Generalizing Multi-Agent Reinforcement Learning: Advancing RPM

Samuel Kleiner and Zachary Vachal

Professors: Mengdi Wang and Benjamin Eysenbach

Submitted in partial fulfillment

of the requirements of the final project

for ECE433: Introduction to Reinforcement Learning

Princeton University

May 2024

Contents

1 Introduction

After realizing that reproducing Multi-Agent PPO with Ranked Policy Memory (RPM) was intractable in the time we had, we decided to use our learned experience with Multi-Agent Reinforcement Learning (MARL) codebases and simulators on another paper that advances MARL through experience sharing. "Selectively Sharing Experiences Improves Multi-Agent Reinforcement Learning" [3] addresses the fundamental need for more data efficient methods in MARL that provide higher performance. Its algorithm is creatively named Selective Multi-Agent Prioritized Experience Relay (SUPER), and it builds off previous variants of DQN.

Multi-agent problems are at the cutting edge of the reinforcement learning space both in terms of algorithms and computational structures. This paper addressed both of these areas of issue. Experience sharing is generally a process of implementing a connection between the experience buffers of separately acting agents in off-policy methods of multi-agent reinforcement learning. This has the advantage of data-efficiency, a common problem in environments which are not easily simulated or are expensive or difficult to sample from. Additionally, we see (and will discuss in this reproduction) that we can also get performance improvements. Intuitively, sharing important or rare experiences between agent, especially in collaborative problems can allow agents to learn more quickly. There are the two main areas SUPER aims to leverage.

1.1 Problem Statement

In MARL, the total rewards from a multi-agent policy depend on a complex entanglement of all agents' actions and interactions in environments that often have complicated dynamics. The exponentially increased complexity of MARL with respect to single agent reinforcement learning means that achieving a reasonable level of data efficiency to allow for practical applications of MARL to be tractable is an even harder problem to solve. "Selectively Sharing

Experiences Improves Multi-Agent Reinforcement Learning" [3] uses the fact that having multiple agents allows for rapid exploration of an environment to turn the complexity induced by multiple agents into performance boost with experience sharing. Their paper seeks to address the need for data efficiency in MARL, and succeeds achieving a significant performance boost by having agents share relevant experiences, which they determine through three heuristics that they test.

Formally, the problem of MARL is formed as a stochastic Markov Game defined by tuples $\langle S, A, R, T, \gamma \rangle$ as follows:

- S: Set of states.
- $A = \{A_i\}_{i=1}^{n}$: A collection of action sets for each agent A_i
- $R = \{R_i\}_{i=1}^{n}$: A collection of reward functions R_i, where each R_i is defined as the reward $r_i(a_t, s_t)$ that agent i receives when the joint action $a_t \in A$ is performed at state s_t.
- T: The probability distribution over possible transitions
- γ: Discount factor.

The specific frame of the learnable problem is a partially observed Markov game where each agent has a set of observations defined as experience tuples $e = \langle S_t, A_t, R_{t+1}, S_{t+1} \rangle$. The approach taken by this paper uses a "decentralized execution setting" whereby each agent trains an individual policy π_i and optimizes for its specific reward function R_i. The proposed algorithm demands that the environment be "anonymous" meaning each actor interacts identically with the environment and vice versa: the same observation and action spaces as well as the same environment responses. This is to the shared experiences are also valid tuples for agents which did not collect them. Although this may seem to be a narrow scope, consider that many common settings are defined similarly. For example, most cooperative settings are constructed as anonymous markov games.

The other important concepts are Temporal Difference Error, a common metric for loss in deep Q-learning, and prioritized replay buffers which allow for optimization on state transitions where we encounter the highest TD error. Both of these concepts are integrated into the algorithm used as a base for this paper, DDQN, or Double Deep Q-Network. This algorithm is a standard off-policy formulation which optimizes to reduce TD error gives batches of gather transition tuples. This paper aims to contribute three heuristics for just how transition tuples are chosen for training the Q-network.

1.2 Problem Importance

Improving fundamental multi-agent reinforcement learning methods to achieve higher performance and and increased data efficiency is important because the world is full of environments that require multiple agents to cooperate or compete for society to function. Resource management systems are everywhere in infrastructure and industry and can be aptly modeled as a multi-agent system [10]. Looking to the future, multi-agent systems have exciting applications in enumerable fields and industries. The field of autonomous vehicles requires developing policies for the complex multi-agent interactions of the road for deployment to ever occur. The environment, economy, and games of strategy can all be modeled as systems of multiple agents cooperating and competing to receive limited available rewards. These environments are inherently complex due to the interactions between multiple agents whose behaviors can affect one another. Allowing multi-agent policies to be learned faster with greater data efficiency while boosting those policies' performances will result in greater rewards for the people who deploy and interact with multi-agent systems.

1.3 Motivation

SUPER-DDQN advances the capability of MARL by increasing data efficiency while improving performance. Making significant progress on the inherent problem of data inefficiency in

MARL training would allow MARL to provide solutions to many systems that can be modeled as having multiple agents that cooperate or compete with each other for limited rewards. Deploying these systems in the real-world requires high performance, especially in safety-critical systems, like autonomous vehicles. It also requires fast learning times that must be significantly reduced through data efficiency since systems need to be able to retrain on new data in order to adapt to new environments, and solve problems, like the sim-to-real gap.

