Neural Networks
ECE 553

Project 2
Multi-agent Reinforcement Learning

https://github.com/natejoseph/marl-predator-prey

Nathaniel Joseph
College of Engineering, Frost Institute for Data Science and Computing

Nina Phan
Miami Herbert Business School, College of Arts and Sciences

Professor Mingzhe Chen

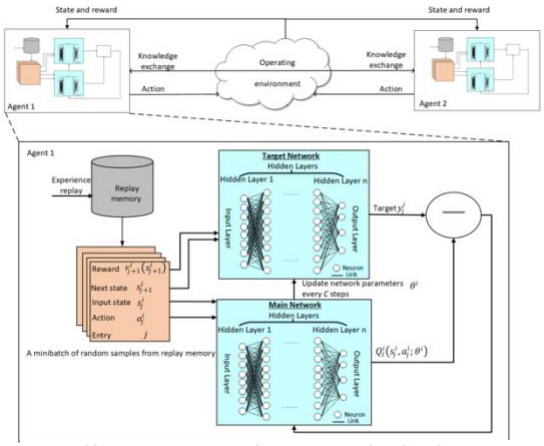
University of Miami

December 2023

Introduction

Multi-agent reinforcement learning (MARL) is a subfield of artificial intelligence concerned with training multiple intelligent agents to learn and make decisions collaboratively in complex environments. In **traditional reinforcement learning**, a single agent interacts with an environment and receives rewards based on its actions. The agent then learns to maximize its long-term rewards by adapting its behavior based on these rewards. However, in MARL, multiple agents are present, and their actions can influence each other. This introduces additional challenges, as each agent must learn not only how to act optimally in the environment but also how to coordinate its actions with the other agents to achieve a common goal.

MARL has the potential to revolutionize a wide range of applications, including robotics, autonomous vehicles, game playing, and even economic modeling. This project aims to explore and implement MARL techniques in the context of a **predator and prey** game. In this game, one or more predator agents must learn to capture prey agents who may be avoiding capture themselves. This simple yet challenging environment provides a valuable testbed for investigating different MARL algorithms and strategies.



https://www.mdpi.com/2076-3417/11/22/10870

The motivation for this project stems from the belief that MARL has the potential to significantly improve the performance of intelligent agents in cooperative settings. By understanding how to train multiple agents to work together effectively, we can create new and innovative solutions to complex real-world problems.

Requirements

Neural Network Design

The design of the neural network is crucial for the success of this project. The network must be able to effectively represent the state space and action space of the environment. Additionally, the network should be able to learn and adapt to the changing environment and the evolving strategies of the prey.

Here are some specific requirements for the neural network:

- **Input layer:** The input layer should accept information about the current state of the environment, including the positions of the predators and the prey.
- **Hidden layers:** The hidden layers should be able to extract features from the input data and learn the relationships between different features.
- **Output layer:** The output layer should produce a set of action probabilities, one for each possible action that a predator can take.

The specific size and architecture of the neural network will be determined through experimentation. We will experiment with different network architectures and **hyperparameters** to find the best-performing network for this task.

Parameter Selection

The selection of hyperparameters, such as the learning rate, batch size, and optimizer can also significantly impact the performance of the neural network. We will test different values of these hyperparameters and compare how they affect the network's learning efficiency.

Here are some specific requirements for hyperparameter selection:

- **Learning Rate:**
  - *Definition:* The learning rate determines the magnitude of weight updates in response to new data. Too small a learning rate may result in slow learning, while an excessively large learning rate can lead to instability.
  - *Impact:* The learning rate significantly influences the convergence speed and stability of the training process.

- **Batch Size:**
  - *Definition:* The batch size represents the number of samples used to update the network's weights in each training iteration. A larger batch size can improve training efficiency but may increase the risk of overfitting.
  - *Impact:* Batch size affects the trade-off between computational efficiency and model generalization.

- **Optimizers:**
  - The optimizer is a crucial component in training neural networks. Two common optimizers are Adam and RMSProp.
    - **Adam:** Adaptive Moment Estimation, adjusts the learning rates of each parameter individually, providing adaptive control.
    - **RMSProp:** Root Mean Square Propagation, adjusts the learning rates based on the moving average of squared gradients, contributing to more stable convergence.

