Capture The Flag (CTF) Pacman with Reinforcement Learning

Instructed by Prof. Demetri Terzopoulos

Kevin Lee[1], Yu-Hsin Weng[2], Varun Kumar[3], and Siddarth Chalasani[4]

[1]Mechanical and Aerospace Engineering, UCLA , kevinlee69720@g.ucla.edu
[2]Computer Science, UCLA , yuhsin1614@ucla.edu
[3]Computer Science, UCLA , vvkumar1@g.ucla.edu
[4]Computer Science, UCLA , darthsid2000@g.ucla.edu

June 21, 2024

Abstract

This report presents the development and implementation of a Capture The Flag (CTF) variation of Pacman using both non-ML and ML approaches for agent behavior. The project leverages reinforcement learning (RL) algorithms such as Q-Learning and Proximal Policy Optimization (PPO) to enable adaptive and intelligent agent strategies. Our interdisciplinary approach integrates machine learning, game development, and artificial life concepts, showcasing the potential of RL in creating life-like behaviors in virtual environments. The integration of ML models within Unity provides a robust platform for visualizing and analyzing agent behaviors, making this project highly relevant to advancements in computer graphics and vision.

1 Introduction

The main motivation for this project is to simulate real-world cooperation through a CTF variation of Pacman. This simulation provides insights into multi-agent systems exhibiting life-like behaviors. By employing Reinforcement Learning (RL) algorithms, agents learn and adapt their strategies over time, allowing us to study complex system dynamics in a controlled environment. This work has practical applications in robotics, autonomous systems, and game development, leveraging Unity for visualization and integrating concepts from machine learning and artificial life [1].

Reinforcement Learning (RL) involves training an agent to make sequences of decisions by rewarding it for desirable actions and punishing it for undesirable ones. [2]. Artificial life, a field closely related to RL, focuses on understanding life through the synthesis of life-like behaviors in computational systems. This includes studying how agents can adapt, evolve, and exhibit complex behaviors in virtual environments. By simulating life-like behaviors, researchers can gain insights into biological processes and develop advanced AI systems that can adapt to real-world.

Several studies have demonstrated the effectiveness of RL in training agents for complex tasks. For example, Mnih et al. (2015) introduced Deep Q-Networks (DQNs), which combined Q-Learning with deep neural networks to achieve human-level performance in Atari games [3]. Similarly, Schulman et al. (2017) developed PPO, an algorithm that balances exploration and exploitation more effectively, leading to significant improvements in policy optimization [4].

Our project leverages these advancements to create a dynamic and interactive CTF game. By then integrating these RL models within Unity, we can visualize and analyze the behaviors of our agents in a rich, interactive environment. This interdisciplinary approach, combining computer graphics, vision, and machine learning, showcases the potential of artificial life simulations in advancing AI research.

2 Rules

Each team (Red Team and Blue Team) starts with two agents in the state of ghosts. When an agent crosses over to the other side, it transforms into a Pacman. The primary objective of the game is for Pacman to eat pellets on the enemy's side and deliver them back to their own side. The team that captures the most pellets wins.

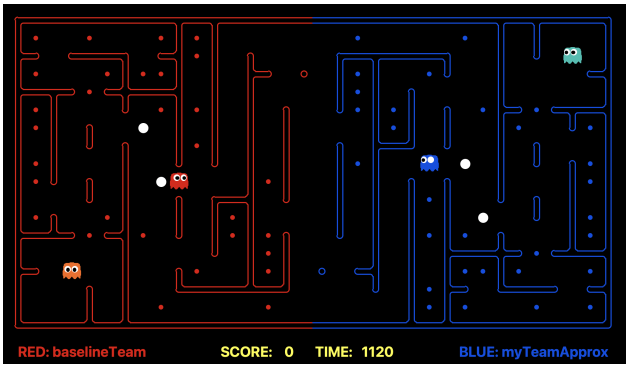

Figure 1: Game layout illustrating the initial setup with two agents per team. Each team must strategically manage their agents' roles between Pacman and ghosts to optimize pellet collection and defense.

