Multi-Agent Reinforcement Learning model implementation to master Knights Archers Zombies game

ECE-GY-7123: Deep Learning, Fall 2021

Project Report

Harini Appansrinivasan (ha1642) and Vaibhav Mathur (vm2134)

Source code:

Our source code can be found in the below Github repository:
https://github.com/vaibhav117/MARL

1. Problem Statement:

Reinforcement learning (RL) is a kind of machine learning method where an agent perceives the state of the environment they are in through observations and takes actions, which causes the environment to transit into a new state. The quality of each transition is evaluated in terms of rewards. By trial and error, the aim of the agent is to learn an optimal policy that can lead to good actions and maximum rewards. By interacting with the environment, RL can be successfully applied to sequential decision-making tasks. Vanilla reinforcement learning works with a single agent in an environment, seeking to maximize the total reward in that environment. However, most of the successful RL applications, e.g., the games of Go and Poker, robotics, etc., involve the participation of more than a single agent. Multi-Agent Reinforcement Learning (MARL) studies how multiple agents in the same environment, collectively learn, collaborate and interact with each other with a certain perspective such as cooperative or competitive. In our project, our aim is to implement a Cooperative MARL model with the goal of mastering a game that involves multiple agents.

2. Literature Survey:

With the motivation of recent success on Deep Reinforcement Learning: Super-human level control on Atari games (which introduced DQN in 2015), mastering the game of Go (2016), routing (2018), etc. the importance and focus on multi-agent systems, and especially Deep MARL has increased.

There are several properties of the system that are important in modeling a multi-agent system: centralized or decentralized control, fully or partially observable environment, cooperative or competitive environment, etc. In our project, we have focused on the decentralized independent learning MARL problem with a cooperative goal. This idea was originally proposed in 1993 by Ming Tan, introducing Independent Q-Learning, IQL (an extension of Q-learning). Many other papers related to Independent Learners were published later, like Tampuu et al. (2017), Foerster et al. (2017), Fuji et al. (2018).

In 2015, Mnih et al. introduced DQN algorithm, which used Experience Replay and Target Network to attain super-human level control on most of the Atari games. Vannila IQL uses a tabular version to store the Q-values. Tampuu et al. in 2017, extended this by combining both these ideas and using DQN algorithm instead of each single Q-learner.

Foerster et al. (2017) proposed two algorithms to stabilize the experience reply in IQL-type algorithms. Fuji et al. (2018) proposed to train one agent at a time, while fixing the policies of other agents in order to attain environment stability. This would result in a stationary environment from the view-point of the single agent.

Many new DQN based approaches have also been proposed. Some of them are: Double-DQN (Van Hasselt et al. 2016), Dueling double DQN (Wang et al. 2016), DRQN (Hausknecht and Stone 2015).

In 2019, Zhang et al. gave a comprehensive overview on the theoretical results, convergence and complexity analysis of MARL algorithms on Markov/stochastic games, and extensive-form games on competitive, cooperative, and mixed environments. In 2020, Nguyen et al. provided a review of deep MARL- focusing on non-stationary, partial observation space, continuous state and action spaces.

3. Environment:

PettingZoo is a Python library of diverse sets of multi-agent environments, it can be considered as an equivalent of a multi-agent version of OpenAI's Gym library. Butterfly environment that comes with PettingZoo is a set of Cooperative graphical games that require a high degree of coordination and learning of emergent behaviors to achieve an op-

timal policy. This can be solved by a Cooperative MARL model. Knights Archers Zombies (KAZ) is a game that comes as a part of Butterfly environment. It involves up to 4 agents (2 knights and 2 archers) working together to kill Zombies that walk from the top border of the screen down to the bottom border in unpredictable paths. Each agent can rotate clockwise or counter-clockwise, move forward or backward and can attack to kill the zombies. Each agent observes the environment as a square region around itself, with its own body at the center of the square. The observation is represented as a 512x512 pixel image around the agent, or in other words, a 16x16 agent sized space around the agent.