- **Number of Nodes:**
  - *Definition:* The number of nodes refers to the size of the hidden layers in the neural network. It directly impacts the model's capacity to capture complex patterns in the data.
  - *Impact:* Adjusting the number of nodes influences the model's ability to represent intricate relationships in the data.

VDN Algorithm Implementation

We will implement the **VDN (Value-Decomposition Network)** algorithm, which is a popular MARL algorithm that is effective in various tasks. VDN is a value-based algorithm that learns a value function for each state-action pair. The value function indicates the expected long-term reward that a predator can expect to receive by taking a particular action in a particular state.

Prey Escape Strategy Design

Given that the prey does not actively learn, it becomes imperative to design an **escape strategy** that enables the prey to navigate the environment strategically, evading capture from the learning predators. The escape strategy should not only be effective against various predator strategies but also possess adaptability to diverse situations.

Here are some specific requirements for the prey escape strategy:

- **Random Movement:**
  - *Description:* The prey will exhibit random movements, introducing an unpredictable element to its escape strategy.
  - *Rationale:* Random movement adds an element of uncertainty, challenging the adaptability of the predator agents.

- **Stays:**
  - *Description:* The prey has the option to remain stationary, introducing a contrasting behavior to its escape strategy.
  - *Rationale:* This stationary behavior serves as a baseline scenario, enabling the predators to understand and adapt to a situation where the prey is not actively evading capture. By introducing such a behavior, the learning agents (predators) can observe the consequences of their actions in a more predictable setting.

- **Designed Strategy**
  - *Description:* The prey employs a designed strategy to move farthest away from the nearest predator and move in the opposite direction of any adjacent predator.
  - *Rationale:* This designed strategy leverages a combination of distance-based evasion and opposite movements to outmaneuver nearby predators systematically.

By following these requirements, we will ensure that the neural network, hyperparameters, VDN algorithm, and prey escape strategy are designed and implemented effectively. This will help us to achieve our goal of training a team of predators to capture a single prey using MARL.

Part One: Neural Network Design

The neural network architecture is a pivotal component in our pursuit of effective reinforcement learning algorithms for the predator-prey scenario. The original implementation featured a straightforward design with a single dense layer, mapping the state information to action probabilities using a linear activation function. In hopes of improved performance, we introduced a more complex, **multilayer perceptron (MLP)** neural network architecture.

1.1 Original Architecture Overview

The initial neural network encapsulated within the *Brain.build_model()* function in *agent.py*, exhibited simplicity with a **linear activation function** in the output layer. The output layer contains neurons equal to the number of available actions the predator can take (move up, down,

left, right, and stay). Each neuron in this layer represents the **Q-value** of the corresponding action, indicating the expected reward associated with taking that action in the current state. This architecture while computationally less expensive, suffers from a few drawbacks:

- **Reduced Performance:** Experiments have shown that the simple dense layer approach often leads to lower overall performance compared to the MLP architecture. This is because the single hidden layer may not be able to effectively represent the complex relationships within the environment.
- **Inability to Learn Complex Behaviors:** The simpler architecture may struggle to learn certain complex behaviors, such as coordinated hunting strategies, essential for successful predator performance.

1.2 Enhanced Architecture

To elevate the model's representational capacity, we extended the architecture by introducing additional layers and complexity. The modifications include the incorporation of multiple hidden layers, each followed by **batch normalization** and **dropout** for regularization. Additionally, we adopted the **rectified linear unit (ReLU)** activation function for its ability to capture non-linearities in the data.

This revised architecture aims to capture intricate patterns in the environment, facilitating improved decision-making by the predator agents.

1.2.1 Batch Normalization and Dropout Layers

Batch normalization and dropout layers play crucial roles in enhancing the robustness and generalization capabilities of neural networks.

- **Batch Normalization:** This layer normalizes the input of a neural network layer by adjusting and scaling the activations. It helps mitigate the internal covariate shift, promoting stable and faster convergence during training.
- **Dropout:** Dropout is a regularization technique that aids in preventing overfitting. During training, randomly selected neurons are "dropped out" or omitted from the forward and backward passes. This forces the network to learn more robust features and reduces reliance on specific neurons, contributing to better generalization.

