GauchoMiner

UCSB CS 165A, Spring 2025, Machine Problem 2

Responsible TA: Xuan Luo
xuan_luo@ucsb.edu

Due: June 14, 2025, 11:59 PM

Notes

- You must implement reinforcement learning algorithms (e.g. Q-Learning, Policy Gradient) to solve this machine problem. We will use a code detector to verify your implementation.
- Please review the "Policy on Academic Integrity" in the course syllabus carefully.
- Make sure you have Python 3.11 installed along with all required libraries to run the code.
- Direct all questions about Machine Problem 2 to the Canvas MP2 Q&A forum.
- Plagiarism detection software will be used. You are expected to complete this project independently and not use code and checkpoints from others.
- If you find a bug or vulnerability in this machine problem, please report it to the responsible TA via email. Extra credit will be awarded based on the significance of the vulnerability.
- The assets are from Minecraft Education Edition and used in compliance with the EULA.

1 Background: Q-Learning with Linear Approximation

Q-Learning is a model-free reinforcement learning algorithm that enables an agent to learn an optimal action-selection policy by interacting with an environment. It aims to maximize the cumulative reward by estimating the expected future rewards for taking specific actions in given states, known as the Q-value. The algorithm updates Q-values using the Bellman equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

where s is the current state, a is the action, r is the reward, s' is the next state, α is the learning rate, and γ is the discount factor.

In real life, Q-Learning is widely applied in domains like robotics (e.g., autonomous navigation), game playing (e.g., Atari), and resource management (e.g., traffic light control). Its advantages include:

- **Model-Free**: No prior knowledge of the environment's dynamics is required.
- **Off-Policy**: It learns the optimal policy even when exploring suboptimal actions.
- **Simplicity**: The algorithm is straightforward to implement and adaptable to various problems.

Q-Learning with Linear Approximation is a variant of Q-Learning where the Q-value function is approximated using a linear combination of features. Instead of maintaining a Q-table for all state-action pairs, which can be infeasible for large or continuous state spaces, the Q-value is represented as:

$$Q(s,a;\theta) = \theta^T \phi(s,a)$$

Here, $\phi(s,a)$ is a feature vector representing the state-action pair, and θ is a weight vector learned during training. The weights are updated using gradient descent to minimize the difference between the predicted and target Q-values. Q-Learning with Linear Approximation is useful when dealing with large or continuous state spaces where traditional Q-tables become impractical due to memory and computational constraints.

2 Game Overview

In this assignment, you will create a program to control a miner in a 2D Minecraft-inspired mining game on an $X \times Y$ grid. The goal is to guide the miner through a partially visible map, digging and collecting gold and other rewards to maximize the score. The miner starts at a specific position with limited energy. Each action, such as moving or digging, consumes energy, with costs varying by block type (e.g., dirt, stone, or gold ore). The map is partially observable, with the miner seeing a 9x9 area around their position. Your program must find the optimal sequence of actions to collect the most rewards before the miner's energy runs out, ending the game.

The state representation, initial state, actions, and termination conditions match those in Machine Problem 1 (MP1), but the map elements are updated, and visibility is limited to the 9x9 local area. New dynamic elements—zombies, skeletons, and creepers—add complexity. Zombies move randomly but chase the miner when nearby. Skeletons wander and may shoot the miner within 4 blocks, costing 20 energy per shot. Creepers detect the miner from a distance and pursue them via the shortest path, with a chance to explode when within 4 blocks, causing 0–400 energy loss based on distance. Explosions will randomly destroy nearby ores, chests, barrels, and monsters, with the miner gaining rewards from destroyed ores or monsters, but chests and barrels yield no rewards. It simply means that you get the scores from the removed ores and monsters. If the miner and a monster occupy the same position, the miner automatically defeats the monster, costing 50 energy for a zombie and 1 energy for a skeleton or creeper. Defeating a zombie or skeleton grants 10 points, while defeating a creeper grants 20 points, though creepers typically explode first. To ensure the 9x9 local map remains consistent in size, even when the miner is near the grid's edges or corners, blocks outside the map boundaries are represented as "void." This maintains a uniform 9x9 view by filling any out-of-bounds areas with "void" blocks. The updated map elements are shown in the table below:

Table 1: Map State Blocks. The $[s, t]$ indicates the range of the random value.