4. Overview of MARL algorithm:

Since KAZ game has a discrete action space, it can be solved by models such as Deep Q-network (DQN) or Multi-Agent Proximal Policy Optimization (MAPPO). DQN algorithm is built on the fundamentals of Q-Learning.

4.1 Q-Learning:

RL algorithms are based on a premise that all environments are Markov Decision Processes (MDP). MDPs have the property that their current state is sufficient to describe the future rewards. Based on the State Transition function, at a time instance t, the agent in a state of s_t takes an action a_t and the environment then returns the reward for that time step r_t and the next state s_{t+1} to the agent. As the agent continues this process till from the start to the end of an episode, it gives rise to trajectories, a sequence of states and actions: $s_0, a_0, r_0, s_1, a_1, r_1, ...$ Each episode can be considered as a logical end to the interaction with the environment, which in our case is when a Zombie reaches the bottom border of the environment. The agent tries to learn a policy that will maximize the cumulative rewards over each episode.

The **Future Discounted Reward**, which the agent tries to maximize, is given by:

$$G_t = \sum_{i=t}^{n} \gamma^{i-t}\, r_i$$

where

- γ is the Discount factor (a number between 0 and 1). γ can be regarded as encoding an increasing uncertainty about rewards that will be received in the future, that is, putting less priority on rewards farther in the future.

Q-learning achieves this is by computing the optimal state-action value function *(Q-function)*.

Q-learning is a typical off-policy value-based and model-free approach used for building a self-playing agent. It works by learning a *Q-function* that gives the *Expected future discounted Return* value or the *Q-value* of performing a particular action when at a particular state and following a fixed policy thereafter. A Q-table, that stores the Q-values for all possible state-action pairs, is maintained and and updated as the model learns more with every interaction.

The Q-function uses the Bellman equation. The **Q-value** is a summation of the Future Discounted Rewards that can be gathered by performing a particular action a_t from the current state s_t:

$$\boldsymbol{Q(s_t, a_t) = \mathbb{E}[G_t \mid s_t, a_t]}$$

Q-learning Algorithm process is as follows:

- Initialize Q-table
- Choose and perform an action when at a particular state
- Measure reward
- Update Q-table
- Repeat the above 3 steps

In order to pick an action, the agent uses what is called an ***ϵ-Greedy*** strategy. It picks the best strategy as per the Q-table with probability $\mathbf{1 - \epsilon}$, and picks a random action with the probability $\boldsymbol{\epsilon}$. This allows the agent to maintain a balance between Exploration and Exploitation. In the beginning, the ϵ rates will be higher, the agent will explore the environment and randomly choose actions in order to learn about the environment.As the agent explores the environment, the epsilon rate decreases and the agent starts to exploit the environment.

The **Q-value update rule** is given by:

$$\boldsymbol{Q_{new}(s_t, a_t) \leftarrow}$$

$$\boldsymbol{Q(s_t, a_t) \; + \; \eta \Big[r_t \; + \; \gamma \quad max_a Q(s_{t+1}, a_{t+1}) \; - \; Q(s_t, a_t) \Big]}$$

where

- $\boldsymbol{Q(s_t, a_t)}$: Q-value for the state-action pair at time $\boldsymbol{t}$
- $\boldsymbol{\eta}$ **:** Learning rate (a number between 0 and 1)
- $\boldsymbol{r_t}$ **:** Reward observed when moving from the state $\boldsymbol{s_t}$ to the state $\boldsymbol{s_{t+1}}$ by taking action $\boldsymbol{a_t}$
- $\boldsymbol{\gamma}$ **:** Discount factor (a number between 0 and 1)
- $\boldsymbol{max_a Q(s_{t+1}, a_{t+1})}$: Target Q-function or the estimate of optimal future Q-value

The core of the algorithm is a Bellman equation as a simple value iteration update, using the weighted average of the old value and the new information. Once the Q-values converge to the actual target Q-values, the Q-Table can be used to find the action having the highest return on that state.

Q-learning has a few disadvantages: The memory and computation required to maintain the Q-table is too high, it

lacks generality as it does not have the ability to estimates values for unseen states, and it only works in environments with discrete and finite state and action spaces.