1.4 Problem Difficulty

While there are a variety of MARL methods that exist with most single agent reinforcement learning methods have multi-agent variants, MARL is a rapidly evolving field as researchers search for ways to make multi-agent policies learn faster and perform better in complicated environments. The most naive approach to MARL in comparison to single-agent reinforcement learning would be to allow each agent to learn its own policy and reduce the problem to an isolated single-agent reinforcement learning problem for each agent. The naive approach would not take into account the fact that each agent's experience in a multi-agent setting is affected by the actions of other agents. It would also expose another reason that MARL problems are more difficult to solve than the inherent difficulties of single-agent reinforcement learning. Every agent's action changes the environment of all other agents. Even with a stationary environment where environment dynamics do not change over time, in a multi-agent setting non-stationary will always be present if more than one agent is acting. The performance of learned policies naturally fluctuates in response to complex environments, but even more so in non-stationary complex environments. The naive approach and other basic approaches from single-agent reinforcement learning will fail to achieve adequate performance in MARL problems because they do not account for the interconnected nature of agents' actions, and they do not take advantage of the greater capability of exploration in multi-agent systems.

1.5 SUPER-DDQN's Innovation

SUPER differs from previous MARL methods by making the complexity introduced by having numerous agents in a system a critical asset in exploration and learning. One of the fundamental difficulties in RL is that you cannot just pick and choose a state for an agent to be in when modeling most real-world environments. An agent in a distant corner of a vast, sparse environment that could make a significant leap in learning by having an experience in a different distant corner of the environment would have to step along a far trajectory to achieve that learning benefit even if an exploration strategy some how took it straight there. By sharing relevant experiences, SUPER allows agents to learn from experiences throughout the environment without increasing the computation load of each agents experiences significantly.

2 Related Work

Multi-agent reinforcement learning exists on a spectrum between centralized MARL with one policy controlling all agents, and decentralized MARL where agents operate independently. This spectrum is characterized by varying levels of agent awareness regarding each other's actions and states. Focusing on a semi-decentralized approach, SUPER enables agents to share experiences, so that it builds off advancements in cooperative communication protocols, modeling of other agents' behavior, and the heuristics developed by Prioritized Experience Replay (PER) for ranking the relevancy of experiences. SUPER leverages the versatility of Deep Q-Network (DQN) variants. Since SUPER can be integrated with any DQN variant, it is highly generalizabile. Furthermore, the work extends the strategic paradigm of centralized training with decentralized execution, emphasizing the modeling of other agents' behaviors

within this framework, which is critical for effective agent cooperation and performance optimization in MARL problems.

2.1 Cooperative Communication Protocols

Leveraging inter-agent communication to accelerate learning and improve performance in dynamic environments is the central theme of the SUPER algorithm. The SUPER algorithm builds off work in the subject of cooperative communication protocols since only the most relevant experiences being shared allows agents communicate with limited bandwidth. One paper, which demonstrates the powerful capabilities of cooperative communication, is "Multi-Agent Cooperation and the Emergence of (Natural) Language" [4]. The authors proposed a multi-agent communication framework within the context of referential games, where a sender and a receiver collaborate to identify a target image based on a limited, fixed vocabulary. With this communication protocol and vocabulary, the agents learned to coordinate and evolved a "natural" language to increase their performance in identification. The SUPER algorithm was primarily tested on the PettingZoo environments, Pursuit and its variants Adversarial Pursuit and Battle. These environments require cooperation between agents for any significant rewards to be received (from cornering or surrounding the evader agents). Likewise the image identification game required the sharing of information between cooperative agents to allow for success to be achieved.

2.2 Discussion of PER Heuristics and TD Error

The heuristics developed by Prioritized Experience Replay (PER) are used by SUPER as its ranking method for experiences, but SUPER could be used with different heuristics and does not depend on PER. PER focuses on the magnitude of the TD error to rank the relevancy of experiences [7]. This methodology allows for the selective sharing of critical experiences

among agents, enhancing the learning speed and overall performance in multi-agent environments. By prioritizing experiences where the prediction error is highest, SUPER ensures that agents learn from the most significant discrepancies in their understanding of the environment, leading to more reliable and efficient learning.

2.3 DQN and Its Variants

DQN's have the ability to achieve human-level performance in complex, dynamic environments like Atari games [5]. SUPER was implemented on DDQN in [3] since it is a standard and simple improvement on DQN, but the SUPER algorithm can be implemented on top of any standard DQN algorithms, demonstrating its flexibility and making it widely applicable. This flexibility is pivotal for adapting DQN's single-agent learning capabilities to the multi-agent context, where training can take significantly longer than in single-agent RL. SUPER's sharing of high-impact experiences can significantly reduce the time required to converge on effective strategies when compared to the basic DQN variants, which have the advantage of being tried and tested on many RL problems.

2.4 Shared Experience Actor Critic

The closest related work to [3] is the Shared Experience Actor Critic approach [2], which is similar to SUPER, yet without any ranking of experiences. In Shared Experience Actor Critic, every experience from every is shared to a joint replay buffer. The authors of the SUPER paper showed that sharing every experience instead of only sharing the most relevant experiences slows down learning and reduces performance.

3 SUPER's Method

Implemented on standard DQN, the SUPER method was outlined succinctly by the paper's authors in Figure 1.

1. (DQN) Collect a rollout of experiences, and insert each agent's experiences into their own replay buffer.
2. (SUPER) Each agent shares their most relevant experiences, which are inserted into all the other agents' replay buffers.
3. (DQN) Each agent samples a minibatch of experiences from their own replay buffer, and performs gradient descent on it.

Figure 1: The SUPER method inserted into standard DQN written by the paper's authors.

We chose to implement the SUPER method on top of DDQN since that was the DQN variant chosen by the paper's authors. SUPER-DDQN has every agent share their most relevant experiences as determined by the chosen selection heuristic for all other agents to insert into their replay buffer.

3.1 Selection Heuristics

The three experience selection heuristics outlined in the paper are explained in this section.