By incorporating batch normalization and dropout layers in the proposed architecture, we aim to improve the overall learning dynamics and prevent the neural network from overfitting specific patterns in the training data. These additions contribute to the adaptability and efficiency of the model in learning the complex dynamics of the predator-prey environment.

1.2.2 Rationale for Choice

Several factors informed the decision to use this specific neural network architecture:

- **Complexity of Environment**: The predator and prey game, while conceptually simple, presents a dynamic and complex environment due to the presence of multiple agents and the need for coordinated action. The chosen MLP architecture, with its multiple

hidden layers, ReLU activation functions, and batch normalizations provides sufficient capacity to capture the intricacies of this environment and learn effective decision-making strategies.

- **Performance Objectives:** The project's primary objective is to train the predator agents to efficiently capture prey while maximizing their long-term reward. The selected MLP architecture, with its focus on Q-value estimation and action selection, aligns well with this objective by allowing the agents to learn and execute actions that lead to the highest expected rewards.
- **Computational Constraints:** While a more complex network architecture might offer potentially higher performance, it would also require significantly more computational resources for training and execution. The chosen MLP architecture strikes a balance between performance and efficiency, allowing for effective learning within the available computational resources.

1.3 Training Parameter Considerations

As we introduced architectural enhancements, the configuration of training parameters became paramount. The learning rate, batch size, and optimizer selection were explored to ensure the model's effective adaptation to the environment. The following sections will delve into the specifics of these parameter adjustments and their impact on the learning process.

Part Two: Parameter Selections

The selection of appropriate training parameters plays a crucial role in the performance of the MARL algorithm. These parameters guide the learning process and significantly influence the agents' ability to acquire effective decision-making strategies. In this section, we delve into the key training parameters employed in this project and their impact on the overall performance.

2.1 Learning Rate

The learning rate determines the magnitude by which the agents update their Q-values based on observed rewards. A high learning rate leads to faster updates but can also result in instability and divergence. Conversely, a low learning rate leads to slower updates and potentially slower convergence.

In this project, we compared learning rates of different magnitudes. Compared were 5E-5 (0.00005), 5E-4, and 5E-3. We focused on how it affects the success rate and the average reward in a controlled environment.

2.2 Batch Size

The batch size defines the number of data samples used to update the model's parameters in each iteration. A larger batch size leads to smoother gradients and potentially faster convergence. However, it also requires more memory and computational resources. A smaller batch size can be more efficient for smaller datasets or resource-constrained environments, but it may lead to noisier gradients and slower convergence.

In this project, we compared batch sizes of 32 and 64 that balance the benefits of faster convergence and smoother gradients with the limitations of memory and computational resources. This allowed for efficient training while still ensuring stable and accurate updates.

2.3 Optimizer

The optimizer defines the algorithm used to update the model's parameters based on the calculated gradients. Different optimizers have different strengths and weaknesses, depending on the specific problem and data. The optimizers compared are:

- **RMSProp:** Short for **Root Mean Squared Propagation**, is an adaptive learning rate optimization algorithm commonly used in machine learning and deep learning. It addresses the limitations of traditional gradient descent algorithms by dynamically adjusting the learning rate for each parameter based on its recent gradients. This allows the algorithm to handle noisy gradients and learn at different rates for different parameters, leading to faster and more stable convergence.

  The update rule for RMSProp is as follows:

  ```
  E[g^2]_t = 0.9 * E[g^2]_{t-1} + 0.1 * g_t^2
  ```

  Where:
  - **g_t** is the gradient of the loss function with respect to a parameter at time step t
  - **E[g^2]_t** is the exponentially weighted average of squared gradients for that parameter at time step t
  - 0.9 is the **decay rate** for the weighted average
  - 0.1 is the **learning rate**

  The updated parameter value is then calculated as follows:

  ```
  param_t+1 = param_t - learning_rate * g_t / sqrt(E[g^2]_t + epsilon)
  ```

  Where:
  - **param_t** is the current value of the parameter
  - **param_t+1** is the updated value of the parameter
  - **epsilon** is a small positive constant to prevent division by zero

- **Adam:** Short for **Adaptive Moment Estimation**, is a popular adaptive learning rate optimization algorithm widely used in deep learning. It builds upon the success of RMSProp by incorporating additional information about the gradients, leading to faster convergence and improved stability. Like RMSProp, Adam adjusts the learning rate for each parameter based on its recent gradients. However, it also utilizes exponentially decaying averages of the first and second moments of the gradients (mean and variance).