This can be overcome by applying function approximation to learn the value function, taking states as inputs, instead of storing the full state-action table. Since deep neural networks are powerful function approximators, we can combine DL and RL to create advanced Deep Reinforcement Learning (DRL) algorithms that uses Neural Networks to estimate Q-values.

4.2 Deep Q-Network (DQN):

DQN is one of the most famous DRL models which learns policies directly from high-dimensional inputs. The Q-table is replaced by a Neural Networks that tries to approximate Q Values - $\boldsymbol{Q(s, a; \theta)}$, where $\boldsymbol{\theta}$ represents the trainable weights of the network.

4.3 Multi-Agent case:

When more than one agent is involved, an MDP is no longer suitable for describing the environment, given that actions from other agents are strongly tied to the state dynamics. A generalization of MDP is given by Markov games (MGs), where the *agent index* that represent an $\boldsymbol{n_{th}}$ agent is also added to the tuple. The action and reward in a tuple will correspond to that agent.

Any fully-cooperative (ie, all agents receive identical reward signals) multi-agent learning task can broadly be divided into 3 different classes based on whether the focus is on incorporating stability of the learning dynamics of the agent, or adaptation to the changing behavior of the other agents: independent learning, centralised multi-agent policy gradient, and value decomposition. Stability essentially means the convergence to a stationary policy, whereas adaptation ensures that performance is maintained or improved as the other agents are changing their policies.

We use a decentralized independent learning model that focuses on stability (convergence). All agents are trained simultaneously, but each agent is an independent decision maker and learns independently using the same DQN architecture, while perceiving other agents as part of the environment. Each agent receives its local history of observations and updates the parameters of the Q-value network by minimising the standard Q-learning loss. The total reward is the cumulative reward obtained by each agent and the overall goal is to maximize the common discounted return.

5. Model Implementation:

Our DQN model employs two Neural Networks – a Policy Network (to predict the action or Q-values for the given input state) and Target Network (to evaluate the quality of a state, ie, retrieve the Target value for calculating the loss function for the prediction made by the Policy Network).

RL is unstable or divergent when a nonlinear function approximator such as a NN is used to represent $\boldsymbol{Q}$. This instability comes from the correlations present in the sequence of observations, the fact that small updates to $\boldsymbol{Q}$ may significantly change the policy of the agent and the data distribution, and the correlations between Q and the target values. DQN addresses these instabilities using 4 techniques:

- *Experience Replay*: An Experience tuple, which corresponds to a single data sample, is a collection of the Current State ($\boldsymbol{s_t}$), the Action Taken ($\boldsymbol{a_t}$), Rewards Received ($\boldsymbol{r_t}$) and the Next State ($\boldsymbol{s_{t+1}}$). Experience Replay stores these experiences in memory and makes random samples of mini-batches to update the NN. Random sampling provides diverse, stable and de-correlated training data, increases speed and reuse of past transitions. We also store the done value and the agent index as a part of Replay Buffer. Each action and reward in a given tuple will correspond to the agent represented by the agent index.

- *Target Network*: Unstable target function makes training difficult. The idea is to fix the Q-value targets temporarily so that we don't have a moving target to chase. The parameters of target function are replaced with the latest network every thousands or so steps.

5.1 Input to the DQN:

The DQN model takes the image of the observation space as the input. The initial layers of DQN are Convolutional layers which allows it to understand and pick features from the raw image input data. The image dimension is reduced to 84*84 and then RGB image forms an input matrix of dimension (84*84*3). We then normalize the observations by linearly scaling them between 0 and 1. All these operations are done using SuperSuit, a library that provides preprocessing functions for PettingZoo environments.

The Experience Replay, which also forms the data for DQN, are necessary to perform Q learning.

5.2 Output from the DQN:

The outputs are Q-values for all the available actions corresponding to each state. By providing the network only the state as input, we receive 6 different Q-values, one for each possible actions at that state. This allows us to calculate all the Q-values in a single pass.

