Autonomous driving using Multi-Agent RL

Ayon Biswas[1] and Shiladitya Biswas[2]

***Abstract*— In this paper we utilise multi-agent Reinforcement learning algorithm, Multi-Agent Deep Deterministic Policy Gradient (MADDPG) on a four lane traffic intersection scenario and compare it against the performance of Proximal Policy Optimization (PPO). In order to improve the performance stability of MADDPG, we employ prioritised experience replay(PER). We study the effect of batch size and multi-cpu training and experience collection on the individual performance of agents.**

I. INTRODUCTION

In robotics, learning is an essential component of intelligent behaviour. However, each individual agent need not learn everything from scratch by its own discovery. Instead they exchange information and knowledge with each other and learn from other agents like humans do the same with their peers or teachers. When a task is too big for a single agent to handle they may cooperate in order to accomplish the task. Human Society is the best example of Multi-Agent Reinforcement Learning.

Unfortunately, traditional reinforcement learning approaches such as Q-Learning or policy gradient are poorly suited to multi-agent environments. One issue is that each agent's policy is changing as training progresses, and the environment becomes non-stationary from the perspective of any individual agent. This presents learning stability challenges and prevents the straightforward use of past experience replay, which become invalid as the training distribution changes. Replay buffer is crucial for stabilizing deep Q-learning. Similarly, policy-gradient algorithms like A3C[1] and PPO[2] may also struggle in multi-agent settings, as the credit assignment problem becomes increasingly harder with more agents. In Multi-agent RL, each agent's action and observation space is restricted to only model the components that it can affect and those that affect it. Therefore, to have higher scalable learning we generally decompose the actions and observations of a single big agent into multiple simpler agents. This not only reduces the dimensionality of agent inputs and outputs, but also effectively increases the amount of training data generated per step of the environment. Inn contrast to a single super-agent that is prone to overfitting to a particular environment, Good decompositions of policies can be transferable across different variations of an environment.

In MADDPG[3], a general-purpose multi-agent learning algorithm is proposed which offers the following:

- learned policies that only use local information
- does not assume a differentiable model of the environment and differentiable communication between agents dynamics
- applicable not only to cooperative interaction but to competitive or mixed interaction involving both physical and communicative behaviour

[1]Ayon Biswas, MS, ECE, University of California, San Diego aybiswas@eng.ucsd.edu

[2]Shiladitya Biswas MS, ECE, University of California, San Diego s1biswas@eng.ucsd.edu

II. BACKGROUND AND METHODS

A. MDP and RL

Markov decision process (MDP) forms a key concept of Reinforcement Learning. RL enables the agent to learn a policy that returns high rewards by interacting with an unknown environment. Such environments are often depicted by a Markov Decision Processes (MDPs), described by a five-tuple (S,A,P,R,γ), where S is the State Space, A is the action space, P is the transition probabilities, and γ represents the discount factor. At each time step t, an agent interacting with the environment observes a state $s_t \in S$,and chooses an action $a_t \in A$, which determines the reward $r_t \sim R(s_t;a_t)$ and next state $s_{t+1} \sim P(s_t;a_t)$.The purpose of RL is to maximize the cumulative discount rewards $G_t = \sum_{\tau=t}^{T} \gamma^{\tau-t} r_\tau$, where T is the time step when an episode ends, t denotes the current time step, $\gamma \in [0,1]$ is the discount factor, and r_τ is the reward received at the time step τ. The action–value function (abbreviated as Q-function) of a given policy π is defined as the expected return starting from a state–action pair (s,a), expressed as

$$Q^{\pi}(s,a) = E[G_t.s_t = s, a_t = a, \pi]$$

Q-learning is a widely used RL algorithm that uses the action–value function $Q^{\pi}(s,a)$ to learn the policy . DQN[4] is a kind of RL algorithm combining Q-learning and neural network, which learns the action–value function Q^* corresponding to the optimal policy by minimizing the loss: $L(\theta) = E^{\pi}[(Q^{\theta}(s,a) - y)^2]$, where $y = r + \gamma \max Q_0^{\theta}(s_0,a_0)$, and y represents the Q-learning target value.

B. Markov games

A Markov game[5], which is a multi-agent extension of Markov decision processes (MDPs), for N agents is defined by a set of states S describing the possible configurations of all agents, a set of actions $A_1,...,A_N$ and a set of observations $O_1,...,O_N$ for each agent. To choose actions, each agent i uses a stochastic policy $\pi_{\theta_i} : O_i \times A_i \to [0,1]$, which produces the next state according to the state transition function $T : S \times A_1 \times ... \times A_N \to S$. Each agent i obtains rewards as a function of the state and agent's action $r_i : S \times A_i \to R$, and receives a private observation correlated with the state $o_i : S \to O_i$. The initial states are determined by a distribution $\rho : S \to [0,1]$.

Each agent i aims to maximize its own total expected return $R_i = \sum_{t=0}^{T} \gamma^t r_i^t$ where γ is a discount factor and T is the time horizon.