3.2 Deterministic Quantile experience selection

$$|\text{td}(e_t)| \geq \text{quantile}_(\{e_{t'}\}_{t'=t-k}^{t})$$

Using this heuristic, SUPER-DDQN maintains a list of the absolute TD-errors from its last k experiences. When a new experience is processed, its absolute TD-error is compared to the TD-error at the β-quantile of the list. If the new experience's TD-error is at least as large, it is considered significant enough to be shared.

3.3 Deterministic Gaussian experience selection

$$|\text{td}(e_t)| \geq \mu + c \cdot \sigma$$

This heurisitc calculates the mean, μ, and variance, σ^2, of the absolute TD-errors from the last k experiences. An experience e_t is shared if its TD-error exceeds $\mu + c \times \sigma$. C is a constant such that the upper tail of the normal distribution beyond c equals . This cutoff targets sensitivity to outlier clusters by potentially capturing entire clusters rather than fragments. This selection heuristic reduces memory usage. Mean and variance are computed iteratively, so all recent TD-errors do not need to be stored.

3.4 Stochastic weighted experience selection

$$p = \min\left(1, \beta \cdot \frac{\sum_k p_i^\alpha}{k}\right)$$

This heuristic shares each experience with a probability proportional to its absolute TD-error. A certain fraction of experiences are sampled from a sliding window of recent experiences, weighted by their TD-errors. This sampling aims to approximate sampling without replacement in expectation [3].

4 SUPER's Limitations

The limitations to SUPER's method is that it is only tested on standard DQN variants for MARL problems, yet the idea of sharing relevant experience though cooperative communication to improve learning could be extended to other families of algorithms. There are also only three heuristics for selecting experience to share that were tested. Other heuristics could be conceived of and tested, perhaps such that they incorporate skill building or planning in some way. This said, it is also not necessary to show many heuristics to prove that experience sharing is a widely applicable and simple leap forward for MARL problems with cooperative agents where communication is possible. The one significant limitation

that exists with SUPER is its dependence on accurate TD error estimation. If the TD error calculation is flawed or biased, it might lead to sub optimal performance or slower learning because non-informative experiences could be shared erroneously. Yet, other heuristics could be designed to avoid this. Furthermore, SUPER was shown to be highly effective by its authors while using heuristics that depend on TD Error estimation.

5 Experimental Procedure

To reproduce the paper, we used the algorithm shown in Figure 2 as our starting point.

Algorithm 1 SUPER algorithm for DQN

for each training iteration **do**
  Collect a batch of experiences b {DQN}
  for each agent i **do**
    Insert b_i into buffer$_i$ {DQN}
  end for
  for each agent i **do**
    Select $b_i^* \subseteq b_i$ of experiences to share[1] {SUPER}
    for each agent $j \neq i$ **do**
      Insert $b*_i$ into buffer$_j$ {SUPER}
    end for
  end for
  for each agent i **do**
    Sample a train batch b_i from buffer$_i$ {DQN}
    Learn on train batch b_i {DQN}
  end for
end for

[1] See section "Experience Selection"

Figure 2: The SUPER-DQN Algorithm written by the paper's authors.

We built off the implementation of DDQN from class. We decided to focus on the Pursuit simulator from PettingZoo because it is the simplest game to learn out of the PettingZoo simulators the authors tested SUPER-DDQN on. We used PettingZoo's documentation extensively to code our implementation [8]. The most helpful page was the "Basic Usage" page, and the documentation of the Pursuit environment. We used google colab to implement SUPER-DDQN since it provided the most accessible platform to run on.

5.1 Challenges

5.1.1 Attempt to Implement RPM

We originally attempted to reproduce a paper on MARL with Ranked Policy Memory. When reading through the repository for our original paper, it seemed that there was a fair amount of code not critical to the main algorithm. It appeared that for our reproduction, understanding the code base and written algorithms enough to implement RPM ourselves would be possible, especially considering that RPM was built on top of on the work of Yu et al., 2021 [9], the algorithm for MAPPO, a more widely known work. Through enumerable hours of work understanding the git repository "BenchMARL" [1], implementing our own algorithm, and getting it to run on Princeton's GPUs, no results were produced by the program. We attempted to solve this in a number of ways before realizing that it was intractable in the time that we had.

Yet, we gained an intimate understanding with simulators like Deepmind's Meltingpot, PettingZoo, and the VMAS simulator. Trying to implement MAPPO on which RPM is based on various code bases as well as extending our own implementation of PPO, gave us a profound understanding and appreciation for the difficulty of solving MARL problems. When we realized we would have to attempt to edit the source code of torchRL to implement RPM on top of MAPPO, we decided it would be better to pivot to a similar paper that we could implement using our new understanding of MARL variants and simulators with the time and resources we had.

5.1.2 Implementation of SUPER-DDQN

We still faced major challenges while implementing SUPER-DDQN and training. MARL algorithms even torchRL, a well documented code base, required many hours of debugging for the program to run. We had difficulties with CUDA storage, and each epoch took around

fifty seconds to run.

6 Results

Due to issues of computation, we were unable to produce reasonable results for this reproduction. This is a classic issue of MARL and one of the reasons it is such a difficult space to work in or reproduce papers for. Although we were unaware of this issue initially, we have become quite aware of the difficulties of dealing with a space that has inconsistently updated code libraries and widely varying system requirements. As the field matures and computation improves to meet the incredibly demanding problem formulation, reproducability will certainly come, but for us to attempt to reproduce this paper took countless hours of coding to get a number of libraries to coordinate with our freshly adapted DDQN and only then did we encounter the computational requirements which made this quite an impossible paper to reproduce with the resources available. That considered, we have learned much about the general problem formulation and the process of reproducing papers which will be useful in the future.