Name	Icon	Energy	Reward
Empty		-1	0
Dirt		-2	0
Stone		-4	0
Deepslate		-10	0
Gold ore		-4	+5
Deepslate gold ore		-10	+5
Zombie		-50	+10
Skeleton		-1	+10
Creeper		-1	+20
Chest		-1	+[0, 20]
Barrel		+[20, 80]	0
Void	-	-1	0

3 Run the Game

To run the GauchoMiner game, you must install the required Python libraries and execute the provided script. The game features a graphical interface powered by Pygame for visualizing the miner's actions on the grid. Install the following Python libraries:

```
pip install pygame opensimplex pycryptodome
```

Ensure Python 3.11 is installed, Verify your Python version with:

```
python --version
```

After implementing your logic in `agent_logic.py` and training your Q-Learning with Linear Approximation model, launch the game by running the main script from the directory containing `new_game.py`:

```
python new_game.py
```

This command starts the game with default settings. You can customize the game environment using command-line arguments to adjust map generation, game mechanics, and display settings. Key arguments include:

- `--seed`: Sets the random seed for map generation (default: 42). Use the same seed to generate identical maps.
- `--width`, `--height`: Define the grid dimensions (default: 50x30).
- `--p_gold`, `--p_dgold`: Set the probability of gold in stone (default: 0.2) and deepslate (default: 0.4).
- `--zombies`, `--creepers`, `--skeletons`, `--chests`, `--barrels`: Specify the number of each entity (default: 20 zombies, 10 creepers, 10 skeletons, 15 chests, 15 barrels).
- `--energy`: Sets the miner's initial energy (default: 1000).
- `--training`: Enables (1) or disables (0) training mode (default: 1). We will only call the `update_q_leanring` functiin training mode. If you want to evaluate your implementation, please set it to 0.
- `--fps`: Controls game speed in frames per second (default: 5). Lower values (e.g., 1) slow the visualization for easier debugging.
- `--fog`: Enables (1) or disables (0) fog of war, limiting visibility to a 9x9 area (default: 1). Note: Disabling fog is for debugging only; the agent still cannot see the entire map.
- `--grid_size`: Sets the size of each grid cell in pixels (default: 24).
- `--render`: Enables (1) or disables (0) graphical rendering (default: 1). Disable rendering for faster training or testing.

4 Implementation Guidelines

Your task is to implement the `agent_logic.py` file, which defines the core decision-making logic for the GauchoMiner game. At each time step, the main game loop calls the `agent_logic` function to determine the miner's next action based on the current game state. In addition to `new_game.py`, which you execute to run the simulation, and `agent_logic.py`, where you implement your algorithm, the game includes `constants.py`, which contains all game constants. You may modify Q-learning related constants, such as learning rate, discount factor, or exploration rate, in `constants.py` to optimize the agent's performance.

4.1 Agent Logic Function Implementation

The `agent_logic` function receives a 9x9 `local_map` representing the visible portion of the game grid, the miner's `position` as global coordinates, the current `energy` and `score`, a `gold_count` dictionary, and a boolean `training` indicating whether training mode is active. In the `local_map`, the miner is positioned at the center. When the miner is near the grid's boundary, out-of-bounds blocks within the 9x9 `local_map` are filled with "void." The `local_map` contains integers from 0 to 11, each corresponding to a different block type, as defined in `constants.py`. The `position` provides the miner's global coordinates on the full grid, unlike the `local_map`'s relative view. The `gold_count` dictionary offers global information about the number of gold blocks remaining in each direction, defined as the count of gold blocks in half-planes relative to the miner's position. For a grid of size $X \times Y$, with the miner's position at (x, y), the `gold_count` is defined as follows:

- `gold_count['W']`: Number of gold blocks in $\{(x', y') \mid y' < y, 0 \le x' < X, 0 \le y' < Y\}$.
- `gold_count['A']`: Number of gold blocks in $\{(x', y') \mid y' > y, 0 \le x' < X, 0 \le y' < Y\}$.
- `gold_count['S']`: Number of gold blocks in $\{(x', y') \mid x' < x, 0 \le x' < X, 0 \le y' < Y\}$.
- `gold_count['D']`: Number of gold blocks in $\{(x', y') \mid x' > x, 0 \le x' < X, 0 \le y' < Y\}$.