Ghosts can kill opposing Pacman agents, adding a layer of strategic depth to the gameplay. This necessitates a balance between offensive and defensive strategies. Teams must carefully decide when to switch their agents from ghosts to Pacman and vice versa, optimizing their strategy to capture more pellets while defending against the opposing team's Pacman agents.

This dynamic interaction between agents, involving state transitions and strategic decision-making, is central to our research. It highlights the importance of developing intelligent agents capable of adapting their strategies in real-time. By employing reinforcement learning algorithms such as Q-Learning and Proximal Policy Optimization (PPO), we aim to simulate and analyze these complex multi-agent systems, providing valuable insights into cooperative and competitive behaviors in artificial life scenarios.

3 Overall System Design

The system architecture of our project comprises several key components, designed to ensure efficient and effective gameplay while leveraging the strengths of both non-ML and ML approaches.

Game Engine: The game engine is responsible for managing the overall game state, including the positions and states of the agents. It handles the interactions between agents, ensuring the game rules are enforced and the game logic is executed correctly. The game engine also updates the game environment based on the actions taken by the agents, maintaining a seamless flow of the game.

Action Models: The action models in our project can be either non-ML (greedy algorithms) or ML-based (reinforcement learning). For non-ML models, we implement basic heuristic strategies that guide the agents' actions based on predefined rules. For ML-based models, we use reinforcement learning algorithms such as Q-Learning and Proximal Policy Optimization (PPO) to enable agents to make informed decisions based on the current game state. These models are trained to optimize their strategies through trial and error, receiving rewards for successful actions and penalties for failures [2].

Integration into Unity: To bridge the gap between training and deployment, we use ONNX (Open Neural Network Exchange) models to transport our models from Python to Unity. ONNX is a standardized format that easily allows us to utilize pre-trained ML models in Unity. This integration facilitates real-time decision-making

and enhances the interactive experience of the game. For the approximate Q-learning models, we trained them in Python to obtain the feature weights. However, integrating these weights and features into Unity posed significant challenges due to the slight difference in game state. As a result, our most effective models were deployed in the Python simulation rather than Unity.

Visualization and Simulation: Unity is used to visualize agent behaviors and the game environment, providing a dynamic and engaging platform for both development and demonstration. The use of Unity allows us to create detailed simulations of the game, where we can observe the interactions between agents and analyze their decision-making processes in real time. However, due to challenges in integrating approximate Q-learning models with Unity, our final product was implemented and tested within the Python PyGame simulation environment instead.

4 Strategies

Q-Learning Algorithm

Q-Learning is a model-free reinforcement learning algorithm designed to help an agent learn the quality of actions, essentially guiding the agent on which action to take under specific circumstances. The core idea behind Q-Learning is to learn a Q-function, $Q(s, a)$, which estimates the expected utility or total reward of taking action a in state s and subsequently following the optimal policy. This approach allows the agent to learn from the environment through trial and error, iteratively improving its policy.

The algorithm begins by initializing the Q-values arbitrarily for all possible state-action pairs. It then observes the initial state and enters the learning loop. During each episode, for each step, the agent selects an action a based on the current state s using a policy derived from the Q-values. A common choice is the ϵ-greedy policy, which balances exploration and exploitation.

Algorithm 1 Q-Learning Algorithm

1: Initialize $Q(s, a)$ arbitrarily for all state-action pairs
2: Observe the initial state s
3: **for** each episode **do**
4: **for** each step in the episode **do**
5: Choose action a from state s using policy derived from Q (e.g., ϵ-greedy)
6: Take action a, observe reward r and next state s'
7: Update $Q(s, a)$ using the update rule:

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

8: $s \leftarrow s'$
9: **end for**
10: **end for**
11: **Output:** The learned Q-values $Q(s, a)$

After choosing an action, the agent executes it, receives a reward r, and observes the next state s'. The Q-value for the state-action pair (s, a) is then updated 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 α is the learning rate, and γ is the discount factor. The term $r + \gamma \max_{a'} Q(s', a')$ represents the estimated optimal future value, and $Q(s, a)$ is adjusted to reduce the discrepancy between the current estimate and this optimal value.