7 Appendix

7.1 Our Code for Implementing SUPER-DDQN

Setup

Make sure to run every single cell in this notebook, or some libraries might be missing. Also, if you are using Colab, make sure to **change your Runtime (change runtime type under Runtime)** to a GPU.

Install the necessary libraries for rendering.

In [ ]:
```
# %pdb
```

In [ ]:
```
!apt-get install x11-utils > /dev/null 2>&1
!pip install pyglet > /dev/null 2>&1
!apt-get install -y xvfb python-opengl > /dev/null 2>&1

!pip install gym pyvirtualdisplay > /dev/null 2>&1
```

In [ ]:
```
!pip install pettingzoo
!pip install 'pettingzoo[sisl]'
!pip install supersuit
```

Part 1: Implementing DQN

In this part, you will be filling out the code for a basic DQN model. Recall that for a DQN, we are approximating the Q-value table in Q-learning with a neural network.

Fill in all sections labelled # FILL ME IN

In [ ]:
```
import os
import pdb
import sys
import copy
import json
import argparse
from datetime import datetime

import random
from collections import deque
from supersuit import color_reduction_v0, frame_stack_v1, resize_v1
import pettingzoo
from pettingzoo.sisl import pursuit_v4
import gym
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
from IPython import display as ipythondisplay

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


class DQN(nn.Module):
    def __init__(self, input, hidden, output):
        super().__init__()
        self.network = nn.Sequential(
            self._layer_init(nn.Conv2d(3, 32, 3, padding=1)),
            nn.MaxPool2d(2),
            nn.ReLU(),
            self._layer_init(nn.Conv2d(32, 32, 3, padding=1)),
            nn.MaxPool2d(2),
            nn.ReLU(),
            nn.Flatten(),
            self._layer_init(nn.Linear(8192, 32)),
            nn.ReLU(),
        )
        self.actor = self._layer_init(nn.Linear(32, output), std=0.01)

    def _layer_init(self, layer, std=np.sqrt(2), bias_const=0.0):
        torch.nn.init.orthogonal_(layer.weight, std)
        torch.nn.init.constant_(layer.bias, bias_const)
        return layer
```

```
    def forward(self, x):
        q_vals = self.actor(self.network(x))
        return q_vals

class QNetwork():
    # This class essentially defines the network architecture.
    # It is NOT the PyTorch Q-network model (nn.Module), but a wrapper
    # The network should take in state of the world as an input,
    # and output Q values of the actions available to the agent as the output.

    def __init__(self, args, input, output, learning_rate):
        # Define your network architecture here. It is also a good idea to define any training operations
        # and optimizers here, initialize your variables, or alternately compile your model here.
        self.weights_path = 'models/%s/%s' % (args['env'], datetime.now().strftime("%Y-%m-%d_%H-%M-%S"))

        # Network architecture.
        self.hidden = 128
        self.model = DQN(input, self.hidden, output).to(device)

        # Loss and optimizer.
        self.optim = torch.optim.Adam(self.model.parameters(), lr=learning_rate)

        if args['model_file'] is not None:
            print('Loading pretrained model from', args['model_file'])
            self.load_model_weights(args['model_file'])

    def save_model_weights(self, step):
        # Helper function to save your model / weights.
        if not os.path.exists(self.weights_path): os.makedirs(self.weights_path)
        torch.save(self.model.state_dict(), os.path.join(self.weights_path, 'model_%d.h5' % step))

    def load_model_weights(self, weight_file):
        # Helper function to load model weights.
        self.model.load_state_dict(torch.load(weight_file))
```

Part 2: Replay Memory

Replay memory or Experience replay is a simple trick used to learn the Q-value network offline. It also ensures the model can learn from past experiences without weighting heavily towards current observations.

In [ ]:
```
import random
from collections import deque
import torch
import math
from scipy.stats import norm

class Replay_Memory():
    def __init__(self, state_dim, action_dim, memory_size=500, burn_in=300, k=300, beta=0.1):
        self.memory_size = memory_size
        self.burn_in = burn_in
        self.buffer = deque(maxlen=memory_size)
        self.burned_in = False
        self.k = k
        self.beta = beta
        self.sum_td = 0
        self.sum_td_squared = 0
        self.td_errors = deque(maxlen=k)

    def append(self, states, actions, rewards, next_states, dones, td_error):
        if len(self.td_errors) == self.td_errors.maxlen:
          oldest_td_error = self.td_errors.popleft()
        else:
          oldest_td_error = 0

        # Here we append to the replay buffer including our absolute td error
        # For each transition. This we utilize to later sample significant transitions
        # based on each of the three SUPER heuristics.
        self.buffer.append((states, actions, rewards, next_states, dones, abs(td_error)))

        # We continually update our statistics used for two of the
        # three SUPER heuristics for sharing
        self.update_mem_statistics(abs(td_error), oldest_td_error)
        if len(self.buffer) > self.burn_in:
            self.burned_in = True
```

```
    def get_significant_experiences_quantile(self):
        if len(self.buffer) < self.k:
            return []

        # implementing the simple quantile heuristic for experience sharing
        sorted_errors = sorted((e[5], idx) for idx, e in enumerate(self.buffer))
        threshold_index = int(len(sorted_errors) * self.beta) - 1
        threshold = sorted_errors[threshold_index][0]

        significant_experiences = [self.buffer[idx] for td, idx in sorted_errors if td >= threshold]


        return [(s, a, r, ns, d) for (s, a, r, ns, d, _) in significant_experiences]

    def update_mem_statistics(self, abs_td_error, oldest_td_error):
        # Here we maintain sums of td error for the gaussian and probability
        # sharing heuristics
        self.sum_td -= oldest_td_error
        self.sum_td_squared -= oldest_td_error ** 2

        self.td_errors.append(abs_td_error)
        self.sum_td += abs_td_error
        self.sum_td_squared += abs_td_error ** 2

    def get_significant_experiences_normal(self):
        if len(self.td_errors) < self.k:
            return []

        # implementation of the gaussian heuristic for experience sharing
        #  using parameter beta.
        mean = self.sum_td / len(self.td_errors)
        variance = self.sum_td_squared / len(self.td_errors) - mean ** 2
        std_dev = math.sqrt(variance)
        threshold = mean + norm.ppf(1 - self.beta) * std_dev

        return [exp for exp in self.buffer if abs(exp[5]) >= threshold]

    def get_significant_experiences_prob(self):
        if len(self.td_errors) < self.k:
            return []

        selected_experiences = []
        total_td_error = self.sum_td_errors

        # implementation of the stochastic heuristic for experience sharing
        for exp in self.buffer:
            td_error = exp[5]
            probability = min(1, (self.beta * td_error) / total_td_error)

            if random.random() < probability:
                selected_experiences.append(exp[:-1])

        return selected_experiences

    def sample_batch(self, batch_size=32):
        sample = random.sample(self.buffer, batch_size)
        batch = tuple(zip(*sample))

        states = torch.stack([s.clone() for s in batch[0]]).requires_grad_().to(device)
        actions = torch.tensor(batch[1], dtype=torch.long).to(device)
        rewards = torch.tensor(batch[2], dtype=torch.float32).requires_grad_().to(device)
        next_states = torch.stack([s.clone() for s in batch[3]]).requires_grad_().to(device)
        dones = torch.tensor(batch[4], dtype=torch.float32).requires_grad_().to(device)

        return states, actions, rewards, next_states, dones
```

