PPO Agent for Dicewars

Eleni Koumparidou
Natalia Anastasia Kousta
Machine Learning

April 2025

1 Description

This project presents the implementation of an AI agent for the game Dicewars using Reinforcement Learning. The report aims to analyze the chosen RL method, describe the implementation of the agent, assess its performance, and propose potential improvements.

2 Proximal Policy Optimization (PPO)

The Reinforcement Learning algorithm that we used to implement our player was a policy gradient method, the **Proximal Policy Optimization (PPO)**. PPO improves the learning progress during training, providing more stable policy upgrades while preventing any performance collapse.

2.1 Learning process

The learning process of the PPO agent is based on two neural networks:

- **Policy Network**: It is used to improve the player's policy by providing a probability distribution over possible actions given a state.
- **Value Network**: Estimates the expected cumulative future reward from a given state (Value function). This helps the agent to assess how good or bad an action was.

2.2 Learning Optimization

The agent optimizes the policy using a clipped objective function for small updates and the prediction of future rewards using the Mean Squared Loss (MSE). All the formulas used for the optimization of the networks are provided bellow:

- $$L_{actor}(\theta) = \hat{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{1}$$

- $$\mathcal{L}_{critic} = \frac{1}{N} \sum_{i=1}^{N} \left(V(s_i) - \hat{R}_i \right)^2 \tag{2}$$

- $$A_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \tag{3}$$

- $$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{4}$$

Where:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the probability ratio between the new and old policy.
- $\hat{A}_t$ is the estimated **advantage function**, measuring how much better an action is performed compared to the average.
- ϵ is a hyperparameter that controls the allowed deviation from the old policy.
- γ: discount factor
- δ_t: Temporal Difference (TD) error at current state t
- λ: GAE parameter for bias-variance tradeoff
- r_t: reward at current state t
- $V(s_t)$: Value estimation at current state t
- $V(s_{t+1})$: Value estimation at next state t+1

2.3 State Representation for Learning

We carefully selected the following state information to provide the agent with meaningful inputs that support effective training, while avoiding unnecessary or redundant data that could mislead learning.

- **Number of dice per neighboring area:** The number of dice that every neighbouring area to our player has. We decided to focus only on the neighbours so that the player can decide on a more beneficial action.
- **Number of dice per player:** The total number of dice each player has at the current state.
- **Number of areas per player:** The number of areas each player has at the current state.

- **Maximum cluster per player:** The size of the largest cluster each player has.
- **Number of stocks per player:** The number of stocks each player has.

2.4 Reward function

Another crucial feature for our agent's efficient learning is the reward function. In order to motivate our agent's learning progress we implemented a reward function that provides positive or negative rewards based on the result that each action has on the parameters used for the state input. So, different small positive rewards are given when the player increases the number of areas, dice or stocks and a large reward in case of winning.

3 Implementation Overview

3.1 Overview of Core Functions

A brief description of the functions used in our agent and player is provided in Table 1 and Table 2.

Function	**Description**
`__init__`	Initializes the PPO agent by setting the necessary parameters for training.
`build_actor`	Builds the policy (actor) neural network.
`build_critic`	Builds the value estimation (critic) neural network.
`compute_advantage`	Computes the advantage function used for policy updates.
`train`	Trains the agent using collected experiences from the games.
`run`	Runs the full training process by initializing and executing multiple games.
`state_descriptor`	Converts the current environment state into the training format.
`get_attack_areas`	Chooses a valid attack by calling `select_action`.
`select_action`	Selects an action using the policy network and epsilon-greedy strategy.
`get_reward`	Calculates the reward based on the current state.
`save_model_weights`	Saves the trained weights of the model for future use.
`training_plots`	Generates and saves plots to help track training performance.

Table 1: Descriptions of main functions in the PPO agent class

Function	Description
`__init__`	Initializes the Player with the PPO agent, either by loading pre-trained weights or by training from scratch.
`load_model_weights`	Loads the pre-trained weights for the PPO agent.
`get_attack_areas`	Selects a valid attack based on the trained Policy network.

Table 2: Descriptions of main functions in the `Player` class

3.2 Policy and Value Networks

We implemented the Policy and Value networks (Actor and Critic) using the TensorFlow library, as seen in Listing 1. Each neural network consists of four layers, including two hidden layers with 256 nodes each and ReLU activation functions. The output layer of the Policy network uses a softmax activation function to produce a probability distribution over actions. To prevent overfitting, we applied L2 regularization and we used the Adam optimizer for updating the network weights during training.

```python
def build_actor(self):
#...
    inputs = Input(shape=(self.state_dim,))
    x = Dense(256, activation='relu', kernel_regularizer=l2
        (1e-4))(inputs)  # for overfitting
    x = Dense(256, activation='relu')(x)
    outputs = Dense(self.num_possible_grid_actions,
        activation='softmax')(x) # Probabilities distribution

    return Model(inputs, outputs)

def build_critic(self):
    #...
    inputs = Input(shape=(self.state_dim,))
    x = Dense(256, activation='relu', kernel_regularizer=l2
        (1e-4))(inputs)
    x = Dense(256, activation='relu')(x)
    outputs = Dense(1, activation=None)(x)  # Value
        estimation

    return Model(inputs, outputs)
```

