# Eco Civilization MDP: A Multi-Agent Environment for Studying Sustainable Growth

**Ege Cakar**
ecakar@college.harvard.edu

**Anne Mykland**
amykland@college.harvard.edu

**Zoe Wu**
zoewu@college.harvard.edu

## 1 Abstract

Our project implements Multi-Agent Proximal Policy Optimization (MAPPO) in the novel setting of a modified Civilization game. In Civilization, players build an empire that can stand the test of time by making strategic decisions about city-building, technology, diplomacy, military conquest, culture, and more. We adapt this game by incorporating environmental penalties, encouraging players to both compete and cooperate to grow their country without furthering climate change. We use MAPPO to solve this game, as previous research shows the algorithm achieves strong performance across various cooperative multi-agent benchmarks. Additionally, we study how the behavior of our agents is influenced by the weight of environmental penalties, and we consider the game strategies that prove successful in worlds that are more/less environmentally conscious.

## 2 The Adapted Civilization Game

### 2.1 Brief Overview of the Game

A maximum of six different agents compete and collaborate to build their own countries across a map while minimizing environmental impacts. The map contains resources, materials and water, which are initialized randomly throughout the map.

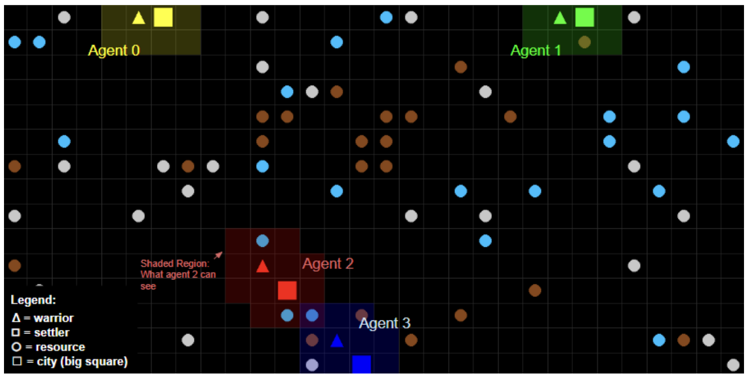


Figure 1: Initial map rendered as part of game

Players can buy or produce units to help them take advantage of resources, build their own cities, and attack other cities. In overview, the following actions can be taken.

1. Expending resources to build a project in a city a country owns: A given project takes several consecutive turns to finish, and a country will continue to engage in actions within the second category while the project is running. At any time, a city can only work on one project. A project may encapsulate working on buildings for the city, or creating units such as military and explorers. Building up a city generates greater economic growth, measured in GDP. In order to capture the environmental impact of city building, two separate kinds of projects exists– environmentally friendly and destructive projects. Destructive projects provide a higher GDP boost to the city but receive higher penalties, which contribute to their negative environmental impact. They also destroy a resource, material or water tile at random and delete it from the map – which is a separate "penalty" for the players, since resources, materials and water within borders contribute to GDP.
2. Utilizing units, which include military and explorers: Soldiers can move to any adjacent squares or choose to attack cities they know have resources, but cannot do both in a single move. Explorers can move to only unclaimed adjacent squares (squares not owned by any country) or choose to settle a city, but similarly cannot do both in a single move. Once an explorer settles a city, the player controls a two square radius around the city center. We incorporate the exploration-exploitation trade-off through the moves of soldiers, which represent exploitation of existing land and cities with resources, and explorers, which represent exploration of unclaimed and unseen land.

## 2.2 States and Actions

Our state is comprised of a dictionary of four tensors:

1. The map: A one-hot encoded 3-D tensor of size *map width* $\times$ *map height* $\times$ *number of specifications for map coordinate*. Within the third dimension of our vector, we encode information about the ownership of the tile coordinate specified by the first two dimensions, the type of unit it contains (city, warrior and settler) as well as the resources, material and water that the tile contains. Thus if we let $n =$ number of agents in the game, the third dimension has size $n + 3n + 3$, where the first $n$ positions encode ownership, the next $3n$ positions encode units for each agent and the last 3 encode the presence or absence of resources.