5.3 Loss function:

Loss is computed using the Q-Learning's Bellman error to improve the estimation made by the network with every interaction. It is the Mean Square Error between the predicted

Q-value and the Target value, which is calculated over a batch of Environment Tuples (s_t, a_t, r_t, s_{t+1}). The variables of the loss function θ are the trainable weights of the NN that approximates the Q-values.

$$L_\theta = \frac{1}{2}\,\mathbb{E}\left[(r(s,a) + \gamma\, max_a Q(s',a';\theta)) - Q(s,a;\theta)\right]^2$$

where γ is the Discount factor determining the agent's horizon.

For more stability, we use ***Gradient Clipping***. It is useful to clip the Loss from the update (squared-error loss term) to be between -1 and 1 in order to prevent gradient blow-ups. Because the absolute value loss function $|x|$ has a derivative of -1 for all negative values of x and 1 for all positive values of x, clipping the squared error to be between -1 and 1 corresponds to using an absolute value loss function for errors outside of the (-1,1) interval. This is equivalent to replacing the squared error loss with the ***Smoothed L1 loss***.

For our DQN, we update the weights of the model in order to adjust its outputs in a way that is analogous to the Q-learning updates. We do this by optimizing the Bellman error through Backpropagation and Gradient Descent. We use Adam Optimizer with a learning rate of 10^{-3}.

5.4 DQN Network Architecture:

Our network has 7 layers:

(1) Convolutional layer with 3 input channels, 32 output channels with kernels size 8 × 8 and stride 4, followed by ReLU activation

(2) Convolutional layer with 32 input channels, 64 output channels with kernels size 4 x 4 and stride 2, followed by ReLU activation

(3) Convolutional layer with 64 input channels, 64 output channels with kernels size 3 x 3 and stride 1, followed by ReLU activation

(4) Fully Connected layer of 512 units, followed by ReLU activation

(5) Fully Connected layer of 256 units, followed by ReLU activation

(6) Fully Connected layer of 512 units, followed by ReLU activation

(7) Output Fully Connected layer that has 6 units, one for each valid action

5.5 Training:

During the training process, we use:

- Experience Replay (using Replay Buffer)
- Occasional updates to the Target Network, once every 1000 iterations
- Network update every 10 iterations
- Gradient clipping to avoid exploding gradients
- ϵ-Greedy strategy, starting with a higher ϵ value to enable more exploration during the early episodes, and slowly decaying it

The below high level DQN Algorithm summarizes the training loop that takes place:

Algorithm 1

Initialize Policy Network, having output Q
Initialize Target Network, having output Q'
Initialize Replay Buffer (it is empty in the beginning)
Import KAZ environment from the PettingZoo library and initialize Agents
Reset environment
while not converged **do**
  Select an action using ϵ-Greedy policy
  Agent at state s_t takes an action a_t, observes reward r_t and moves to the next state s_{t+1}
  Experience tuple for each agent ($agent\_index$, s_t, a_t, r_t, s_{t+1}, $done_i$) is stored in the Replay Buffer
  if enough experiences in Replay Memory **then**
    # *Begin training*
    Sample random mini-batch of experiences from Replay Buffer
    if s_{t+1} is terminal/ episode $done_i$ **then**
      Predicted value = r_i
      Reset environment
    else
      Predicted value = $r_i + \gamma\, max_a Q(s',a')$
    end if
    Compute Loss over the predicted Q value
    Update Q using SGD
    Decay ϵ
    Hard update Q' to Q for every 1000 steps
  end if
end while

Figure 1 illustrates the training loop and the overall resulting data flow.