Listing 1: Actor and Critic Networks

3.3 Reward Function

We implemented the `get_reward` method to give rewards to the agent based on the game state following each action, as described before. After testing the effectiveness of different reward combinations, we decided to provide small positive rewards for beneficial outcomes, while emphasizing on territorial expansion. To encourage this behavior, area conquest is rewarded with a larger scaled value (multiplied by 0.1). Additionally, a significantly larger reward (+10) is given when the player wins the game to motivate strategies that lead to victory.

```python
def get_reward(self, state, next_state):
    """
    Gives a reward based on the current and the next state
    """
    reward = 0.0

    # Rewards criteria
    area_diff = next_state.player_num_areas[0] - state.
        player_num_areas[0]  # number of areas
    dice_diff = next_state.player_num_dice[0] - state.
        player_num_dice[0]    # number of dice
    stock_diff = next_state.player_num_stock[0] - state.
        player_num_stock[0] # number of stock dice

    reward += 1.0 * area_diff
    reward += 0.05 * dice_diff
    reward += 0.05 * stock_diff

    if state.winner == 0:   # Big reward when agent wins
        reward += 10.0

    return reward
```

Listing 2: Reward function

3.4 State Representation

To adapt the current game state for learning purposes, we created a descriptor called `state_descriptor`. The selected state features are those outlined in the ”Input of Learning” chapter.

First, we identify the neighboring areas of the player. Then, using a helper mask array, we construct an array that contains the number of dice for each neighboring area, while assigning zeros to all non-neighboring areas, so that the agent can focus on the information that are important for the next move. Then we add all the other state features to the state array.

```python
def state_descriptor(self, grid, state):
```

```
#...

# DICES OF NEIGHBOURS
neighbours_states = []
neighbours_areas = []

# Find the areas that are neighbours of the player's
    areas
for area in state.player_areas[0]:
    neighbours_areas.extend(grid.areas[area].neighbors)

# List of neighbouring areas
neighbours_areas = list(set(neighbours_areas))  # Keep
    unique indices

# Mask for neighbours
mask = np.zeros_like(state.area_num_dice, dtype=np.
    float32)
mask[neighbours_areas] = 1

area_num_dice = np.array(state.area_num_dice, dtype=np.
    float32)
num_dice_neighbours = area_num_dice * mask  # Get
    neighbour's dices


#...
# Create the state list
neighbours_states.append(num_dice_neighbours)   # Number
     of neighbour's dices ---> Length = 30
neighbours_states.append(player_dices)          # Number
     of dices per player ---> Length = 4
neighbours_states.append(player_areas_flat)     # Number
     of areas per player ---> Length = 4
neighbours_states.append(player_max_size)       # Size
    of largest cluster per player ---> Length = 4
neighbours_states.append(player_num_stock)      # Number
     of stock dices per player ---> Length = 4

# Concatenate all features into a single array
state_features = np.concatenate([np.atleast_1d(feature)
    for feature in neighbours_states]).astype(np.float32)

return state_features
```
Listing 3: State Descriptor

3.5 Action Selection

Our agent attacks with `get_attack_areas` and selects an action with `select_action`. An action is selected based on the **Greedy-Epsilon** strategy. This means that the agent chooses a random action with probability ϵ to encourage exploration of less frequently chosen strategies. This helps discover potentially valuable actions that may not be currently favored by the policy. Otherwise, the agent selects the action with the highest probability, as predicted by the actor network, to exploit known optimal behavior.

3.6 Training details

We train the agent using two main functions, `run` and `train`. The first one executes the training by playing a pre-defined number of games (`episodes`) and the second one trains the agent after a pre-defined number of finished games (`train_freq`). While playing the game the agent collects experience (states, actions, rewards, probabilities) and when the training comes, a random batch of the training experience is selected to train the agent.

4 Training Evaluation and Performance

The training parameters are available in Table 3.

Hyperparameter	**Value**
Actor learning rate	`lr_actor = 0.0003`
Critic learning rate	`lr_critic = 0.0001`
Discount factor	`gamma = 0.99`
Clipping parameter	`epsilon = 0.4`
Exploration parameter	`e_explore = 0.6`
Minimum exploration value	`e_min = 0.1`
Exploration decay rate	`e_decay = 0.99`
Number of games (episodes)	`episodes = 2000`
Training frequency	`train_freq = 20`

Table 3: Hyperparameters used for training the PPO agent

The player was trained against Random Players, achieving an average winning rate of approximately 0.30% during training. The winning rate of the trained model evaluated before the tournament was 0.305%, while its performance during the tournament—against three Random Players over 1000 games—dropped to 0.18%. These results indicate that the overall performance of our agent remains relatively low.

4.1 Actor Loss Evaluation

In Figure 1, we observe that the actor loss remains relatively stable during the initial training sessions. However, as training progresses, the loss begins to fluctuate significantly, particularly toward the final episodes. Moreover, there is no evident downward trend throughout the training process.