This process continues until the Q-values converge, meaning that the agent has learned the optimal policy for maximizing its expected reward in the given environment. The final output of the algorithm is the set of learned Q-values, $Q(s, a)$, which can be used to derive the optimal policy for the agent.

4.0.1 State Representation

We defined the state of the game as a combination of the agent's position, the positions of the pellets, and the po-

sitions of the ghosts. This comprehensive state representation allows the Q-Learning algorithm to make informed decisions based on the entire game context.

4.0.2 Action Selection

In each state, the agent can choose from a set of actions: move up, down, left, right, or stay idle. We used an ϵ-greedy policy to select actions, where the agent chooses a random action with probability ϵ and the best-known action otherwise. This approach balances exploration and exploitation, allowing the agent to discover optimal strategies while still leveraging known high-reward actions.

4.0.3 Pros and Cons

Q-Learning offers several advantages, primarily its simplicity and ease of implementation. At its core, Q-Learning is essentially a key/value mapping that updates Q-values iteratively based on the current state-action pair and the received reward. This simplicity makes it straightforward to understand and implement. Additionally, Q-Learning efficiently improves the policy by learning from the Q-values, enabling the agent to make informed decisions that maximize the expected reward [5].

However, Q-Learning also has significant drawbacks. One of the main challenges is the need for careful engineering of the reward function, which can be complex and time-consuming. Moreover, Q-Learning struggles with large state spaces due to its reliance on storing and updating Q-values for every possible state-action pair. This limitation makes it impractical for environments with vast or continuous state spaces, where it becomes almost impossible to learn every state representation. Consequently, if an opposing agent makes an unfamiliar move, the model may resort to choosing a random, uninformed action, leading to suboptimal performance.

4.0.4 Suitability for the Project

Due to the state representation issue, basic Q-Learning was not suitable for our project. The large state space and the need for precise handling of dynamic and complex environments made it impractical to use Q-Learning for our Pacman agents.

4.1 Proximal Policy Optimization (PPO) Algorithm

Proximal Policy Optimization (PPO) is an advanced reinforcement learning algorithm that optimizes the policy network (neural network) by improving the likelihood ratio of new and old policies within a clipped range. PPO was introduced to address the instability and inefficiency of traditional policy gradient methods [4]. The key innovation of PPO is the use of a clipped objective function, which ensures that the policy updates are not too large, thereby maintaining stability during training. Additionally, its advantage estimate determines how good a chosen action was compared to the average action taken in that state.

Algorithm 2 Proximal Policy Optimization (PPO) Algorithm

1: Initialize policy parameters θ and value function parameters ϕ
2: **for** each iteration **do**
3: Collect set of trajectories $\{(s_t, a_t, r_t, s_{t+1})\}$ by running policy π_θ in the environment
4: Compute advantage estimates $\hat{A}_t$
5: Update the policy by maximizing the PPO-Clip objective:

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]$$

6: Update the value function by minimizing the value function loss:

$$L^{\text{VF}}(\phi) = \hat{\mathbb{E}}_t \left[(V_\phi(s_t) - \hat{R}_t)^2 \right]$$

7: **end for**

PPO begins by initializing the policy parameters θ and the value function parameters ϕ. In each iteration, a set of trajectories is collected by running the current policy π_θ in the environment. These trajectories include sequences

of states, actions, rewards, and subsequent states.

The algorithm then computes advantage estimates $\hat{A}_t$, which measure how good the chosen actions were compared to the average action taken in those states. The policy is updated by maximizing the PPO-Clip objective:

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]$$

where $r_t(\theta)$ is the probability ratio of the new and old policies, and ϵ is a hyperparameter that controls the clipping range. The clipping mechanism prevents the policy updates from being too large, ensuring stability.