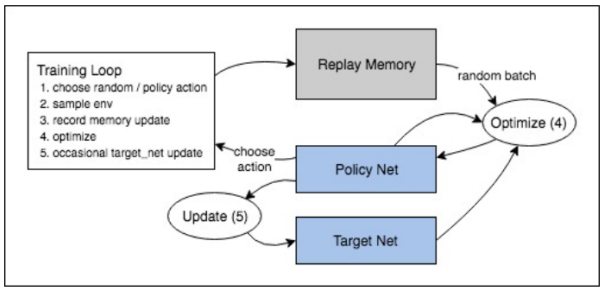

Figure 1: Data flow

Hyper-parameters used:

- Number of Agents = 4
- Replay Memory Buffer size = 100000
- Batch size for sampling = 512 (observations with other batch sizes are included under Results)
- Target Update = every 1000 iterations
- Network update = every 10 iterations
- Initial Epsilon = 1
- Epsilon decay rate = .0001
- Epsilon minimum = 0.15
- Discount factor γ = 0.99
- Adam Optimizer learning rate = $\mathbf{10^{-3}}$
- Reward scaling factor = 10

6. Results:

We attempted a few different approaches with our Decentralized Independent DQN model.

6.1 Observations and Mean Reward:

Observations from each of our approaches are mentioned below. Note that the Reward function has been scaled by a factor of 10 for representation purpose.

(1) **Training a single Archer vs a single Knight vs all 4 agents**

We initially did a dry run of our model using a single agent before diving into multi-agent scenario. A single Archer performed better than a single Knight since the range of attack for an Archer is much bigger than that of a Knight. A Knight swings a mace in an arc in front of its current heading direction, while an Archer fires an arrow in a straight line in the direction of heading.

With all 4 agents (2 Knights and 2 Archers), we observed more kills and better overall mean reward. But the total rewards gained by Archers were more than Knights due to the same reason.

(2) **Full observation space vs Partial observation space**

Since Decentralized Learning algorithm's performance is limited in partially observable environments, we attempted training our model with both partially observable environments (512, 512, 3) and full observation space (720, 1280, 3).

With full observation space, our model took a very long time to train and did not result in good mean rewards. We think that the primary reason behind this is the exponential increase in the size of the input state space. This drastically decreased the learning speed and the feature extractor was not able to cope with size of frame. Also, the is no way for an agent to identify another similar agent in the environment.

(3) **Normal Replay buffer vs Priority Replay buffer**

Priority Replay Buffer extends DQN's experience replay function by learning to replay memories where the real reward significantly diverges from the expected reward, ie, sampling transitions that have a large target gap. Thus keeping the experiences that made the neural network learn a lot and letting the agent adjust itself in response to developing incorrect assumptions.

This, however, did not outperform normal Replay Buffer and KAZ did not benefit from prioritizing experiences with higher divergence. Prioritized Replay can easily create bias or lead to overfitting the network for a subset of experiences.

Figures 2 and 3 are the Mean Rewards and Loss obtained for Independent DQN with partial observation space, Full observation space and Priority replay buffer.

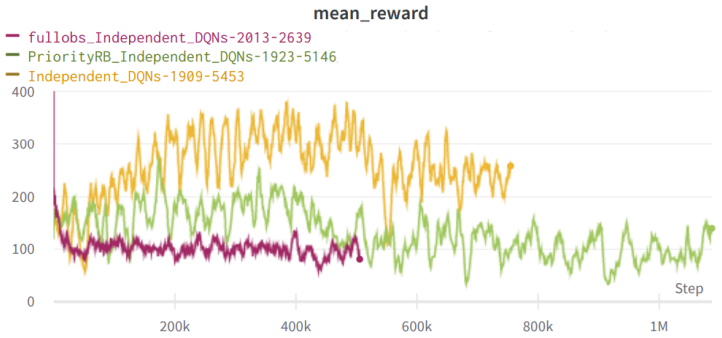

Figure 2: Mean Reward

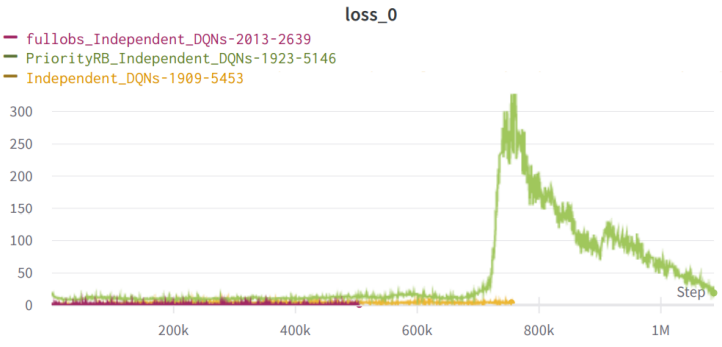

Figure 3: Loss

(4) **Batch sizes of 64, 128, 256, 512, 1024**

Smaller batch sizes of 64, 128, 256 did not provide good results. And so did very high batch size of 1024, leading to high fluctuations. We observed that the optimal batch size for our model was 512.