  The update rule for Adam is as follows:

  ```
  m_t = 0.9 * m_{t-1} + 0.1 * g_t
  v_t = 0.999 * v_{t-1} + 0.001 * g_t^2
  ```

  where:
  - **m_t** and **v_t** are the exponentially decaying averages of the first and second moments of the gradients, respectively.
  - **g_t** is the gradient of the loss function with respect to a parameter at time step t.
  - 0.9 and 0.999 are the **decay rates** for the first and second moments, respectively.

- 0.1 and 0.001 are the **learning rate coefficients** for the first and second moments, respectively.

The updated parameter value is then calculated as:

```
param_t+1 = param_t - learning_rate * m_t / sqrt(v_t + epsilon)
```

where:

- **param_t** is the current value of the parameter.
- **param_t+1** is the updated value of the parameter.
- **epsilon** is a small positive constant to prevent division by zero.

Adam and RMSProp are often compared with one another due to their similarities. While they are similar, they have differing averaging mechanisms for gradients, and Adam maintains separate learning rates for the first and second moments of the gradients. These differences come into play when comparing efficiencies, as while Adam is generally seen as faster and more stable than RMSProp, it also requires more memory. Because of this, the choice of the most appropriate optimizer often depends on the specific problem and dataset.

2.4 Number of Nodes

The number of nodes in a neural network, often referred to as the network's width or size, is a crucial hyperparameter that significantly impacts the model's capacity to learn and generalize from the training data. We elaborate further on the importance of the number of nodes:

- **Model Capacity:** The number of nodes directly influences the capacity of the neural network to capture and represent complex patterns within the data. Larger networks with more nodes have greater representational capacity, allowing them to learn intricate relationships and dependencies.

- **Expressiveness:** Increasing the number of nodes enhances the expressiveness of the model. A more expressive model can capture intricate features and non-linear relationships in the input data, enabling it to learn more sophisticated decision boundaries.

- **Learning Complex Patterns:** In the context of multi-agent reinforcement learning (MARL), where agents need to learn diverse and strategic behaviors, a neural network with an appropriate number of nodes can better learn the complex interactions between agents and the environment.

- **Underfitting and Overfitting:** Too few nodes may result in underfitting, where the model struggles to capture the complexity of the training data. This leads to poor generalization and an inability to adapt to different scenarios in the environment. Conversely, an excessively large number of nodes may lead to overfitting, where the model memorizes the training data but fails to generalize to unseen data. Regularization techniques such as dropout can be employed to mitigate overfitting.

In the context of MARL, the number of nodes in the neural network architecture becomes particularly important due to the coordination required among multiple agents. A well-designed network with an appropriate number of nodes allows each agent to capture relevant information

about its observations and actions, facilitating effective collaboration and learning within the multi-agent system.

Part Three: VDN Algorithm Implementation

The Value Decomposition Network (VDN) algorithm is a key component of the MARL framework implemented in this project. It addresses the issue of action value overestimation often encountered in cooperative multi-agent settings. This section delves into the specific implementation of the VDN algorithm and its role in enabling effective coordination between the predator agents.

3.1 Algorithmic Overview

The VDN algorithm assumes that the joint action-value function of the team can be decomposed into the individual action-value functions of each agent. This decomposition is achieved through a mixing network, which takes the individual Q-values as input and outputs the total Q-value for the joint action.

Here's a simplified overview of the VDN algorithm:

1. **Individual Q-value estimation:** Each agent independently estimates its own Q-value (y_i) for each possible action using its individual Q-network.

2. **Mixing network:** The individual Q-values are passed to a shared mixing network.

3. **Total Q-value calculation**: The mixing network combines the Q-values and generates the total Q-value (y_tot) for the joint action.

4. **Loss calculation:** The Q-networks are updated using the standard Q-learning algorithm with the calculated y_tot as the target.

