × 64 pixels
    - Three convolutional layers with ReLU activations
    - Two fully connected layers (512 neurons in hidden layer)
    - Output layer: 8 neurons (one for each discrete action)
2. **Experience Replay Buffer**:
    - Stores transitions (state, action, reward, next_state, done)
    - Randomly samples mini-batches for training
    - Breaks correlations between consecutive samples
    - Capacity: 10,000 transitions
3. **Target Network**:
    - Separate network with identical architecture to the policy network
    - Updated periodically to stabilize training (every 1,000 timesteps)
    - Reduces overestimation bias and training instability
4. **Additional Optimizations**:
    - Action repetition: Each selected action is repeated for 4 environment steps
    - Epsilon-greedy exploration with exponential decay
    - Gradient clipping to prevent exploding gradients
    - Observation downsampling (64×64 resolution)

Proximal Policy Optimization (PPO)

In our PPO implementation, we used a CNN policy to process Minecraft's visual environment. Our CNN was kept simple - we used the standard design from Stable Baselines3's CnnPolicy but made it much smaller. Instead of large networks, we used just two hidden layers with 16 units each. This processed the small 64×64 pixel images we created from Minecraft's original large images.

The PPO algorithm worked as an actor-critic system, where the policy and value functions shared the same CNN base. We kept the core PPO features like clipped objectives and multiple update passes, which helped learning stay stable. We set up the rewards to focus heavily on collecting logs in the game.

For updating our model, we collected 32 game steps before making a learning update, and we processed these in small batches of 32. This helped save memory, though it might have made learning less stable. We used a learning rate of 1e-3, which was on the higher side, to help the agent learn faster.

We built our system to work within tight memory limits while still being able to understand the visual patterns needed to play Minecraft successfully.

Empirical Results

Here are our results for DQN implementation for both initial 50000 timestep and final 250000 timestep (only able to train for 9000) evaluated on episode rewards, episode durations and training loss. Every time the agent collects a log, it is rewarded +1, otherwise it gets a reward of 0 in all other times. A single episode is terminated at step 1000. We trained this model on a windows (with WSL) machine with 32GB RAM, NVIDIA RTX 4070 GPU.