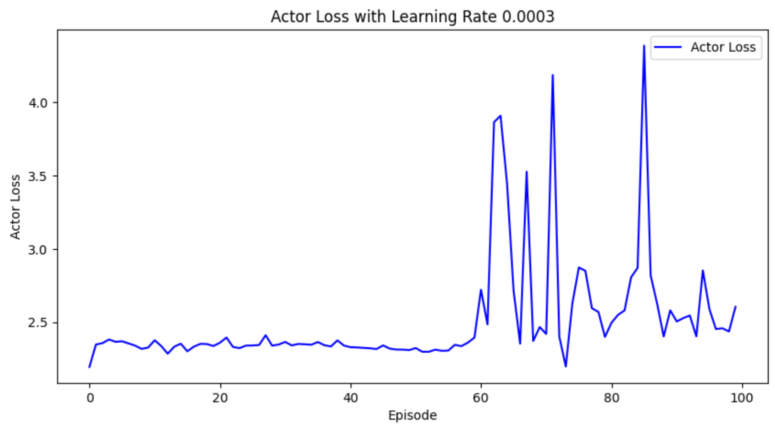

Figure 1: Actor loss function

4.2 Critic Loss Evaluation

In Figure 2, we observe that the critic loss reaches lower values and demonstrates greater stability compared to the actor loss. While occasional drops in the critic loss are noticeable, there is no consistent decreasing trend over time. This could indicate that the value estimates are not progressively improving.

4.3 Rewards Evaluation

Figure 3 shows the total rewards obtained in each episode during training. Although the agent manages to achieve high rewards at certain points, the overall trend does not show consistent improvement over time.

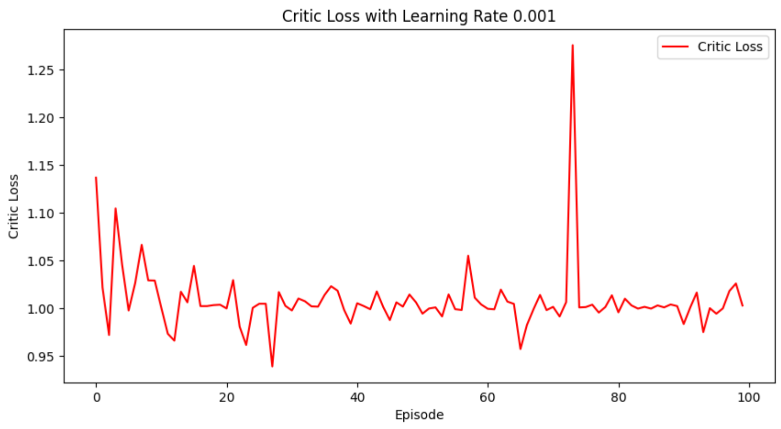

Figure 2: Critic loss function

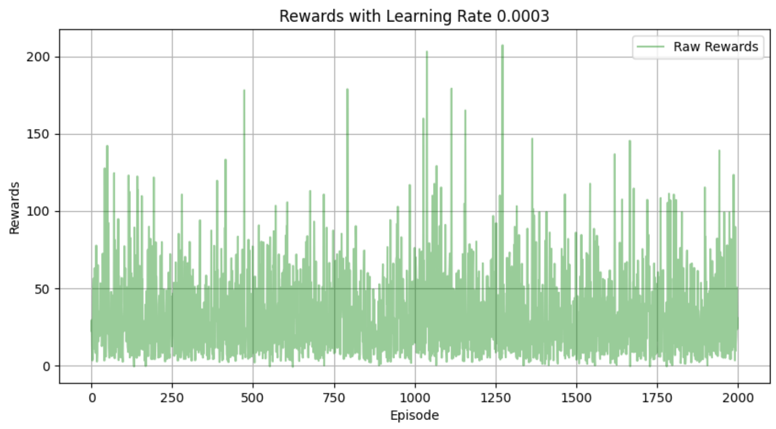

Figure 3: Rewards during training

5 Improvements

The evaluation of the training and the performance of our player highlight opportunities for further optimization and improvement. We provide some suggestions that may enhance the learning procedure :

- **Reward Function Adjustment**:

  It is possible that the current reward function does not provide sufficient or consistent feedback to effectively guide the agent during training. One potential improvement is to adjust the scaling parameters of the reward function. Increasing the reward magnitude could make the objective more explicit. Alternatively, reducing the reward scale may provide more stable learning.

- **Training Enhancement**:

  Another way to improve our agent's learning is by intensifying the training process. This can be achieved by increasing the number of training episodes, for example, extending it to 5.000, or by retraining the agent after the initial training phase to refine its previously learned weights. Additionally, in later training stages, the agent could be trained against more diverse opponents instead of only a Random Player. This would encourage the development of more powerful strategies.

- **Input State Adjustment**:

  We can also adjust the input state of the training, adding more features of the current state that may be helpful for a more meaningful learning procedure.

- **Hyperparameter Optimization**:

  While we conducted extensive tests to identify the optimal hyperparameters for the training procedure, it is possible that other combinations could further enhance the learning process of our agent.