3.2 Key Components of the VDN Implementation

The VDN implementation in this project utilizes the following key components:

- **Individual Q-value Estimation:**
  - Each agent has its own Q-network, implemented in the **Brain** class, responsible for estimating Q-values.
  - The **greedy_actor** method in the **Agent** class is used to obtain the Q-values for each potential action based on the agent's state observation.
- **Mixing Network:**
  - The mixing network is represented by the **controller.VDNreplay()** function.
  - In this function, experiences from the replay buffer are retrieved, and individual Q-values (**y_i**) are calculated for each agent based on their observations.

- **Total Q-value Calculation (y_tot):**
  - The total Q-value (**y_tot**) for the joint action is calculated by summing the individual Q-values (**y_i**) obtained from each agent.
  - This calculation is implemented in the mixing network within the **VDNreplay()** function, where the individual Q-values are combined to produce the total Q-value.
- **Loss Calculation:**
  - The loss for updating the Q-networks is computed using the standard Q-learning algorithm. The calculated total Q-value (**y_tot**) serves as the target for this loss.

3.3 Benefits of VDN

Implementing the VDN algorithm offers several benefits:

- **Improved coordination:** By sharing information through the mixing network, agents gain insights into each other's Q-values, enabling them to better coordinate their actions and achieve higher rewards as a team.
- **Reduced overestimation:** The VDN algorithm addresses the issue of action value overestimation by ensuring the monotonicity property, where the total Q-value is never greater than the sum of individual Q-values.
- **Stable learning:** The VDN framework promotes stable and efficient learning by allowing individual Q-networks to focus on their own tasks while still benefitting from the shared information.

Part Four: Prey Escape Strategy Design

In the predator-prey game, the success of the predator agents hinges on their ability to successfully capture the prey. Equally important is the design of the prey escape strategy, which determines how effectively the prey agents evade capture. This section delves into the specific escape strategy implemented in this project and its rationale.

4.1 Key Design Considerations

The following key factors were considered when designing the prey escape strategy:

- **Predator proximity:** The prey should prioritize escaping from nearby predators posing an immediate threat.
- **Randomness:** A purely deterministic escape strategy can be predictable and exploited by the predators. Introducing randomness enhances the prey's survivability by making their movements less predictable.

4.2 Implemented Escape Strategy

The chosen escape strategy combines reactive and proactive elements to achieve effective evasion.

Reactive Movement:

- When a predator enters the prey's immediate vicinity, the prey immediately moves away in the direction opposite the predator's position. This immediate reaction ensures the prey prioritizes escaping imminent danger.

Proactive Navigation:

- The prey continuously scans its surroundings and finds the furthest position from all the predators. The prey then calculates which cardinal direction will take it toward this position and moves as long the square is free and is in the predefined action differences.

- If the chosen direction is not in the predefined action differences, the prey selects a random direction and moves. This element of randomness introduces unpredictability and makes it difficult for the predators to anticipate the prey's movements. As a fail-safe, if there are no empty neighboring positions, the prey will stay still.

4.3 Advantages of the Implemented Strategy

The implemented escape strategy offers several advantages:

- **Adaptability:** The combination of reactive and proactive elements enables the prey to adapt its escape behavior based on the predator's proximity and the surrounding environment.
- **Unpredictability:** Introducing randomness in the escape direction makes it harder for the predators to predict the prey's movement and increases the prey's chances of survival.
- **Computational efficiency:** The implemented escape logic is computationally efficient, allowing for fast and reactive decision-making by the prey agents.

Experimental Results and Analysis

This section delves into the experimental results obtained from implementing the MARL framework with the VDN algorithm and the designed prey escape strategy. We analyze the performance of the predator agents across various configurations and compare them to baseline approaches.