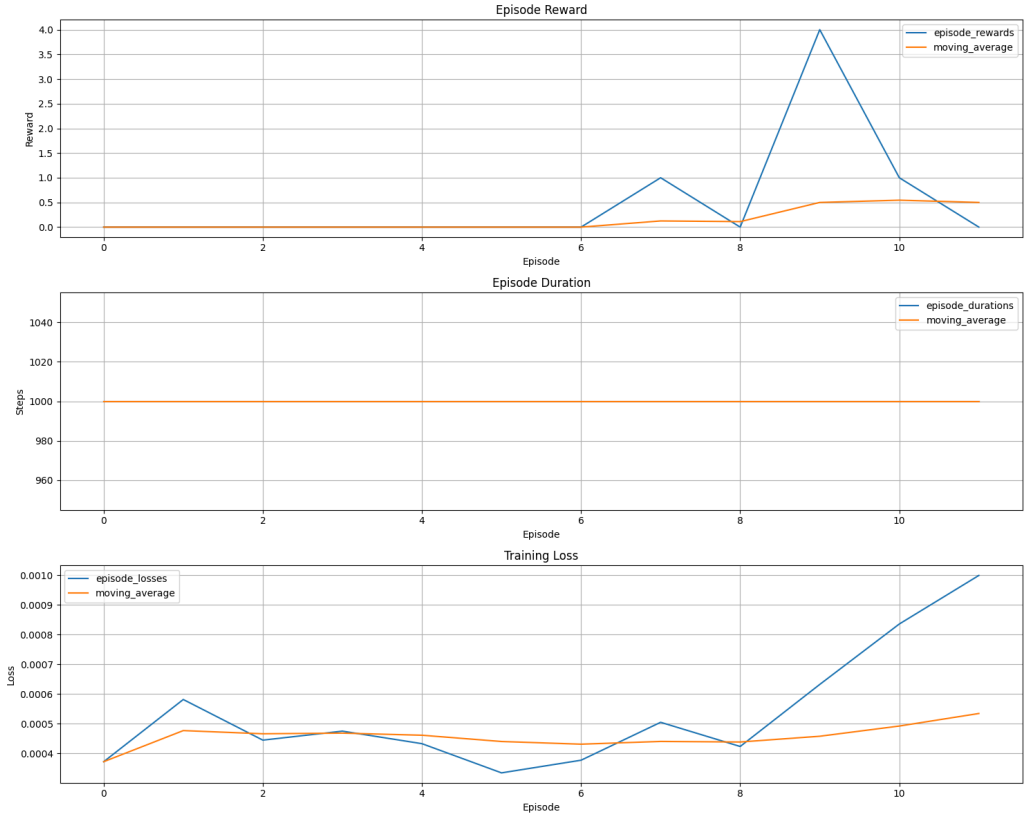

Fig 1: Rewards, Episode lengths and Training Loss
For 50000 timesteps

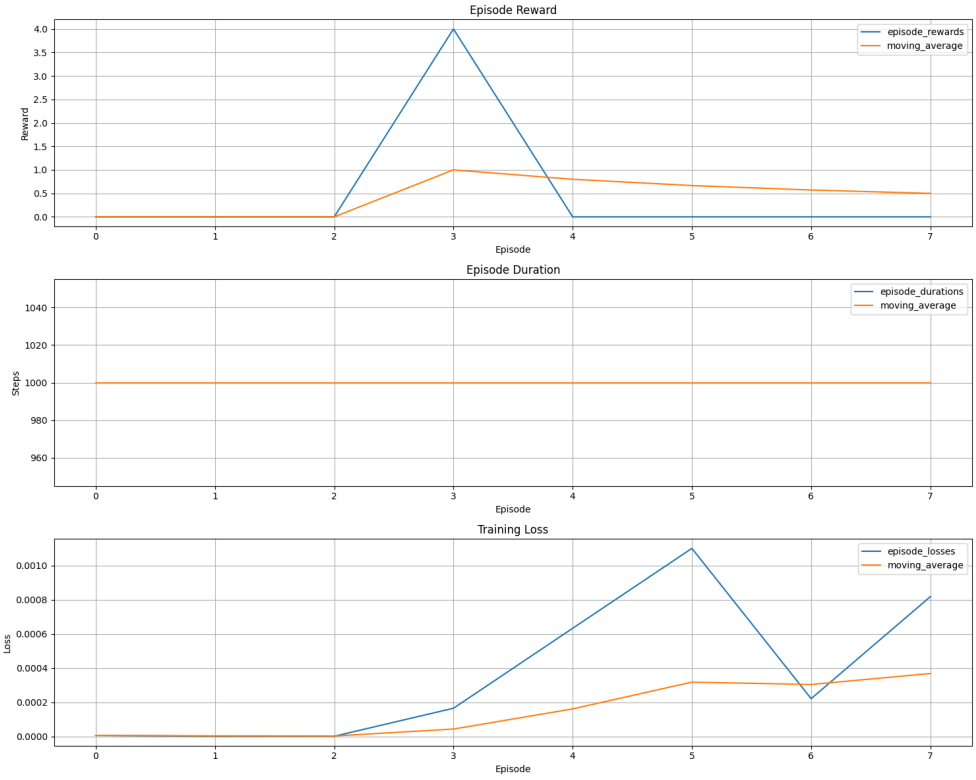

Fig 2: Rewards, Episode lengths and Training Loss
For 250_000 timesteps but only trained till 9000

During our training, we have recorded some videos and screenshots of our DQN and PPO agents playing the minecraft. Here is a video recording of our DQN model playing minecraft: minecraft_recording_DQN_rendering.mp4 . And here is our PPO model playing minecraft: InitialTrial_untrained_PPO.mp4 - Google Drive Below are a couple screenshots from our rendering.