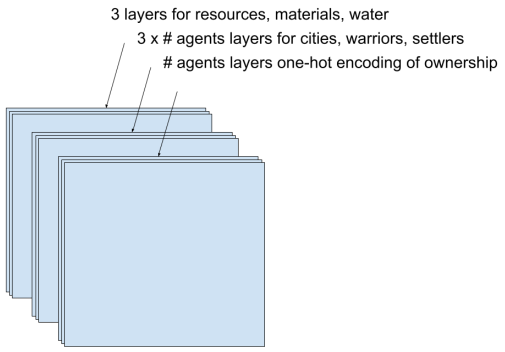

3 layers for resources, materials, water
3 x # agents layers for cities, warriors, settlers
# agents layers one-hot encoding of ownership

Figure 2: Encoded information about the map

2. Units: A tensor of size *max number of units*$\times 4$. There are two types of units – warriors and settlers. The Unit tensor provides the x and y coordinates of the unit, the health of the unit and the type of the unit for every unit belonging to the agent.
3. Cities: A tensor of size *max number of cities*$\times 6$. The City tensor provides the x and y coordinates of the city, the health of the city, as well as the resources, materials and water in the city, for every city belonging to the agent.
4. Money (scalar). This is calculated based on GDP.

There are seven different actions that an agent can take. Agents can move units, attack units, found cities, assign projects, buy warriors and buy settlers. If moving a unit, attacking a city or founding a city, the id of the unit used must be specified. When moving and attacking units, direction of movement must be specified. To assign a project, city and project id must be specified. Below is an example of a potential action.

```
action = {
    "action_type": self.MOVE_UNIT,
    "unit_id": 0,
    "direction": 1,  # 0: up, 1: right, 2: down, 3: left
    "city_id": 0,    # Ignored for MOVE_UNIT
    "project_id": 0  # Ignored for MOVE_UNIT
}
```

## 2.3 Rewards

The goal of the game is for countries to maximize country growth while minimizing environmental impacts. As such, we want to reward actions that aid growth, but also minimize the environmental impact. Each action taken has some environmental score associated with it, where the environmental score accumulated is taken out from the reward of the agent. Here is our current formulation of the reward, which has slightly changed from the one within our proposal:

$R(s, a, s\prime) = k_1 \cdot P_{\text{progress}}(s, s\prime) + k_2 \cdot P_{\text{completion}}(s, s\prime) + k_3 \cdot C_{\text{tiles}}(s, s\prime) + k_4 \cdot C_{\text{cities}}(s, s\prime) - k_5 \cdot L_{\text{cities}}(s, s\prime) + k_6 \cdot C_{\text{units}}(s, s\prime) - k_7 \cdot L_{\text{units}}(s, s\prime) + k_8 \cdot \Delta_{\text{GDP}}(s, s\prime) + k_9 \cdot \Delta_{\text{Energy}}(s, s\prime) + k_{10} \cdot C_{\text{resources}}(s, s\prime) - \gamma \cdot E_{\text{impact}}(s, s\prime) + K_{\text{entropy}} \cdot H(\pi_\theta) - \beta \cdot \text{Stalling}$

Where $P_{\text{progress}}$ is the number of ongoing projects, $P_{\text{completion}}$ is the number of completed projects, $C_{\text{tiles}}$ is the number of explored tiles, $C_{\text{cities}}$ is the number of captured cities, $L_{\text{cities}}$ is the number of cities lost, $C_{\text{units}}$ is the number of enemy units eliminated, $L_{\text{units}}$ is the number of units lost, $\Delta_{\text{GDP}}$ is the change in GDP, $\Delta_{\text{Energy}}$ is the change in energy output, $C_{\text{resources}}$ is the number of resources gained control over, and $E_{\text{impact}}$ is the environmental impact per turn. The last two terms are for entropy and stalling, discussed below. Values $k_1, \ldots, k_{10}$ and $\gamma$ are constants, where $\gamma$ is its own "environmental impact" constant we will change to see its effects on policies.