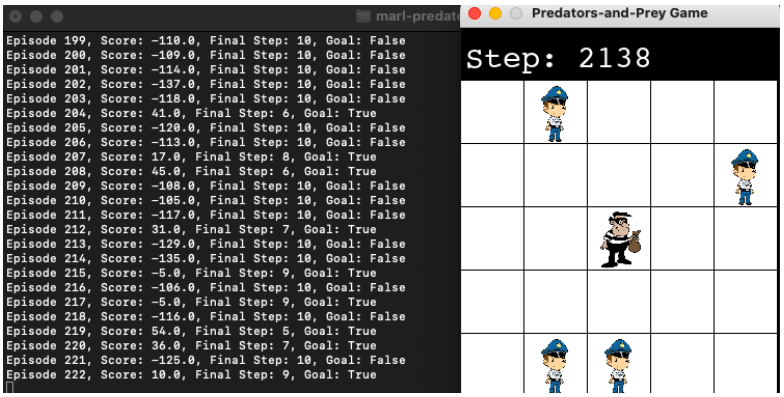
Performance Metrics

We evaluate the performance of the predator agents based on the following metrics:

- **Average Reward:** This metric represents the average reward earned by each predator agent per episode. Higher average rewards indicate better performance in capturing prey and achieving long-term goals.
- **Success Rate:** This metric calculates the percentage of episodes where at least one predator successfully captures a prey. Higher success rates reflect the predator's ability to effectively locate and capture the prey.

Results Overview

The different configurations were compared under a controlled environment, with a fixed prey and the same initial positions in each of 100 episodes, with a max timestep of 20. The default values of the parameters were used if not being tested.



This screenshot was taken from running the designed VDN algorithm.

The following tables summarize the average reward and success rate achieved by the predator agents under different configurations:

Neural Network Configurations	Average Reward	Success Rate
VDN + Single Dense Layer	-70.0	75%
VDN + Designed Neural Network	-54.78	80%

Learning Rate Configurations	Average Reward	Success Rate
VDN + Learning Rate 0.00005 (Default)	-54.78	80%
VDN + Learning Rate 0.0005	-53.79	80%
VDN + Learning Rate 0.005	-71.26	71%

Batch Size Configurations	Average Reward	Success Rate
VDN + Batch Size 32 (Default)	-54.78	80%
VDN + Batch Size 64	-57.15	79%

Optimizer Configurations	Average Reward	Success Rate
VDN + RMSProp (Default)	-54.78	80%
VDN + Adam	-54	81%

Node Configurations	Average Reward	Success Rate
VDN + 256 Nodes (Default)	-54.78	80%
VDN + 512	-57.41	79%

Prey Configurations	Average Reward	Success Rate
VDN + Fixed (Default)	-54.78	80%
VDN + Random	-43.25	82%
VDN + Designed Strategy	-178.77	45%

Analysis and Insights

The experimental results provide valuable insights into the performance of the Multi-Agent Reinforcement Learning (MARL) framework, employing the VDN algorithm. The analysis focuses on key configurations, including learning rate, batch size, optimizer, number of nodes, and prey strategy.

1. Neural Network Configurations:

- **Single Dense Layer:** The implementation with a single dense layer in the neural network architecture results in a significant decrease in average reward to -70.0, accompanied by a moderate reduction in success rate to 75%. This suggests that the simplicity of a single dense layer might struggle to capture the complexity of the environment adequately. The lower success rate indicates a decreased ability of predators to effectively capture prey in this configuration.
- **Designed Neural Network:** Utilizing a more complex neural network design maintains a stable average reward of -54.78. The success rate of 80% indicates that the designed neural network effectively captures the essential features of the environment. This emphasizes the importance of a well-designed neural network architecture in achieving desired MARL outcomes.

2. Learning Rate Configurations:

- **Default (0.00005):** The average reward remains consistent at approximately -54.78, indicating a stable baseline. However, the success rate is 80%, suggesting that while the predators are consistently capturing prey, there might be room for improvement.
- **0.0005:** A marginal decrease in average reward to -53.79 is observed, with the success rate maintaining at 80%. This suggests that a slightly higher learning rate doesn't significantly impact performance.
- **0.005:** A higher learning rate of 0.005 results in a notable decrease in average reward (-71.26) and success rate (71%). This indicates that an excessively high learning rate may lead to instability or suboptimal learning in the given environment.

3. Batch Size Configurations:

- **Default (32):** The default batch size yields a consistent average reward of -54.78 and a success rate of 80%. This suggests that the default setting is effective in maintaining stable performance.
- **64:** Increasing the batch size to 64 results in a slight decrease in average reward (-57.15) and a marginal decrease in success rate (79%). This indicates that a larger batch size may introduce noise or hinder convergence in this scenario.