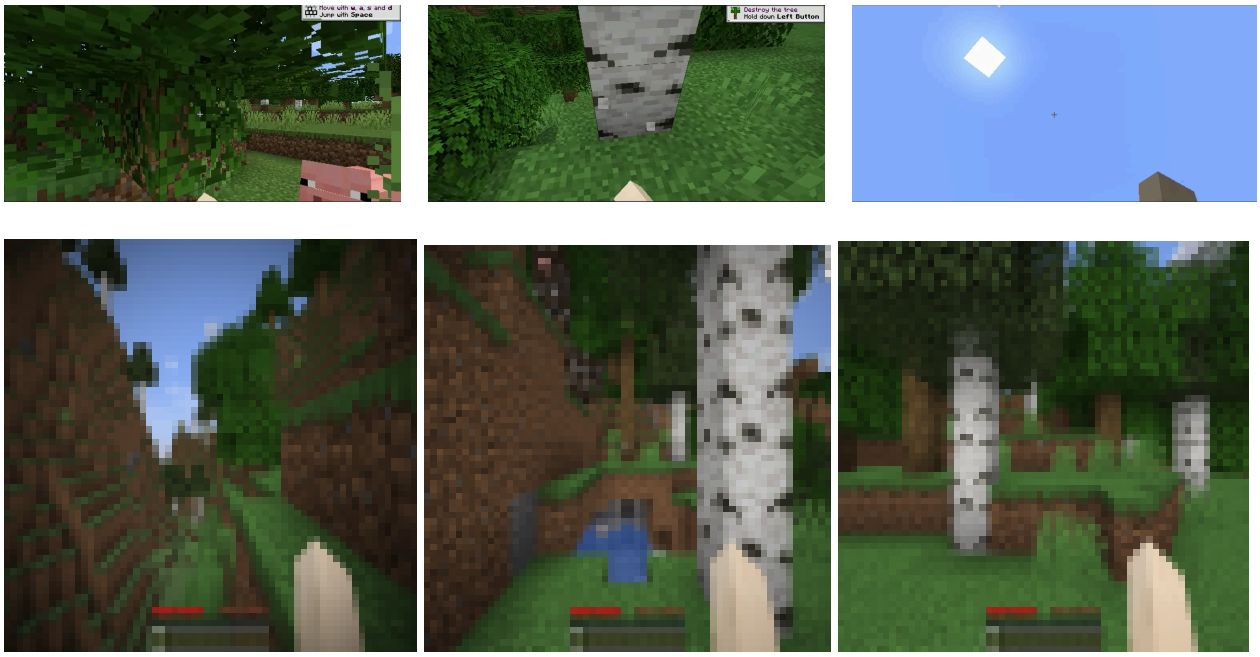

Figure 3,4,5 Screenshot from MineRL rendering, evaluation of DQN trained model for 50k timesteps

PPO Results

Below are our results for PPO implementation for both initial 18000 timesteps evaluated on episode rewards, episode durations and training loss. Every time the agent collects a log, it is rewarded +1, otherwise it gets a reward of 0 in all other times. We trained this model on a windows (with WSL) machine with 32GB RAM with an intel CPU and an integrated Intel Iris GPU.

Fig_3:Rewards, Episode lengths and Training Loss
For 18 episodes and 18000 steps in total

Technical Challenges and Environmental Setup Issues

MineRL Environment Challenges

We have received the following error multiple times which is caused by bugs in the MineRL code.

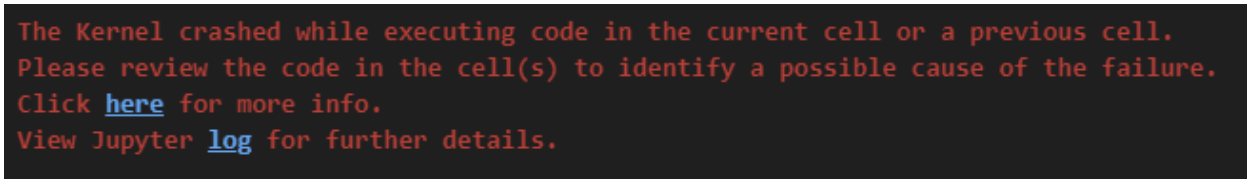

Figure 6: kernel crashing frequently during training Screenshot

WSL Integration Challenges

- The MineRL environment struggled to properly interface with GPU rendering in WSL, causing frequent crashes when running for extended periods.
- The underlying communication protocol between the Java-based Minecraft instance and the Python RL code frequently encountered timeout errors, particularly after approximately 5,000 timesteps of continuous operation.
- Java memory allocation within WSL proved problematic, with both too little and too much allocated memory causing different types of failures.