## 2.4 Entropy

One of the key challenges in traditional RL setups is the exploration-exploitation trade-off. Our civilization game rewards sparsely: given the action space, rewards don't come until after many steps, and thus algorithms that do not sufficiently explore new states may fail to learn anything meaningful. Instead, algorithms tend to stick to their "favorite" action.

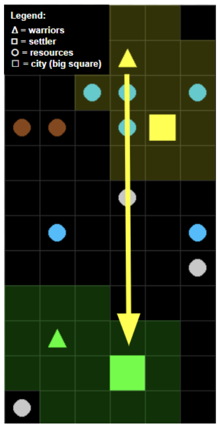


Figure 3: Yellow can move its units in four directions every move, but to capture a city and earn a reward, it has to consistently choose to move its warrior down

To encourage exploration, we incorporate an entropy bonus to the reward function. In sparse reward environments, the RL agent's objective is to maximize the expected cumulative reward:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t R(s_t, a_t)\right]$$

where:

- $\pi_\theta(a|s)$: Policy parameterized by $\theta$, mapping states $s$ to actions $a$.
- $\gamma$: Discount factor, weighting the importance of future rewards.
- $R(s_t, a_t)$: Reward function, which may be zero for most state-action pairs in sparse reward settings.

In sparse reward environments:$R(s_t, a_t) \approx 0$ for most $t$, making the gradient signal for policy updates weak or noisy. The agent may prematurely exploit suboptimal actions $a$ with observed non-zero rewards, failing to discover actions leading to better long-term outcomes.

To address this, exploration is encouraged by adding an entropy regularization term to the objective:

$$J_{\text{entropy}}(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t R(s_t, a_t)\right] + K_{\text{entropy}} H(\pi_\theta)$$

where:

- $H(\pi_\theta) = -\sum_a \pi_\theta(a|s) \log \pi_\theta(a|s)$: Shannon entropy of the policy's action distribution.
- $K_{\text{entropy}}$: Entropy coefficient, controlling the trade-off between exploration and exploitation.

This formula sums over the negative product of probabilities and their logarithms, capturing the uncertainty or randomness of the action distribution. A small offset is used to ensure numerical stability and avoid issues with logarithms of zero. This computation was adapted to work with our complex, hierarchical action distribution produced by the actor network, ensuring that the policy remains exploratory while refining its performance over time.[1]

### 2.5 Stalling

We define stalling

$$\text{Stalling reward}_t = \beta \cdot 1.0 \cdot \mathbf{1}(s = s')$$

where states remain the same after an agent takes an action. Discouraging stalling pushes agents to continuously interact with new states or achieve meaningful progress. When agents are penalized for remaining in the same state or performing repetitive actions that fail to advance the environment, they are effectively incentivized to seek alternative strategies and explore uncharted areas of the state space. This mechanism prevents the agent from falling into local optima where it might exploit short-term rewards without seeking potentially higher rewards elsewhere. By penalizing stalling, the learning process shifts toward prioritizing diverse experiences, encouraging the agent to discover novel actions and environments that may yield greater cumulative rewards. This, in turn, leads to a richer dataset for learning and improves the robustness and adaptability of the agent's policy, particularly in environments with sparse rewards where exploration is essential for long-term success. Another reason for adding the punishment for stalling is knowing the structure of the game – we know that it's better on average to do things, even if it is to just move units, than to not do things. That's why we push for the agents to be active.

## 3 Multi-Agent Proximal Policy Optimization

### 3.1 Proximal Policy Optimization (PPO) in a Single-Agent Setting

In a single-agent reinforcement learning setting, Proximal Policy Optimization (PPO) aims to improve upon the policy gradient approach by taking more stable and constrained policy updates. Instead of taking large gradient steps that could drastically change the policy, PPO ensures that new policies stay close to the old ones within a specified trust region. Concretely, PPO uses a clipped objective function that restricts the ratio of the new policy probability to the old policy probability, preventing excessive deviation that might destabilize learning.