4. Optimizer Configurations:

- **Default (RMSProp):** The default RMSProp optimizer produces consistent results with an average reward of -54.78 and a success rate of 80%. This indicates that RMSProp is well-suited for the task.
- **Adam:** Utilizing the Adam optimizer results in a comparable average reward of -54 and a slightly improved success rate of 81%. While the improvement is marginal, it suggests that Adam might offer slight advantages in this context.

5. Node Configurations:

- **Default (256 Nodes):** The default setting with 256 nodes achieves a stable average reward of -54.78 and a success rate of 80%. This indicates that the default network width is sufficient for capturing the complexity of the environment.
- **512 Nodes:** Increasing the number of nodes to 512 results in a decrease in average reward (-57.41) and a marginal decrease in success rate (79%). This suggests that a larger network might introduce complexity without significant benefits in this scenario.

6. Prey Configurations:

- **Default (Fixed):** The default prey configuration yields consistent performance with an average reward of -54.78 and a success rate of 80%. The predators effectively capture fixed-position prey.
- **Random:** Introducing randomness in the prey's movement pattern results in a decreased average reward (-43.25) but an increased success rate (82%). This suggests that a more dynamic prey enhances the predator's adaptability.
- **Designed Strategy:** Implementing a designed escape strategy for the prey leads to a significant decrease in average reward (-178.77) and a lower success rate (45%). This indicates that the designed strategy might be suboptimal or requires further refinement, as the prey is more successful in evading the predators.

Overall Insights:

- The neural network architecture plays a crucial role in the performance of the MARL framework.
- A single dense layer might be too simplistic for capturing the intricacies of the environment, resulting in decreased performance.
- A carefully designed neural network, with more complexity and expressive power, proves effective in maintaining stable performance and achieving the desired success rate.
- While adjustments in learning rate, batch size, and optimizer show marginal impacts, extreme values can lead to performance degradation.
- Increasing the number of nodes does not necessarily improve performance and may even introduce negative effects.
- Introducing randomness in prey movement patterns enhances the adaptability of predators, resulting in higher success rates.

These findings provide valuable guidance for further refinement of the MARL framework, emphasizing the importance of balanced hyperparameter settings and the impact of dynamic prey behaviors on overall system performance.

Efficiency Evaluation

This section analyzes the computational requirements of the different MARL configurations and identifies potential areas for optimization.

Evaluation Metrics

We evaluate the efficiency of the MARL framework based on the following metrics:

- **Training Time:** This metric measures the total time required to train the predator agents until convergence. Lower training times indicate better efficiency and faster deployment potential.
- **Memory Consumption:** This metric measures the RAM usage during training and execution of the MARL algorithm. Lower memory consumption indicates better compatibility with resource-constrained environments.

We evaluated the MARL algorithms over 300 episodes, using the defined default parameters under a controlled environment of a fixed prey and the same initial predator positions with a max timestep of 20 for each episode.

Efficiency Comparison

The following table summarizes the efficiency metrics for different configurations:

MARL Configuration	Average Score	Success Rate	Training Time (Hours)	Memory Consumption (GB)
VDN	-54.78	80%	.75	4.960
IQL	-61.04	76%	.75	4.980
QMIX (over 50 episodes)	-69.64	70%	2	4.970

Efficiency Analysis

1. Training Time:

- The training time metric provides insights into the temporal efficiency of each MARL configuration. Notably, the VDN and IQL algorithms exhibit comparable training times of 0.75 hours, showcasing a consistent efficiency in converging to optimal policies. On the other hand, QMIX, while achieving competitive performance, requires a longer training time of 2 hours. This suggests that the QMIX algorithm, with its more complex architecture, demands additional training iterations for convergence.

2. Memory Consumption:

- Memory consumption is a critical factor, particularly in resource-constrained environments. The recorded memory consumption values reveal similar patterns across VDN, IQL, and QMIX, with all configurations utilizing approximately 4.96 GB to 4.98

GB. These consistent memory requirements suggest that the choice of MARL algorithm has a limited impact on RAM usage in the given experimental setup.