The value function is updated by minimizing the value function loss:

$$L^{\text{VF}}(\phi) = \hat{\mathbb{E}}_t \left[(V_\phi(s_t) - \hat{R}_t)^2 \right]$$

where $V_\phi(s_t)$ is the estimated value of state s_t according to the value function, and $\hat{R}_t$ is the estimated return.

By iteratively updating the policy and value function using these objectives, PPO achieves stable and efficient learning, allowing the agent to improve its performance in complex environments.

4.1.1 Pros and Cons

Proximal Policy Optimization (PPO) has several notable advantages. One of the key benefits is its stability, achieved through clipping, which prevents large updates to the policy and ensures more stable learning. Additionally, PPO is highly efficient in terms of data usage, as it performs multiple updates on the same batch of data. This efficiency allows the algorithm to make the most out of the collected experiences, leading to faster and more reliable convergence [6].

Despite these advantages, PPO also has some drawbacks. The algorithm requires significant computational power and memory, especially when applied to large and complex environments. This requirement can make it challenging to implement PPO in resource-constrained settings. Moreover, PPO is sensitive to hyperparameters, necessitating very careful fine-tuning to achieve optimal performance. This sensitivity can make the training process more time-consuming and complex, as finding the right set of hyperparameters often involves extensive experimentation and validation.

4.1.2 Suitability for the Project

PPO's ability to handle complex environments makes it ideal for dynamic and unpredictable settings like Pac-Man. However, the sensitivity to hyperparameters and the computational resources required made it challenging to implement effectively in our project. Although PPO offers improved stability and performance over simpler algorithms like Q-Learning, the practical constraints led us to explore alternative methods.

5 Approximate Q-Learning Algorithm

5.1 Approximate Q-Learning Algorithm

Approximate Q-Learning is a variant of Q-Learning that uses function approximation to generalize Q-values across similar states. This method is particularly useful when dealing with large state spaces where storing a Q-value for each state-action pair is impractical. Initial tests showed promising results, and after further tuning, its effectiveness was obvious [1].

In Approximate Q-Learning, instead of maintaining a table of Q-values for each state-action pair, we use a parameterized function, typically a linear function or a neural network, to approximate the Q-values. The parameters of this function are referred to as weights w.

The algorithm begins by initializing the weights arbitrarily. During each episode, the agent interacts with the environment, choosing actions according to a policy derived from the current Q-function, which is often an ϵ-greedy policy to balance exploration and exploitation.

Algorithm 3 Approximate Q-Learning Algorithm

Initialize weights w arbitrarily
for each episode **do**
  for each step in the episode **do**
    Choose action a from state s using policy derived from Q (e.g., ϵ-greedy)
    Take action a, observe reward r and next state s'
    Compute the temporal difference error:

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

    Update weights w using the update rule:

$$w \leftarrow w + \alpha \delta \phi(s, a)$$

  end for
end for
Output: The learned weights w

After taking an action a in state s and observing the reward r and the next state s', the agent computes the temporal difference error δ:

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

where γ is the discount factor.

The weights w are then updated using the computed error δ and the feature vector $\phi(s, a)$ associated with the state-action pair:

$$w \leftarrow w + \alpha \delta \phi(s, a)$$

where α is the learning rate.

By iteratively updating the weights, the algorithm learns an approximation of the Q-function that can generalize across similar states, making it suitable for environments with large or continuous state spaces.

5.2 Implementation in Pac-Man

In the Pac-Man project, Approximate Q-Learning was employed to manage the large state space more efficiently. Instead of using a table to store Q-values for each state-action pair, we used a linear function approximator parameterized by weights. The features included but were not limited to: distances to pellets, ghosts, and walls.
The implementation involved several key steps:

- **Feature Extraction**: We defined features such as the distance to the nearest pellet, the distance to the nearest ghost, and whether Pac-Man was in a power pellet state.
- **Weight Initialization**: We initialized the weights arbitrarily and updated them based on the temporal difference error during training.
- **Action Selection**: Pac-Man selected actions using an ϵ-greedy policy, which balances exploration and exploitation.
- **Updating Weights**: After each action, the weights were updated using the gradient of the Q-value function with respect to the weights.