When updating parameters, consider a policy parameterized by $\theta$. After collecting a set of trajectories using the current policy $\pi_{\theta_{\text{old}}}$, the algorithm forms an objective function that compares the new policy $\pi_\theta$ to the old one. The ratio of probabilities of taking an action $a$ in state $o$ under the new and old policies is defined as:

$$r_\theta(o, a) = \frac{\pi_\theta(a \mid o)}{\pi_{\text{old}}(a \mid o)}.$$

PPO modifies the policy gradient objective by clipping this ratio within a range $[1-\epsilon, 1+\epsilon]$ to prevent large updates. By optimizing this clipped objective using gradient ascent, we improve the policy steadily and avoid catastrophic policy updates. The parameters $\theta$ are updated by taking a gradient step on this clipped objective until convergence criteria are met.

## 3.2 Actor-Critic Framework for Multi-Agent Settings

Extending PPO to a multi-agent scenario, we adopt an actor-critic framework where each agent maintains an actor (policy) network and a critic (value) network. Each agent has its own actor network that determines the action probabilities given the agent's local observations. All agents share a single critic network that estimates the value function to provide a baseline and reduce variance in policy gradient updates. By incorporating both an actor and critic, agents learn to both cooperate and compete.

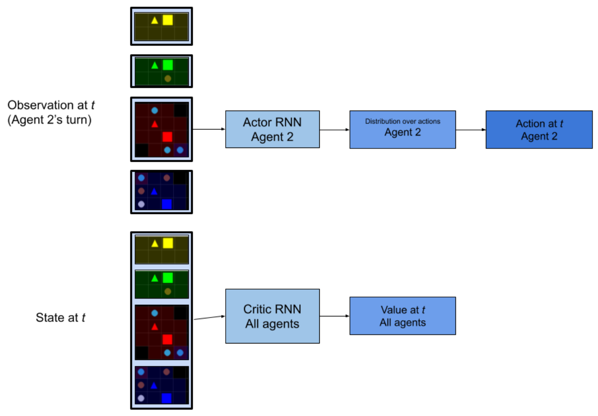


Figure 4: Inputs and ouputs of RNN's. Hidden states are also inputted and outputted for each RNN, but omitted in this diagram for simplicity. The actor updates the policy distribution in the direction suggested by the critic

For each agent $a$, at timestep $t$, we define:

$$p_t^{(a)}, h_{t,\pi}^{(a)} = \pi\left(o_t^{(a)}, h_{t-1,\pi}^{(a)}; \theta\right),$$

where $p_t^{(a)}$ is the action probability distribution given the agent's observation $o_t^{(a)}$ and the previous hidden state of the policy network $h_{t-1,\pi}^{(a)}$. The hidden state $h_{t,\pi}^{(a)}$ captures temporal dependencies in the agent's policy. To obtain an action, we sample:

$$u_t^{(a)} \sim p_t^{(a)}.$$

We adapt this MAPPO agent framework to incorporate a hierarchical action landscape. We first apply a multi-layer gated recurrent unit (GRU) RNN to our input sequence: observations and hidden states at time $t$. We then apply independent affine linear transformation to GRU output, one linear transformation per action. We apply a softmax to the result to create a distribution for each type of action.

Simultaneously, the critic network, parameterized by $\phi$, provides a value estimate:

$$v_t^{(a)}, h_{t,V}^{(a)} = V\left(s_t^{(a)}, h_{t-1,V}^{(a)}; \phi\right),$$

where $s_t^{(a)}$ is the agent's state input to the value function. The hidden state $h_{t,V}^{(a)}$ allows the critic to incorporate temporal information. During data collection, these tuples $(o_t^{(a)}, u_t^{(a)}, p_t^{(a)}, v_t^{(a)})$ and their hidden states are stored in trajectories. After an episode or a training interval, these trajectories are used to compute the policy gradients and value function updates.

We chose to have the same critic network across all agents, as states are global: the state at a given time step aggregates all information among agents.