These counts can guide your agent toward gold-rich areas and should be considered when designing features for Q-Learning with Linear Approximation. You have to firstly finish the agent logic, which is called every time step. Your implementation should follow these three steps:

1. **Feature Extraction**: Convert the input data into a feature vector suitable for linear Q-Learning. The `local_map` is a 9x9 grid, where each cell can be one of 12 possible block types (e.g., Void, Empty, Dirt, Stone, Deepslate, Stone Gold, Deepslate Gold, Chest, Barrel, Creeper, Zombie, Skeleton), as defined in `constants.py`. A baseline feature extraction implementation is provided, which achieves reasonable performance. It uses a one-hot encoding for the `local_map`, creating a vector of length $9 \times 9 \times 12 = 972$, where each position is 1 if the corresponding block type exists at a specific grid cell and 0 otherwise. Additionally, it allocates 16 bits for `energy` (representing values from 0 to $2^{16} - 1$), 16 bits for `score`, 32 bits for `gold_count` (8 bits per direction for the four directions), and 1 bit for the `training` flag, resulting in a total feature dimension of $972 + 16 + 16 + 32 + 1 = 1037$. While this baseline is effective, you can explore more efficient feature designs to improve performance, such as computing the distance to the nearest gold ore, counting the number of zombies or creepers in each direction, or removing less impactful features to simplify the learning process. A well-designed feature set that captures key game dynamics (e.g., proximity to rewards or threats) can significantly enhance the efficiency and effectiveness of Q-Learning. ning.

2. **Q-Value Calculation**: Compute the Q-values for each possible action (`W`, `A`, `S`, `D`, `I`) using the linear approximation $Q(s, a; \theta) = \theta^T \phi(s, a)$, where $\phi(s, a)$ is the feature vector for the state-action pair, and θ is the weight vector. Store the Q-values for all actions and identify the `max_q_value` (the highest Q-value among all actions), which will be used later for updating the weights during training.

3. **Epsilon-Greedy Action Selection**: Implement an epsilon-greedy strategy to select the next action, balancing exploration and exploitation. Use the `get_epsilon()` function to obtain a decaying epsilon value, which decreases as the number of steps increases. With probability ϵ, choose a random action to explore; otherwise, select the action with the highest Q-value. Notice that you may not use this strategy during testing. Finally, convert the selected action to its corresponding character using `ACTION_TO_CHAR` and return it.

4.2 Agent Weight Update

You will implement the `update_q_learning` function, which is called at each timestep during training to update the Q-learning weights based on environmental feedback. The function receives `delta_energy` and `delta_score`, which represent the changes in energy and score after executing the action produced by `agent_logic`. The `delta_energy` reflects energy consumed or gained, while `delta_score` is a non-negative value, indicating points earned from collecting gold, defeating zombies, or opening chests. These values are used to compute a reward and update the weights for Q Learning with Linear Approximation. Another parameter is `game_over`, which indicates if the game will over at this turn. The reward in the game over turn should ... Your implementation should follow these two steps:

1. **Update Weights**: Use the Temporal Difference (TD) error to update the Q-learning weights. The TD error is calculated as:

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, a)$$

where r is the `prev_reward`, γ is the `discount_factor`, $\max_{a'} Q(s', a')$ is the `max_q_value` from the current state, and $Q(s, a)$ is the Q-value for the previous state-action pair. The weight update rule is:

$$\Delta w = \alpha \cdot \delta \cdot \phi(s, a)$$

where α is the `learning_rate` and $\phi(s, a)$ is `prev_features`. To ensure training stability, the code includes gradient clipping, limiting Δw to the range $[-0.01, 0.01]$.