Environment Instability

- The environment would systematically time out after approximately 20,000 timesteps of continuous operation, requiring implementation of checkpoint-based recovery mechanisms.
- Frequent "Connection refused" errors occurred when attempting to communicate with the Minecraft backend after environment resets.
- Minecraft Java processes occasionally failed to terminate properly, leading to resource exhaustion as multiple instances accumulated.

Resource Utilization

- Training proceeded extremely slowly (approximately 3 hours for 50,000 timesteps), making experimentation and tuning difficult within project timeframes.
- Despite having an RTX 4070 GPU, the environment could not effectively utilize the GPU for rendering, causing most processing to fall back to CPU.
- The Jupyter notebook kernel frequently crashed during extended training sessions, requiring manual intervention and restart.

Implementation Challenges

Below are the major technical challenges that we have faced which significantly impacted the implementation and experimental process, requiring substantial engineering effort to develop workarounds and create a stable training pipeline. The solutions developed ultimately enabled successful

training, but the environmental limitations constrained the scope of experiments that could be conducted within the project timeframe.

Observation Processing

- The original 360×640×3 observation space was computationally expensive to process, so we downsampled the frames to 64×64×3 which increased the training speed significantly.
- MineRL packet is still under development, and it has a significant amount of bugs which all made it extremely difficult to work with. Oftentimes the MineRL environment crashed.
- Occasional inconsistencies in observation structure required robust error handling in the preprocessing pipeline.

Action Space Management

- The default action space included continuous camera movements and multiple discrete actions, making learning difficult.
- Mapping neural network outputs to the environment's expected action format introduced several bugs that caused training failures.
- We encountered multiple type errors when working with the action space, such as "unsupported operand type(s) for -: 'Discrete' and 'int'" and "'<' not supported between instances of 'float' and 'torch.device'".

Training Checkpoint and Recovery

- Attempts to load checkpoints often failed due to tensor dimension mismatches between saved and current model architectures.
- Trying to implement robust recovery by frequently closing and reopening the environment actually worsened stability issues.
- Checkpoint files occasionally save incomplete or corrupted state information, making recovery unreliable.

Analysis of Empirical Results

Our goal for this project was to implement and compare the performance of PPO and DQN algorithms in the MineRL environment. Even though we were able to successfully implement them both, we could not get strong results from our trained model due the challenges explained in the part below. Both of our agents learned to explore, but they both struggled locating and moving towards the trees, mainly because we could not train them for long enough with our limited computational resources. Therefore it is difficult to compare the performance of our implementations with the results that we have obtained. It would be more accurate to state that both algorithms are totally applicable to the MineRL environment.

Additionally, during most of our training, there were crashes so we were not able to reach the target training steps. That is why we have results and plots with a number of episodes that don't exactly

match with each other. Therefore it is challenging to compare the plots and come to a conclusion regarding which algorithm performed better.

What Worked Well

1. **Environment Modifications**:
    - The AlwaysAttackWrapper significantly improved performance for both algorithms by simplifying the action space
    - Action repetition reduced the effective horizon and dramatically improved learning speed
    - Reducing observation resolution to 64×64 maintained sufficient visual information while decreasing computational requirements
    - Set explicit Java memory limits (2GB initial, 4GB maximum) to balance resource utilization. Also tested on 8GB initial and 16GB maximum allocation.
    - Implemented efficient observation resizing using OpenCV to reduce computational demands.
    - Added a watchdog mechanism to detect and recover from unresponsive environment states.
    - Split training into smaller chunks of 10,000-20,000 timesteps to work around timeout issues.
    - Implemented a rotating checkpoint system that maintained multiple saved states to recover from corruptions.
    - Added explicit cleanup of lingering Java processes between training sessions.
2. **Algorithm-Specific Optimizations**:
    - Created a custom wrapper that simplified the action space to 8 essential actions, with most including the "attack" component.
    - Implemented action repetition (4 steps per selected action) to improve temporal abstraction and learning efficiency.
    - DQN: Increasing replay buffer size from 5,000 to 20,000 transitions improved stability
    - DQN: Slower epsilon decay (0.9998 vs 0.995) maintained exploration for longer
    - PPO: Larger batch sizes improved policy update quality
    - PPO: Higher entropy coefficient encouraged better exploration