By using Approximate Q-Learning, Pac-Man learned to generalize from past experiences to new situations, allowing for efficient learning even with a large state space. This approach improved the agent's ability to make decisions in a dynamic and complex environment.

5.2.1 Pros and Cons

Approximate Q-Learning offers several advantages. One significant benefit is its ability to handle large state and action spaces by approximating many different states to the same Q-value. This capability allows the algorithm to generalize across similar states, improving learning efficiency. By using a linear combination of features and weights, Approximate Q-Learning can effectively manage complex environments where storing Q-values for every state-action pair would be impractical.

However, Approximate Q-Learning also has some drawbacks. It is more complex to implement than basic Q-Learning and requires careful tuning of the function approximator. This added complexity can make the development process more challenging, but the trade-off is worth it simply because of its great performance. Thus,

it was suitable for our project. However, due to challenges in integrating approximate Q-learning models with Unity, due to the game-state differences affecting our feature functions, the final product was implemented within the Python environment.

5.3 Suitability for the Project

Approximate Q-Learning was suitable for our project due to its ability to handle large state spaces and generalize across similar states. Initial tests showed promising results, and after further tuning, its effectiveness was evident [1].

6 Agent Design and Implementation

In our Capture the Flag variation of Pacman, we developed two primary types of agents using Approximate Q-Learning: the Offensive Agent and the Defensive Agent. These agents were designed to work together to optimize their strategies for capturing pellets and defending against the opposing team. Below, we explain the design and implementation details of each agent and how they cooperate to achieve their objectives.

6.1 Offensive Agent

The Offensive Agent is responsible for capturing pellets on the enemy's side and delivering them back to the home side. The agent was designed using Approximate Q-Learning, leveraging a set of carefully chosen features to guide its decision-making process. The agent uses these features to approximate the Q-values for different state-action pairs, allowing it to make informed decisions [7].

6.1.1 Features

The features used by the Offensive Agent include:

- **Bias**: A constant feature to help the agent learn a baseline value.
- **Distance to Nearest Pellet**: Encourages the agent to move towards pellets.
- **Eats Food**: Detects if the agent is about to eat a pellet.
- **Distance to Nearest Capsule**: Encourages the agent to move towards power-up capsules.
- **Eats Capsule**: Detects if the agent is about to eat a power-up capsule.
- **Distance to Home**: Encourages the agent to return home when carrying pellets.
- **Number of Ghosts Within Two Steps**: Counts the number of ghosts within two steps of the agent.

6.1.2 Strategy

The Offensive Agent employs a balance of exploration and exploitation through an ϵ-greedy policy. Initially, the agent explores the environment to gather information about the optimal strategies. As the training progresses, it exploits the learned Q-values to make more informed decisions. The agent prioritizes capturing pellets and returning them safely while avoiding ghosts when carrying pellets.

6.2 Defensive Agent

The Defensive Agent aims to protect its side by preventing the opposing team's Pacman agents from capturing pellets. This agent also uses Approximate Q-Learning, with features designed to detect and intercept enemy agents.

6.2.1 Features

The features used by the Defensive Agent include:

- **Number of Invaders**: Counts the number of enemy Pacmen in the agent's territory.
- **Distance to Invaders**: Measures the distance to the nearest invader to prioritize closer threats.
- **On Defense**: Indicates whether the agent is currently a ghost and encourages this behavior since this is a defensive agent.

- **Stop and Reverse Penalties**: Discourages the agent from stopping or reversing direction unless strategically necessary.

6.2.2 Strategy

The Defensive Agent's strategy involves patrolling its side and intercepting enemy Pacmen. It prioritizes actions that bring it closer to invaders and deters them from capturing pellets. By continuously updating its Q-values based on the observed state transitions and rewards, the agent improves its defensive tactics over time.