### 3.3 Mini-Batch RNN Updates

When using recurrent neural networks (RNNs) in the actor or critic, it is crucial to maintain proper hidden states during training. Training with full trajectories would be computationally expensive and might not generalize well. Instead, we shuffle and sample mini-batches of trajectories. Random mini-batches help decorrelate samples, reducing overfitting and improving the stability of training. By resetting and passing the appropriate hidden states for each mini-batch segment, we ensure that the RNN captures temporal dependencies while still benefiting from stochastic gradient-based optimization techniques. This maintains efficiency and stable learning updates throughout the entire training process.

### 3.4 Loss Equations

#### 3.4.1 Actor Loss

The actor loss function in PPO uses the clipped probability ratio to limit step size:

$$L(\theta) = \frac{1}{Bn}\sum_{i=1}^{B}\sum_{k=1}^{n}\min\left(r_{\theta,i}^{(k)}A_i^{(k)},\ \text{clip}\big(r_{\theta,i}^{(k)}, 1-\epsilon, 1+\epsilon\big)A_i^{(k)}\right),$$

where

$$r_{\theta,i}^{(k)} = \frac{\pi_\theta\big(a_i^{(k)} \mid o_i^{(k)}\big)}{\pi_{\text{old}}\big(a_i^{(k)} \mid o_i^{(k)}\big)}.$$

In this equation, $A_i^{(k)}$ denotes the advantage function at the $k$th step of the $i$th batch element. The advantage $A_i^{(k)}$ is often computed using the Generalized Advantage Estimator (GAE), which provides a low-variance, low-bias advantage estimate. The objective encourages increasing the probability of actions that lead to higher-than-expected returns and decreasing the probability of actions that are worse than expected. The clipping function ensures that we do not deviate too far from the old policy, stabilizing updates.

#### 3.4.2 Critic Loss

The critic, parameterized by $\phi$, is trained to predict the value function $V_\phi(s)$ that approximates the expected return. PPO uses a clipped value loss similarly to the policy objective:

$$L(\phi) = \frac{1}{Bn}\sum_{i=1}^{B}\sum_{k=1}^{n}\max\left(\big(V_\phi(s_i^{(k)}) - \hat{R}_i\big)^2,\ \big(\text{clip}(V_\phi(s_i^{(k)}), V_{\phi_{\text{old}}}(s_i^{(k)}) - \epsilon, V_{\phi_{\text{old}}}(s_i^{(k)}) + \epsilon) - \hat{R}_i\big)^2\right),$$

where $\hat{R}_i$ is the discounted reward-to-go for the $i$th trajectory. By clipping the predicted value function around the old value estimate, we avoid large and destabilizing updates to the value function. This stabilizes training and keeps the critic's predictions consistent over consecutive updates.

Here, $B$ is the batch size and $n$ is the number of agents (or parallel trajectory segments). Both the actor and critic losses are averaged over all agents and sampled trajectory segments, ensuring stable and robust learning updates.

### 3.5 Choice of Adam Optimizer

The Adam optimizer is chosen due to its adaptive step size and momentum-based updates. Adam computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients. This property allows for more stable and faster convergence in practice, as it naturally adjusts the learning rates to the scale of the gradients. In the context of PPO, where both policy and value function parameters are continuously updated, Adam helps balance stability and convergence speed without extensive hyperparameter tuning. Its empirical success across a wide range of tasks makes it a practical and effective choice for optimizing deep neural networks in reinforcement learning.

### 3.6 Hyperparameters

Our hyperparameters are batch_size (batch size), $K$ (number of minibatches to process), $T$ (number of steps in trajectory), step_max (number of iterations), hidden_size (hidden layer size) and $lr$ learning rate. We set our final hyperparameters based on Yu et. Al's paper [2], computational considerations and hyperparameter tuning. In particular, we experimented with various values for batch_size and $K$. For larger batch_size, the agents fail to learn (see "cumulative_reward_large_batch.png" in our codebase, which uses batch_size$=10$ and $K=5$). Additionally, we chose to train our agents for 100 iterations, which allowed the agents to learn while simultaneously mitigating computational cost. See Table 1 for our final hyperparameter values.