Part 3: The agent class

This section is focused on building the agent that interacts with the environment. The agent wrapper defines a policy (which you will implement), as well as all of the functionality for learning the network and using experience replay. You will implement a large chunk of this class.

In [ ]:
```
def batchify_obs(obs, device):
  """Converts PZ style observations to batch of torch arrays."""
  # convert to list of np arrays
```

```
  obs = np.stack([obs[a] for a in obs], axis=0)
  # transpose to be (batch, channel, height, width)
  obs = obs.transpose(0, -1, 1, 2)
  # convert to torch
  obs = torch.tensor(obs).to(device)

  return obs

def unbatchify(x, env):
  """Converts np array to PZ style arguments."""
  x = x.cpu().numpy()
  x = {a: x[i] for i, a in enumerate(env.possible_agents)}

  return x
```

In [ ]:

```
class DQN_Agent():
    # In this class, we will implement functions to do the following.
    # (1) Create an instance of the Q Network class.
    # (2) Create a function that constructs a policy from the Q values predicted by the Q Network.
    #       (a) Epsilon Greedy Policy.
    #       (b) Greedy Policy.
    # (3) Create a function to train the Q Network, by interacting with the environment.
    # (4) Create a function to test the Q Network's performance on the environment.
    # (5) Create a function for Experience Replay.

    def __init__(self, args):
        # Create an instance of the network itself, as well as the memory.
        # Here is also a good place to set environmental parameters,
        # as well as training parameters - number of episodes / iterations, etc.

        # Inputs
        self.args = args
        self.environment_name = self.args['env']
        self.render = self.args['render']
        self.epsilon = args['epsilon']
        self.network_update_freq = args['network_update_freq']
        self.log_freq = args['log_freq']
        self.test_freq = args['test_freq']
        self.save_freq = args['save_freq']
        self.learning_rate = args['learning_rate']
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.exp_selection = args['exp_selection']
        self.beta = args['beta']
        self.k = args['k']
        self.share_freq = args['share_freq']
        self.batch_size = args['batch_size']


        # Env related variables
        if self.environment_name == 'CartPole-v0':
            self.env = gym.make(self.environment_name, render_mode='rgb_array')
            self.discount_factor = 0.99
            self.num_episodes = 200
        elif self.environment_name == 'MountainCar-v0':
            self.env = gym.make(self.environment_name, render_mode='rgb_array')
            self.discount_factor = 0.999
            self.num_episodes = 10000
        elif self.environment_name == 'pursuit':
            self.frame_size = (64, 64)
            self.stack_size = 3
            env = pursuit_v4.parallel_env(render_mode='rgb_array', n_pursuers=2)
            env = color_reduction_v0(env)
            env = resize_v1(env, self.frame_size[0], self.frame_size[1])
            self.env = frame_stack_v1(env, stack_size=self.stack_size)
            self.discount_factor = 0.99
            self.num_episodes = 2000
            self.num_agents = len(self.env.possible_agents)
            self.agent_z = self.env.possible_agents[0]
        else:
            raise Exception("Unknown Environment")

        # Other Classes
        print(self.env.observation_space(self.agent_z).shape, self.env.action_space(self.agent_z).n, self.learni
        self.q_nets = dict()
        self.target_q_nets = dict()
        self.memories = dict()
        self.obs_size = self.env.observation_space(self.agent_z).shape[0]
        # self.action_size = self.env.action_space(self.agent_z).n
```

```
        for agent in self.env.possible_agents:
          self.q_nets[agent] = QNetwork(args, self.obs_size, self.env.action_space(agent).n, self.learning_rate)
          self.target_q_nets[agent] = QNetwork(args, self.obs_size, self.env.action_space(agent).n, self.learnir
          self.memories[agent] = Replay_Memory(self.obs_size, self.env.action_space(agent).n, memory_size=args[

        # Plotting
        self.rewards = []
        self.td_error = []
        self.batch = list(range(32))

        # Save hyperparameters
        self.logdir = 'logs/%s/%s' % (self.environment_name, datetime.now().strftime("%Y-%m-%d_%H-%M-%S"))
        if not os.path.exists(self.logdir):
            os.makedirs(self.logdir)
        with open(self.logdir + '/hyperparameters.json', 'w') as outfile:
            json.dump((self.args), outfile, indent=4)

    def epsilon_greedy_policy(self, q_values, epsilon, agent):
        # Creating epsilon greedy probabilities to sample from.
        p = np.random.uniform(0, 1)
        if p < epsilon:
            return self.env.action_space(agent).sample()
        else:
            return torch.argmax(q_values).item()

    def greedy_policy(self, q_values):
        return torch.argmax(q_values).item()

    def train(self):
        # In this function, we will train our network.
        # If training without experience replay_memory, then you will interact with the environment
        # in this function, while also updating your network parameters.

        # When use replay memory, you should interact with environment here, and store these
        # transitions to memory, while also updating your model.
        self.burn_in_memory()
        for step in range(self.num_episodes):
            # print("episode", step)
            # Generate Episodes using Epsilon Greedy Policy and train the Q network.
            self.generate_episode(policy=self.epsilon_greedy_policy, mode='train',
                epsilon=self.epsilon, frameskip=self.args['frameskip'])

            # Test the network.
            if step % self.test_freq == 0:
                test_reward, test_error = self.test(episodes=5)
                self.rewards.append([test_reward, step])
                self.td_error.append([test_error, step])

            # Update the target network.
            if step % self.network_update_freq == 0:
                self.hard_update()

            # Logging.
            if step % self.log_freq == 0:
                print("Step: {0:05d}/{1:05d}".format(step, self.num_episodes))

            # # Save the model.
            # if step % self.save_freq == 0:
            #   for agent in self.env.agents:
            #     self.q_nets[agent].save_model_weights(step)

            # Share experiences dependent on chosen heuristic.
            if step % self.share_freq == 0:
              # print("Sharing experiences")
              sharable_experiences = []
              for agent in self.env.agents:
                  if self.exp_selection == 'gaussian':
                    sharable_experiences.append(self.memories[agent].get_significant_experiences_normal())
                  elif self.exp_selection == 'quantile':
                    sharable_experiences.append(self.memories[agent].get_significant_experiences_quantile())
                  elif self.exp_selection == 'probability':
                    sharable_experiences.append(self.memories[agent].get_significant_experiences_probability())
                  else:
                    raise Exception("Unknown Experience Selection")

              for agent in self.env.agents:
                  for experience in sharable_experiences:
                    self.memories[agent].append(experience)