6.3 Cooperation Between Agents

The offensive and defensive agents work together to achieve the team's objectives. We first trained them separately to have a good basic offensive and defensive agent and then trained them for 100 episodes together to ensure their cooperation. The Offensive Agent focuses on capturing and returning pellets, while the Defensive Agent protects the home side from invaders. This division of roles allows the team to balance offense and defense effectively. Communication between agents is implicit through their learned behaviors and the shared game state, enabling them to coordinate their actions without explicit communication protocols.

7 Evaluation and Results

To test the efficacy of our newly developed Approximate Q-Learning agents, we conducted a series of games where our team competed against a team of greedy agents. The opposing team consisted of a greedy offensive agent and a greedy defensive agent. The greedy offensive agent had a straightforward strategy: collect pellets and return home. The greedy defensive agent remained on its side, targeting any invaders.

Our Approximate Q-Learning team developed a more sophisticated strategy through learning. The defensive agent patrolled the border, ready to intercept and eliminate any invaders immediately. Meanwhile, the offensive agent played conservatively, collecting a few pellets at a time and returning them home quickly. This cautious approach minimized the risk of being captured by the enemy's ghosts.

We evaluated the performance of the two teams by running 100 games. The results demonstrated the effectiveness of our Approximate Q-Learning agents. Our team won 84 out of the 100 games, tied 14 times, and lost only 2 games, indicating that our learning-based agents significantly outperformed the greedy agents. The initial Q-Learning and PPO agents that we created couldn't even beat the greedy agent team once!

Metric	Value
Blue Win Rate	84%
Tie Rate	14%
Red Win Rate	2%
Average Blue Team Win Margin	6.18 Pellets

Table 1: Performance Metrics of Approximate Q-Learning Agents

The successful deployment of our Approximate Q-Learning agents in this competitive scenario demonstrates the potential of reinforcement learning techniques in creating intelligent, adaptive agents capable of outperforming traditional rule-based approaches. Our findings align with existing research, such as the work by Mnih et al. (2013) on deep Q-networks in Atari games, which showcased how Q-learning can achieve superior performance in complex environments [7].

Our evaluation and results underscore the effectiveness of Approximate Q-Learning in game-based simulations and beyond.

8 Future Work

While our project has demonstrated the efficacy of Approximate Q-Learning in developing intelligent agents for a Capture The Flag variation of Pacman, there are several avenues for future work to further enhance and expand our research.

- **Enhance Team-Based Strategies:** Our current implementation focuses mainly on individual agent behaviors. Future research can extend this to incorporate more sophisticated team-based strategies, finding unique ways to defeat opponents.
- **Integrate Advanced Learning Techniques:** Future work can explore integrating advanced learning techniques such as deep Q-networks (DQN) and Double DQN to further enhance the learning capabilities of our agents. These techniques have shown significant improvements in learning efficiency and performance in various environments, which could lead to even more intelligent and adaptive behaviors in our Pacman agents.
- **Explore Curriculum Learning:** Lastly, to improve the generalization capabilities of our agents, we can investigate the use of curriculum learning. This involves training agents on simpler tasks before gradually increasing the complexity, leading to more robust learning outcomes.

9 Conclusion

The Capture The Flag variation of Pacman project demonstrates the significant potential of reinforcement learning (RL) algorithms in developing adaptive and intelligent game agents. By integrating Q-Learning and Proximal Policy Optimization (PPO) with game development frameworks like Unity, we have created a dynamic platform that effectively showcases complex agent behaviors and interactions.

The project not only advances our understanding of RL and its applications but also sets the stage for further explorations into optimizing RL algorithms, developing more complex multi-agent interactions, and applying these techniques to a wider array of disciplines. The promising results from our Pacman agents suggest that with continued refinement and exploration, RL can play a crucial role in the next generation of intelligent systems.

References

[1] Yuxi Li. "Deep reinforcement learning: An overview". In: *arXiv preprint arXiv:1701.07274* (2017).

[2] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[3] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[4] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[5] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *arXiv preprint arXiv:1712.01815* (2018).

[6] Yuanhao Li et al. "A survey on deep reinforcement learning". In: *arXiv preprint arXiv:1906.00979* (2019).

[7] Volodymyr Mnih et al. "Playing Atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).