2. **Calculate the Reward**: Design a reward function that effectively guides the Q-learning process by incorporating `delta_energy` and `delta_score`. A simple starting point is the provided example: reward $= a *$ delta_score $+ b *$ delta_energy, which balances score gains with energy costs. However, you should refine this to improve exploration and learning efficiency. To prevent the agent from oscillating or revisiting recent positions, consider maintaining a memory (e.g., a short history of recent positions) and subtract a small penalty if the agent returns to a recently visited state. Additional features, such as the distance to the nearest gold ore or creeper (computed from `local_map`), can be included in the reward. For example, add a bonus for moving closer to gold or a penalty for approaching a creeper. Experiment with the relative weights of these terms to balance exploration and exploitation, ensuring the reward function drives the agent toward high-scoring, energy-efficient behavior.

5 Training

To train your Q-learning agent, run the command `python training.py` from the terminal. You can customize the training process by modifying `training.py` to suit your specific needs. The script accepts the following command-line arguments:

- `--episodes`: Sets the number of training episodes (default: 10000). Higher values enable the agent to learn from more diverse experiences.
- `--fps`: Controls the game speed in frames per second during training (default: 1000).
- `--save_interval`: Specifies the frequency, in episodes, for saving model checkpoints and printing training statistics (default: 100).

To improve your agent's robustness, modify the game environment in `training.py` to train under diverse conditions. The test set on Gradescope evaluates your implementation on maps with different distributions (e.g., varying numbers of zombies and creepers) compared to the default settings. Experiment with parameters to expose your agent to a range of scenarios, ensuring it generalizes well. You may also modify the training parameters like the learning rate within the `constant.py`. Hyperparameters like learning rate can greatly influence the performance.

After training, test your implementation with different settings by running:

```
python new_game.py --training 0 --seed xxx
```

Ensure disable training mode during evaluation, as this uses the learned weights without further updates. Test with various seeds and environment parameters to verify your agent's performance across diverse scenarios.

5.1 Submission Instructions

Submit your implementation to the Gradescope assignment named `MP2`. Include the following files:

- `agent_logic.py`: The Python script containing your RL implementation. **Note: Your program must not print any output and should only return a single character from** $\{'W', 'A', 'S', 'D', 'I'\}$.
- `ckpt.npz`: The weights of the trained q learning algorithm.
- Any additional files required for execution.

5.2 Evaluation

The autograder will automatically execute and evaluate your code on a test set comprising diverse map configurations. The final score of your submission will be determined based on its performance relative to our baseline score, according to the following criteria:

- $0\% \leq$ score $< 100\%$ of the baseline: Scores will be linearly interpolated between 0% and 100% of the base grade.
- $100\% \leq$ score $< 150\%$ of the baseline: Scores will be linearly interpolated between 100% and 120% of the base grade, with a maximum of 20% extra credit.
- Out of time: If your implementation exceeds the Gradescope time limit of 10 minutes, you will receive a score of 0.
- Due: June 14, 2025, 11:59 PM.

The submission will be named "MP2". Every student must submit their implementation before the deadline.

5.3 Leaderboard

A leaderboard will be enabled for this assignment to encourage competition and reward top performers. A separate submission portal named "MP2-Competition" will open on June 14, 2025, and remain available until 11:59 PM that day. Participation in the competition is optional and intended for extra credit only. The portal uses maps with different seeds for competition purposes, and you can submit the same files used for the "MP2" submission. The rewards are as follows:

- 1st place: Receives 50% extra credit, cumulative with the score-based extra credit.
- 2nd and 3rd place: Each receives 25% extra credit, cumulative with the score-based extra credit.
- Top 10: Each receives 10% extra credit, cumulative with the score-based extra credit.

To maximize your score, ensure your implementation is efficient, generalizes well across diverse environments, and is thoroughly tested with various settings (e.g., different seeds, map sizes, and entity counts) using `new_game.py` in evaluation mode (`--training 0`). Optimize your feature extraction, reward function, and Q-Learning parameters, such as learning rate, discount factor, and exploration rate, to achieve high scores while staying within the 10-minute time limit. Test your agent extensively in different scenarios to ensure robustness and avoid overfitting to specific map configurations.