```

```
            step += 1
            self.epsilon_decay()

            # Render and save the video with the model.
            if step % int(self.num_episodes / 3) == 0 and self.args['render']:
                # test_video(self, self.environment_name, step)
                self.q_network.save_model_weights(step)

    def td_estimate (self, state, action, agent):
        q_values = self.q_nets[agent].model.forward(state)
        # print(q_values.shape)
        # print(action.shape)
        q_values = q_values.gather(1, action.long().unsqueeze(1)).squeeze()
        # print('est q_vals', q_values.shape)
        return q_values

    def td_target (self, reward, next_state, done, agent):
        with torch.no_grad():
            target_next_q = self.target_q_nets[agent].model.forward(next_state)
            best_action = torch.argmax(target_next_q, axis=1)
            q_values = target_next_q.gather(1, best_action.unsqueeze(1)).squeeze()
            # print('targ q_vals', q_values.shape)
            return reward.squeeze() + self.discount_factor * (1 - done.squeeze()) * q_values

    def train_dqn(self, agent):
        # Sample from the replay buffer.
        state, action, rewards, next_state, done = self.memories[agent].sample_batch(batch_size=self.batch_size)

        # Network Input - S | Output - Q(S,A) | Error - |Y - Q(S,A)|
        # compute td targets and estimate for loss
        # print(state.shape, action.shape, rewards.shape, next_state.shape, done.shape)
        td_estimate = self.td_estimate(state, action, agent)
        td_target = self.td_target(rewards, next_state, done, agent)
        # print('est and targ', td_estimate.shape, td_target.shape)

        if (td_target.shape != td_estimate.shape):
            pass

        # compute loss and backpropogate
        loss = F.smooth_l1_loss(td_estimate, td_target)
        loss.backward()
        self.q_nets[agent].optim.step()
        self.q_nets[agent].optim.zero_grad()

        return loss

    def hard_update(self):
        for agent in self.env.agents:
          self.target_q_nets[agent].model.load_state_dict(self.q_nets[agent].model.state_dict())

    def test(self, model_file=None, episodes=2):
        # Evaluate the performance of your agent over 100 episodes, by calculating cumulative rewards for the 10
        # Here you need to interact with the environment, irrespective of whether you are using a memory.
        cum_reward = []
        td_error = []
        for count in range(episodes):
            reward, error = self.generate_episode(policy=self.epsilon_greedy_policy,
                mode='test', epsilon=0.05, frameskip=self.args['frameskip'])
            cum_reward.append(reward)
            td_error.append(error)
        cum_reward = torch.tensor(cum_reward)
        td_error = torch.tensor(td_error)

        # print(cum_reward, td_error)
        print("\nTest Rewards: {0} | TD Error: {1:.4f}\n".format(torch.mean(cum_reward), torch.mean(td_error)))
        return torch.mean(cum_reward), torch.mean(td_error)

    def burn_in_memory(self):
        # Initialize your replay memory with a burn_in number of episodes / transitions.
        while not self.memories[self.agent_z].burned_in:
            print("still burning", len(self.memories[self.agent_z].buffer))
            self.generate_episode(policy=self.epsilon_greedy_policy, mode='burn_in',
                epsilon=self.epsilon, frameskip=self.args['frameskip'])
        print("Burn Complete!")

    def generate_episode(self, policy, epsilon, mode='train', frameskip=10):
        """
```

```
        Collects one rollout from the policy in an environment.
        """
        # print('generating new ep, mode =', mode)
        done = False
        next_obs, info = self.env.reset(seed=None)
        obs = {agent: torch.Tensor(next_obs[agent]).permute(2, 0, 1).to(device) for agent in self.env.agents}
        total_ep_rewards = 0
        td_error = []
        gamma = 0.99

        while self.env.agents and not done:
            actions = {agent: policy(self.q_nets[agent].model.forward(obs[agent].unsqueeze(0).to(device)), epsi
            # print("new actions")
            # cyc_num += 1
            # if cyc_num % 100 == 0:
            #   print("cycle", cyc_num)
            i = 0
            while (i < frameskip) and not done:
                next_obs, rewards, dones, _, info = self.env.step(actions)

                total_ep_rewards += sum(rewards.values())
                i += 1

            if mode in ['train', 'burn_in']:
                next_obs_tensor = {agent: torch.Tensor(next_obs[agent]).permute(2, 0, 1).to(device) for agent in
                for agent in self.env.agents:

                    action_tensor = torch.tensor(actions[agent], dtype=torch.long).unsqueeze(0).unsqueeze(1).to(
                    current_q = self.q_nets[agent].model(obs[agent].unsqueeze(0).to(device)).gather(1, action_te


                    next_q = self.q_nets[agent].model(next_obs_tensor[agent].unsqueeze(0).to(device)).max(1)[0].


                    current_td_error = rewards[agent] + gamma * next_q * (1 - dones[agent]) - current_q

                    self.memories[agent].append(obs[agent], actions[agent], rewards[agent], next_obs_tensor[agen
                    # print("current_td", current_td_error)
                    td_error.append(current_td_error)
            elif mode in ['test']:
                next_obs_tensor = {agent: torch.Tensor(next_obs[agent]).permute(2, 0, 1).to(device) for agent in
                for agent in self.env.agents:

                    action_tensor = torch.tensor(actions[agent], dtype=torch.long).unsqueeze(0).unsqueeze(1).to(
                    current_q = self.q_nets[agent].model(obs[agent].unsqueeze(0).to(device)).gather(1, action_te

                    next_q = self.q_nets[agent].model(next_obs_tensor[agent].unsqueeze(0).to(device)).max(1)[0].

                    current_td_error = rewards[agent] + gamma * next_q * (1 - dones[agent]) - current_q
                    # print("current_td", current_td_error)
                    td_error.append(current_td_error)

            total_ep_rewards += sum(rewards.values())
            obs = next_obs_tensor if mode in ['train', 'burn_in'] else {agent: torch.Tensor(next_obs[agent]).per
            if mode == 'train' :
                # print('training run')
                for agent in self.env.agents:

                    self.train_dqn(agent)
        # print(td_error)
        # print(torch.mean(torch.stack(td_error)))
        if not td_error:
            return total_ep_rewards, []
        return total_ep_rewards, torch.mean(torch.stack(td_error))

    def plots(self):
        """
        Plots:
        1) Avg Cummulative Test Reward over 20 Plots
        2) TD Error
        """
        reward, time =  zip(*self.rewards)
        plt.figure(figsize=(8, 3))
        plt.subplot(121)
        plt.title('Cummulative Reward')
        plt.plot(time, reward)
        plt.xlabel('iterations')
        plt.ylabel('rewards')
```

```
        plt.legend()
        plt.ylim([0, None])

        loss, time =  zip(*self.td_error)
        plt.subplot(122)
        plt.title('Loss')
        plt.plot(time, loss)
        plt.xlabel('iterations')
        plt.ylabel('loss')
        plt.show()

    def epsilon_decay(self, initial_eps=1.0, final_eps=0.05):
        if(self.epsilon > final_eps):
            factor = (initial_eps - final_eps) / 10000
            self.epsilon -= factor
```