| **hyperparameters** | **values** |
|---|---|
| batch_size | 5 |
| $K$ | 10 |
| $T$ | 500 |
| step_max | 100 |
| hidden_size | 1024 |
| $lr$ | 0.001 |

Table 1: Tuned hyperparameters

### 3.7 Complete Algorithm

Our complete algorithm is based off of Yu et al's paper on Multi-Agent Proximal Policy Optimization.[2]Some adaptations include:

1. We expanded our action space to incorporate a hierarchical structure, as noted in the section on Actor-Critic Framework for Multi-Agent Setting
2. We simplified loss values to exclude entropy terms. Entropy in the loss focuses on encouraging action diversity. While helpful, we omitted this entropy term due to (1) greater simplicity (2) based on the distribution of actions in our evaluation run, we had a good amount of diversity.

```
MAPPO Algorithm

Initialize optimizers for actor and critic policies and set
    initial policy parameters

For each training iteration:
    For each batch:
        For each step in the environment:
            For each agent:
                - For every type of action sample actions based on
                    the actor RNN and update hidden states
                - Obtain value base don the critic RNN and update
                  hidden states
                - Update trajectory with states, observations,
                  actions, and hidden states
        Process the collected trajectories:
            - Compute advantage function
            - Compute returns (discounted rewards-to-go)


    For each minibatch in all trajectories collected:
        For each datachunk in minibatch:
            Update RNN hidden states from first hidden state in
                data chunk and propagate

    Update actor policies:
        - Compute policy loss
        - Perform backpropagation and update actor parameters
           using the optimizer

    Update critic policies:
        - Compute value loss
        - Perform backpropagation and update critic parameters
           using the optimizer

Run eval with trained RNNs.
```

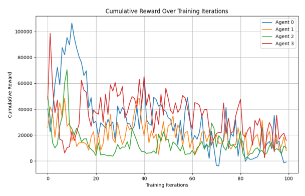
### 3.8 Code

We use online training and generate our data synthetically. At the beginning of the game, settlers and warriors for each agent are spawned randomly on the map. At each update step, we generate data by sampling trajectories with our current actor RNN. The code can be accessed at https://github.com/Ege-Cakar/COMPSCI184-Final-Project

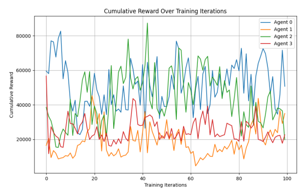
## 4 Results

### 4.1 Cumulative Reward over Training Iterations

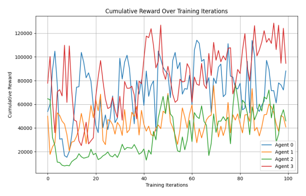
We train our four agents over 3 different environmental impact values ($\gamma$), where a higher environmental impact puts more punishes environmentally unfriendly actions more. Below is the cumulative reward by agent for $\gamma = 10^{-3}$, $10^{-4}$ and $10^{-5}$. See the "outputs" folder in the codebase for larger plots.