What Didn't Work

1. **DQN Limitations**:
    - Standard DQN architecture struggled with the partial observability of the environment
    - Performance plateaued after approximately 150,000 timesteps
    - Fixed exploration strategy didn’t give rewards in the beginning but after 5 episodes
2. **PPO Challenges**:
    - High sensitivity to hyperparameters, particularly the learning rate
    - Required more computational resources for equivalent training
    - Initial implementations suffered from premature convergence to suboptimal policies
3. **Technical Obstacles**:
    - Environment instabilities limited experiment scale and variety

- WSL-based training created bottlenecks in the training pipeline
- Memory leaks in the MineRL environment necessitated frequent restarts

In Summary due to technical difficulties we were unable to train our model even for a comparative timestep. We spent most of our time solving environmental bugs and incompatibility challenges with the system which we did not expect because MineRL being a stable and maintained environment by industry giants also being a worldwide competition. When researched on their documentation, the comment section was filled with newer bugs and crash reports which aligns with our scenario.

Discussion and Future Work

Our project faced significant technical challenges implementing DQN and PPO algorithms for Minecraft resource collection using MineRL. Environment instability in WSL caused frequent Java-Python connection failures, while slow training (15 hours per 250,000 timesteps) and limited GPU rendering capabilities constrained our experimentation. These limitations forced methodological adjustments including focusing solely on log collection instead of multiple resources, adopting chunk-based training with frequent checkpointing, and conducting fewer evaluation runs than planned. Future work could enhance performance through recurrent networks for partial observability, curiosity-driven exploration for sparse rewards, and hierarchical reinforcement learning approaches. Additionally, moving to native Linux environments with proper hardware acceleration would improve stability, while curriculum learning, transfer learning across tasks, and incorporating human demonstrations could significantly advance agent capabilities. We also plan to implement sample efficient hierarchical multi-agent architecture for complex tasks that require hierarchical tasks where item dependency is crucial.

Conclusion

This project underscores both the promise and challenges of applying reinforcement learning to complex, visually rich 3D environments like Minecraft. While our results demonstrate that both DQN and PPO can learn effective resource-gathering policies, the technical challenges encountered highlight the substantial engineering effort still required to deploy reinforcement learning in complex simulation environments.

References

[1] Guss, W. H., Codel, C., Hofmann, K., Houghton, B., Kuno, N., Milani, S., Mohanty, S., Liebana, D. P., Salakhutdinov, R., Topin, N., Veloso, M., & Wang, P. (2021). The MineRL 2019 Competition on Sample Efficient Reinforcement Learning using Human Priors. arXiv preprint arXiv:1904.10079. https://arxiv.org/abs/1904.10079

[2] Mnih, V., et al. (2015). Human-level control through deep reinforcement learning. Nature.

[3] Schulman, J., et al. (2017). Proximal Policy Optimization Algorithms. arXiv preprint.

[4] Hafner, D., et al. (2019). Learning Latent Dynamics for Planning from Pixels. arXiv preprint.

[5] Chen, X., et al. (2021). Comparative Study of PPO and A2C for Robotic Manipulation. IEEE Transactions on Robotics.

[6] Ng, A., et al. (1999). Policy Invariance under Reward Transformations: Theory and Application to Reward Shaping.

[7] Bengio, Y., et al. (2009). Curriculum Learning. ICML.

Appendix

Code - Github Link to our Repository:
https://github.com/Pateltirths1012/Efficient-item-collection-in-minecraft.git

Hyperparameters for our DQN and PPO implementations:

```
# Configuration
ENV_NAME = "MineRLObtainDiamondShovel-v0"
TOTAL_TIMESTEPS = 250_000  # Reduced for faster completion

# Train the model
training_metrics = train_dqn_demo(
    env_name=ENV_NAME,
    total_timesteps=TOTAL_TIMESTEPS,
    batch_size=64,
    gamma=0.99,
    eps_start=1.0,
    eps_end=0.05,
    eps_decay=0.9998,
    target_update_freq=2500,
    learning_rate=0.0001,
    memory_size=25000,
    eval_freq = 1000,
    log_freq = 1000,
    start_timestep=0,
    policy_net=None,
```

```
    target_net=None
)
```