Helpers and Hyperparameters

This class contains helper functions, as well as some extra arguments that you can use to tune or play around with your DQN. There is no required parts to fill in here.

In [ ]:

```
# Note: if you have problems creating video captures on servers without GUI,
#       you could save and relaod model to create videos on your laptop.
def test_video(agent, env_name, episodes):
    # Usage:
    #   you can pass the arguments within agent.train() as:
    #       if episode % int(self.num_episodes/3) == 0:
    #           test_video(self, self.environment_name, episode)
    save_path = "%s/video-%s" % (env_name, episodes)
    if not os.path.exists(save_path): os.makedirs(save_path)

    # To create video
    env = agent.env # gym.wrappers.Monitor(agent.env, save_path, force=True)
    reward_total = []
    state = env.reset()
    done = False
    print("Video recording the agent with epsilon {0:.4f}".format(agent.epsilon))
    while not done:
        q_values = agent.q_network.model.forward(torch.from_numpy(state.reshape(1, -1)))
        action = agent.greedy_policy(q_values)
        i = 0
        while (i < agent.args['frameskip']) and not done:
            screen = env.render(mode='rgb_array')
            plt.imshow(screen[0])
            ipythondisplay.clear_output(wait=True)
            ipythondisplay.display(plt.gcf())

            next_state, reward, done, info = env.step(action)
            reward_total.append(reward)
            i += 1
        state = next_state
    print("reward_total: {}".format(torch.sum(torch.tensor(reward_total))))
    ipythondisplay.clear_output(wait=True)
    env.close()


def init_flags():

    flags = {
        "env": "pursuit", # Change to "MountainCar-v0" when needed.
        "render": False,
        "train": 1,
        "frameskip" : 2,
        "network_update_freq": 10,
        "log_freq": 10,
        "test_freq": 25,
        "save_freq": 500,
        "learning_rate": 5e-4,
        "memory_size": 1000,
        "epsilon": 0.5,
        "model_file": None, # './models/CartPole-v0/{}/model_5000.h5
        "exp_selection": "gaussian",
        "share_freq": 1,
        "beta": 0.1,
        "k": 1500,
        'batch_size': 16
```

```
    }

    return flags


def main(render=False):
    args = init_flags()
    args["render"] = render

    # You want to create an instance of the DQN_Agent class here, and then train / test it.
    q_agent = DQN_Agent(args)

    # Render output videos using the model loaded from file.
    if args['render']: test_video(q_agent, args['env'], 1)
    else:
      q_agent.train()  # Train the model.
      test_video(q_agent, args['env'], 1) # test your implementation in a video
    return q_agent
```