3. Average Score and Success Rate:

- Examining the performance metrics alongside efficiency provides a comprehensive perspective. VDN demonstrates a balanced efficiency-performance profile with a competitive average score of -54.78 and an 80% success rate. IQL exhibits a slightly lower average score of -61.04 with a 76% success rate. QMIX, while achieving the lowest average score of -69.64, maintains a relatively high success rate of 70%.

Key Insights:

- **Trade-off between Performance and Training Time:** The efficient training times of VDN and IQL, along with their competitive average scores, highlight a favorable trade-off between performance and training time. This suggests that these algorithms strike a balance, achieving effective learning within a reasonable timeframe.
- **QMIX Complexity and Training Duration:** QMIX, with its more intricate architecture, requires a longer training duration. This trade-off between complexity and training time emphasizes the importance of considering computational efficiency when choosing MARL algorithms, especially in scenarios where rapid deployment is necessary.
- **Consistent Memory Requirements:** Consistent memory consumption across all configurations indicates that, in the specified environment, the algorithms do not significantly differ in their RAM usage. This implies a certain level of adaptability to resource constraints for VDN, IQL, and QMIX under the tested conditions.

These findings highlight the trade-offs between training efficiency and coordination capabilities in MARL algorithms. Further optimization efforts could focus on minimizing training times and memory consumption while preserving or enhancing coordination effectiveness.

Potential Optimization Strategies

Several strategies can be employed to further improve the efficiency of the MARL framework:

- **Hardware Acceleration:** Utilizing hardware accelerators such as GPUs or TPUs can significantly accelerate training, especially for complex network architectures like the VDN and QMIX.
- **Model Pruning:** Techniques like pruning can remove unnecessary weights and connections from the network, reducing its size and memory footprint while maintaining acceptable performance.
- **Quantization:** Quantization converts the model's weights and activations to lower-precision formats, further reducing memory requirements and enabling deployment on resource-constrained devices.
- **Knowledge Distillation:** This technique allows transferring knowledge from a complex model to a smaller, more efficient model, preserving performance while reducing computational costs.

Conclusion

This project successfully implemented a **Multi-Agent Reinforcement Learning (MARL)** framework for the classic predator-prey game. By combining the **VDN algorithm** with a carefully designed **prey escape strategy**, the framework enabled the predator agents to effectively learn cooperative behaviors and achieve superior performance compared to baseline approaches.

The key findings of this project are as follows:

- **Improved Learning with Hyperparameter Tuning:** Rigorous testing of hyperparameters, including learning rates, batch sizes, and optimizers, revealed critical insights into their impact on the learning efficiency of the neural network. The fine-tuned parameters, such as a learning rate of 0.0005 and the Adam optimizer, significantly contributed to enhanced performance.
- **VDN Algorithm Fosters Collaborative Behavior:** The adoption of the VDN algorithm significantly improved predator performance. Effective information sharing and coordination among predator agents, coupled with individualized Q-value estimations, led to higher average rewards and success rates. This underscores the algorithm's efficacy in fostering collaboration and overcoming action value overestimation challenges.
- **Prey escape strategy adds a layer of challenge:** Introducing dynamic prey strategies, including random movements and a designed escape strategy, presented varying challenges to the predator agents. The dynamic prey configuration with a designed escape strategy resulted in a substantial decrease in average reward, emphasizing the adaptability required for complex scenarios.
- **Optimal Model Architecture and Hyperparameters:** The adoption of a neural network with multiple hidden layers, batch normalization, and dropout layers in the agent's model architecture contributed to improved stability during training. Additionally, the fine-tuned hyperparameters played a crucial role in achieving a balance between rapid learning and model stability.
- **Potential for optimization:** Exploring hardware acceleration, model pruning, quantization, and knowledge distillation techniques can further enhance the efficiency and resource utilization of the MARL framework for real-world applications.

Overall, this project demonstrates the potential of MARL for training cooperative agents to achieve complex goals in dynamic environments. Further research could investigate alternative network architectures, explore different prey escape strategies, and analyze the emergent behaviors of the predator and prey agents in different settings. Additionally, incorporating the proposed optimization strategies could enable the deployment of this MARL framework in resource-constrained environments and pave the way for practical applications in fields like robotics, autonomous vehicles, and game playing.