Figures 4 and 5 are the Mean Rewards and Loss obtained with different batch sizes.

(5) **Grey scale input image (single channel) vs RBG input image (3 channels)**

In order to reduce the computation cost, a common practice is to alter the input image frames to grey scale (single channel) before passing it to the DQN, and then frame stacking.

This, however, did not result is good training as the movement of the Zoombies were not observable in gray scale. Retaining image color resulted in better performance.

6.2 Task Visualization:

We attempted training with a single agent and with all 4 agents. A small video illustration of the evolution of the agents' learning progress for an episode can be found here: https://youtu.be/d-JKY_LGcnI

7. Conclusion:

We sought to implement a Decentralized DQN MARL model. We started by trying to understand the PettingZoo environment API and studying the tutorials provided by the PettingZoo creators. We implemented DQN on simpler single agent Gym environment and also attempted training just a single agent in KAZ environment.

During the process of training multi-agents, we attempted multiple approaches as detailed in the Results section. After multiple runs, the most optimal model has been used to train our Agents and the source code is uploaded on Github. Our Decentralized DQN MARL algorithm was able to successfully learn from the input images and perform satisfactorily in achieving human-level control over the game of KAZ from the PettingZoo environment.

8. Possible extensions:

In the original publication for Priority Replay Buffer, the authors have implemented it on top of a double Q-network algorithm. One possible extension could be to implement two dueling Q-networks instead of a DQN which might enable Priority Replay buffer's full potential.

Another possible extension could be to implement Centralised Training Decentralised Execution. This can increase the complexity of the training process, but will allow for reasoning over a larger information space. MADDPG, which is a variation of DDPG algorithm for MARL, relies on this idea. In order to achieve this, MADDPG uses two network- a policy and a critic. During training, all critics have access to information from other agents, which makes the environment stationary from the critic's perspective. This additional information from the critic is used by the actor for its training.

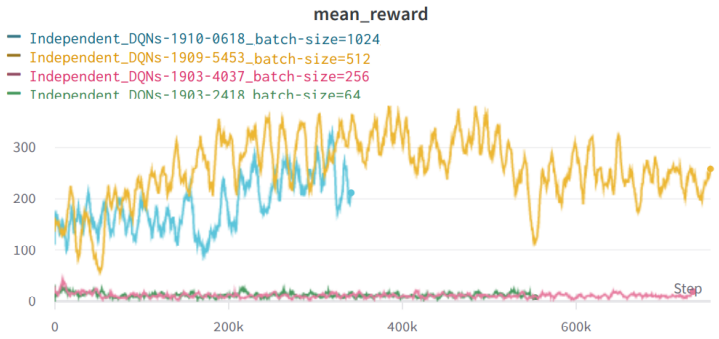

Figure 4: Mean reward for varying Batch Sizes

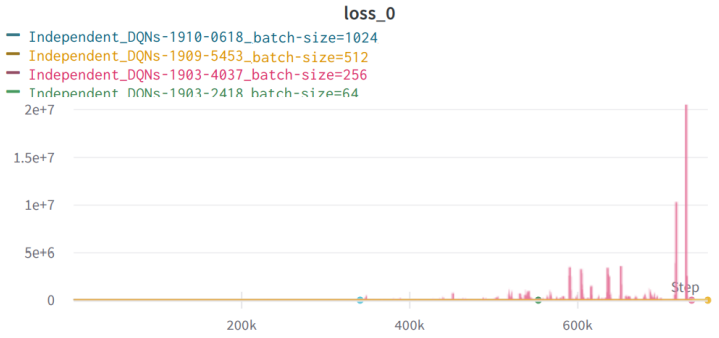

Figure 5: Loss for varying Batch Sizes

9. References:

(1) Human-level control through deep reinforcement learning https://doi.org/10.1038/nature14236