In [ ]:
```
# For training
q_agent = main()

# For just evaluating video. Pass in model_file in args.
# main(render=True)
```

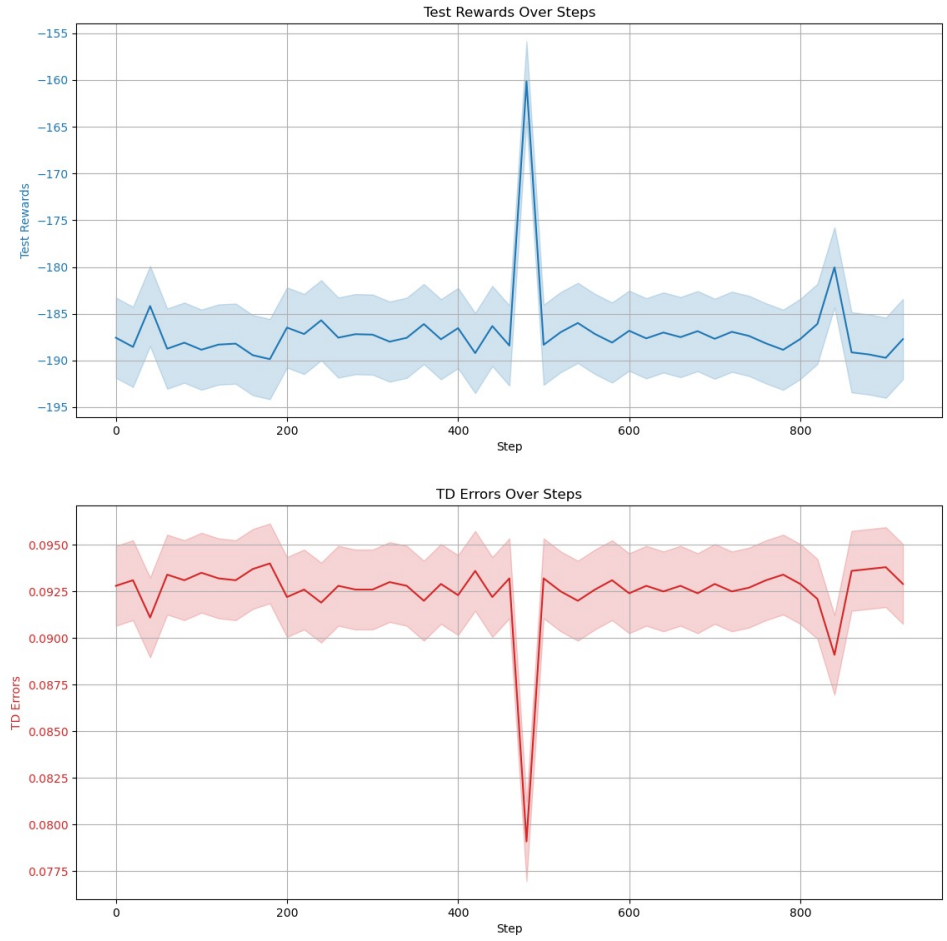

Figure 3: Our results after implementing SUPER-DDQN on Colab

References

[1] M. Bettini, A. Prorok, and V. Moens. Benchmarl: Benchmarking multi-agent reinforcement learning. *arXiv preprint arXiv:2312.01472*, 2023.

[2] F. Christianos, L. Schäfer, and S. V. Albrecht. Shared experience actor-critic for multi-agent reinforcement learning. *CoRR*, abs/2006.07169, 2020.

[3] M. Gerstgrasser, T. Danino, and S. Keren. Selectively sharing experiences improves multi-agent reinforcement learning, 2024.

[4] A. Lazaridou, A. Peysakhovich, and M. Baroni. Multi-agent cooperation and the emergence of (natural) language. *CoRR*, abs/1612.07182, 2016.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[6] W. Qiu, X. Ma, B. An, S. Obraztsova, S. Yan, and Z. Xu. Rpm: Generalizable multi-agent policies for multi-agent reinforcement learning. *International Conference on Learning Representations, https://iclr.cc/virtual/2023/poster/11728*, 2023.

[7] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. In Y. Bengio and Y. LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[8] J. Terry, B. Black, N. Grammel, M. Jayakumar, A. Hari, R. Sullivan, L. S. Santos, C. Dieffendahl, C. Horsch, R. Perez-Vicente, et al. Pettingzoo: Gym for multi-agent reinforcement learning. *Advances in Neural Information Processing Systems*, 34:15032–15043, 2021.

[9] C. Yu, A. Velu, E. Vinitsky, Y. Wang, A. Bayen, and Y. Wu. The surprising effectiveness of ppo in cooperative, multi-agent games. *arXiv preprint arXiv:2103.01955*, 2021.

[10] Y. Zhu, Y. Zhan, X. Huang, Y. Chen, yujie Chen, J. Wei, W. Feng, Y. Zhou, H. Hu, and J. Ye. OFCOURSE: A multi-agent reinforcement learning environment for order fulfillment. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023.