(a) $\gamma = 10^{-3}$



(b) $\gamma = 10^{-4}$

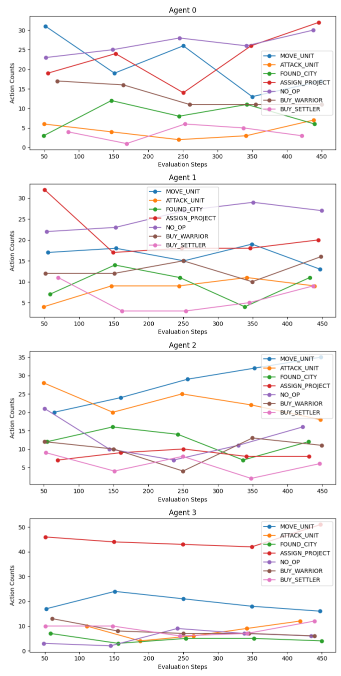


(c) $\gamma = 10^{-5}$

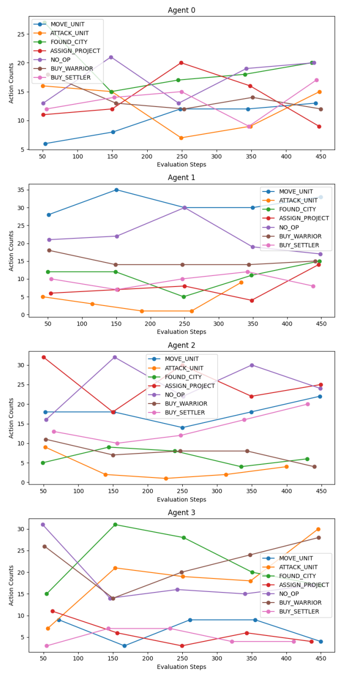
Figure 5: Cumulative Rewards by Environmental Impact.

### 4.2 Diversity of Actions

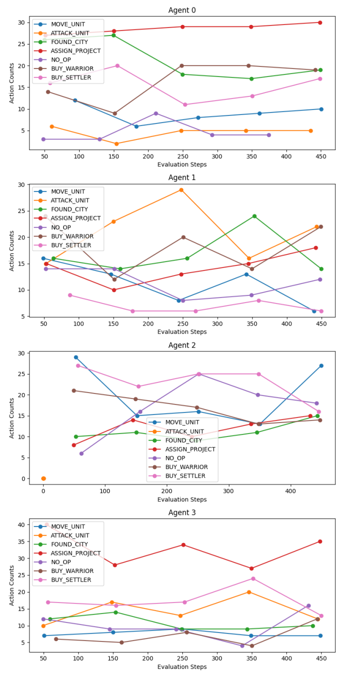
Our agents perform a diverse range of actions throughout the game for all three environmental impact values. The reason why we're still seeing so many NO-OP actions is because we are lacking in training iterations and the state space is still not explored properly. There are **no cases** where taking the action NO-OP is beneficial – even just moving a warrior to a square already explored is better, since it avoids the NO-OP penalty. We can see that for some agents, the frequency of the NO-OP action drops to zero in later iterations, suggesting that proper training is occurring. See the "outputs" folder in the codebase for larger plots.



(a) $\gamma = 10^{-3}$



(b) $\gamma = 10^{-4}$



(c) $\gamma = 10^{-5}$

Figure 6: Actions by Environmental Impact.

## 5 Discussion

The Effects of the Environmental Discount factor are very apparent in the plots. First of all, we see a general decrease in the rewards earned by agents, which makes sense. Looking first at $\gamma = 10^{-5}$, we see that Agent 3 is in the lead, with Agent 0 coming in slightly behind. Thus, it is appropriate to regard these as the best strategies for this gamma value – and for these, we see that both agents prioritize assigning projects. From the fact that Agent 3 founds less cities but assigns even more projects, we see that when $\gamma$ is this small, the benefits of finishing destructive projects put Agent 3 in the lead.

For $\gamma = 10^{-4}$, we see that Agent 0 has a clear lead. And it is here that we see the limits of the limited time and compute we had available to us – the fact that the best Agent performs this many NO-OP actions means that we have not been able to converge to the best policy yet, when environmental impact as its greatest. We see that Agent 2, who focuses on Assign Project and NO-OP, is doing semi good, but we cannot necessarily say for sure.

For $\gamma = 10^{-3}$, we see that the rewards of agents are so close to each other it's not reasonable to make a convergence argument yet. However, we see stark differences in policies learned, and all leading to very similar rewards, meaning that the agents are struggling to find effective methods of growing. This might be because we are at the point that our agents need to learn and explore much longer, or just that $\gamma$ is too large. Collecting a dataset with an exploratory policy $\pi_e$, then training online might be able to help solve this problem.

## 6 Conclusion

In conclusion, we see that if the agents don't need to care for the environment, they're able to converge to a relatively simple solution very fast, whereas when needing to care for the environment, they struggle to converge to a solution in the same amount of time, and the environmental punishment might be too strong.

## References

[1] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. 31:2980–2991, 2018.

[2] Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of ppo in cooperative, multi-agent games. *arXiv.org*, Nov 2022. URL `https://arxiv.org/abs/2103.01955`.