(2) Mastering the game of Go without human knowledge https://www.nature.com/articles/nature24270

(3) Reinforcement Learning for Solving the Vehicle Routing Problem https://proceedings.neurips.cc/paper/2018/file/9fb4651c05b2ed70fba5afe0b039a550-Paper.pdf

(4) Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms https://arxiv.org/abs/1911.10635

(5) Deep Reinforcement Learning for Multiagent Systems: A Review of Challenges, Solutions, and Applications https://ieeexplore.ieee.org/document/9043893

(6) Multi-Agent Reinforcement Learning Independent vs Cooperative Agents https://web.media.mit.edu/~cynthiab/Readings/tan-MAS-reinfLearn.pdf

(7) Multiagent Cooperation and Competition with Deep Reinforcement Learning https://arxiv.org/abs/1511.08779

(8) Stabilising Experience Replay for Deep Multi-Agent Reinforcement Learning https://arxiv.org/abs/1702.08887

(9) Deep Multi-Agent Reinforcement Learning using DNN-Weight Evolution to Optimize Supply Chain Performance https://www.researchgate.net/publication/322677430_Deep_Multi-Agent_Reinforcement_Learning_using_DNN-Weight_Evolution_to_Optimize_Supply_Chain_Performance

(10) Deep Reinforcement Learning with Double Q-learning https://arxiv.org/abs/1509.06461

(11) Dueling Network Architectures for Deep Reinforcement Learning https://arxiv.org/abs/1511.06581

(12) Deep Recurrent Q-Learning for Partially Observable MDPs https://arxiv.org/abs/1507.06527

(13) Multi-Agent Reinforcement Learning: A Review of Challenges and Applications https://www.mdpi.com/2076-3417/11/11/4948/htm

(14) A REVIEW OF COOPERATIVE MULTI-AGENT DEEP REINFORCEMENT LEARNING https://arxiv.org/pdf/1908.03963.pdf

(15) PettingZoo official Github repo https://github.com/Farama-Foundation/PettingZoo

(16) PettingZoo - Multi-agent Reinforcement Learning Environments https://www.youtube.com/watch?v=IMpf_X1IN_0

(17) REINFORCEMENT LEARNING (DQN) TUTORIAL https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

(18) Message-Dropout: An Efficient Training Method for Multi-Agent Deep Reinforcement Learning https://arxiv.org/abs/1902.06527

(19) Multi-Agent Reinforcement Learning: The Gist https://medium.com/swlh/the-gist-multi-agent-reinforcement-learning-767b367b395f

(20) RL — DQN Deep Q-network https://jonathan-hui.medium.com/rl-dqn-deep-q-network-e207751f7ae4

(21) Prioritized Experience Replay https://arxiv.org/abs/1511.05952

(22) Multi-agent reinforcement learning: An overview https://www.dcsc.tudelft.nl/~bdeschutter/pub/rep/10_003.pdf

(23) Rainbow is all you need https://github.com/Curt-Park/rainbow-is-all-you-need

(24) Why Going from Implementing Q-learning to Deep Q-learning Can Be Difficult https://towardsdatascience.com/why-going-from-implementing-q-learning-to-deep-q-learning-can-b

(25) Welcome to Deep Reinforcement Learning Part 1 : DQN https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b

(26) Using PettingZoo with RLlib for Multi-Agent Deep Reinforcement Learning https://towardsdatascience.com/using-pettingzoo-with-rllib-for-multi-agent-deep-reinforcement-lear

(27) Crash Course: Reinforcement Learning 101 Deep Q Networks in 10 Minutes https://towardsdatascience.com/qrash-course-deep-q-networks-from-the-ground-up-1bbda41d3677