C. RL Approach

1) Q-learning: Q-Learning is an Off-policy temporal difference(TD) control technique. At time step t, an action is picked according to Q values, $A_t = argmax\ _{a \in A} Q(St, a)$ and $\in$-greedy exploration is commonly applied. After applying action A_t, we observe reward R_{t+1} and get into the next state S_{t+1}. We update the Q-value function as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a) - Q(S_t, A_t)) \tag{1}$$

Many-times people use a function (i.e. a machine learning model) to approximate Q values, this technique is called function approximation. Deep Q-Network ("DQN"; Mnih et al. 2015, [6]) aims to greatly improve and stabilize the training procedure of Q-learning by two innovative mechanisms:

- **Experience Replay**: All the episode steps $e_t = (S_t, A_t, R_t, S_{t+1})$ are stored in one replay memory $D_t = \{e_1, \ldots, e_t\}$. D_t has experience tuples over many episodes.
- **Periodically Updated Target**: Q is optimized towards target values that are only periodically updated. The loss function looks like this:

$$L(\theta) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \Big[\big(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \big)^2 \Big] \tag{2}$$

2) Policy-gradient: Policy Gradient methods [7] instead learn the policy π directly with a parameterized function with respect to model parameter $\theta, \pi(a|s;\theta)$. The key underlying idea is to push up the probabilities of actions that lead to higher return, and push down the probabilities of actions that lead to lower return, until we arrive at the optimal policy. The Gradient of the policy can be written as follows:

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla \ln \pi(a|s,\theta) Q_\pi(s,a)] \tag{3}$$

Deterministic policy gradient (DPG)[8] models the policy as a deterministic function: $a = \mu(s)$. The policy gradient for this case can be stated as:

$$\begin{aligned} \nabla_\theta J(\theta) &= \int_S \rho^\mu(s) \nabla_a Q^\mu(s,a) \nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)} ds \\ &= \mathbb{E}_{s \sim \rho^\mu}[\nabla_a Q^\mu(s,a) \nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)}] \end{aligned} \tag{4}$$

Deep deterministic policy gradient (DDPG)[9] is a variant of DPG where the policy μ and critic Q_μ are approximated with deep neural networks. DDPG is an off-policy algorithm, and samples trajectories from a replay buffer of experiences that are stored throughout training.

D. Policy optimisation

Trust Region Policy Optimization or TRPO updates policies by taking the largest step possible to improve performance, while satisfying an optimisation constraint on how close the new and old policies are allowed to be. TRPO aims to maximize the objective function $J(\theta)$ subject to trust region constraint which enforces the distance between old and new policies, which is measured by KL-divergence to be small, within a parameter δ:

$$\mathbb{E}_{s \sim \rho^{\pi_{\theta_{\text{old}}}}} [D_{\text{KL}}(\pi_{\theta_{\text{old}}}(.|s) \| \pi_\theta(.|s)] \leq \delta \tag{5}$$

Proximal policy optimization (PPO) simplifies it by using a clipped surrogate objective while retaining similar performance. PPO imposes the constraint by forcing $r(\theta)$ to stay within a small interval around 1, precisely $[1-\in, 1+\in]$, where $\in$ is a hyper-parameter.

$$J^{\text{CLIP}}(\theta) = \mathbb{E}[\min(term_1, term_2)] \tag{6}$$

$term_1 = r(\theta)\hat{A}_{\theta_{\text{old}}}(s,a)$ and $term_2 = \text{clip}(r(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_{\theta_{\text{old}}}(s,a)$

The objective function of PPO takes the minimum one between the original value and the clipped version and therefore the motivation for increasing the policy update to extremes for better rewards is lost.

1) Prioritised experience replay: Prioritised experience[10] is a strategy to sample from the experience buffer by selecting samples with high TD-error, which can lead to faster learning of polices and Q-functions. However, sampling purely based on TD error introduces bias to the learning as experiences with low TD error might not be replayed for a long time and making the model prone to over-fitting. To counter this, importance sampling weights for bias correction is performed, which involves multiplying the gradient by the importance sampling weights of the experiences after every update step. The importance sampling weights are computed as:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)} \right)^\beta \tag{7}$$

where N is the batch size and P(i) is the rank of the sample in the buffer, which represents the probability of sampling data point i according to priorities.

E. Multi-Agent Actor Critic

A primary motivation behind MADDPG[3] is that, if we know the actions taken by all agents, the environment is stationary even as the policies change, since $P(s_0|s, a_1, ..., a_N, \pi_1, ..., \pi_N) = P(s_0|s, a_1, ..., a_N) = P(s_0|s, a_1, ..., a_N, \pi_{01}, ..., \pi_{0N})$ for any $\pi_i \neq \pi_i'$. This is not the case if not explicitly conditioned on the actions of other agents, as done for most traditional RL methods. MADDPG differs from the previous efforts in the following ways:

- the learned policies can only use local information (agent's own observations) at execution time
- it does not assume a differentiable model of the environment dynamics
- it does not assume any particular structure on the communication method between agents

It accomplishes the above goals by adopting the framework of centralized training with decentralized execution. More

concretely, consider a game with N agents with policies parameterized by $\theta = \{\theta_1, ..., \theta_N\}$, and let $\pi = \{\pi_1, ..., \pi_N\}$ be the set of all agent policies. Then we can write the gradient of the expected return for agent $i, J(\theta_i) = E[R_i]$as:

$$\nabla\theta_i J(i) = E_{sp^{\mu}, a_i \pi_i}[\nabla\theta_i log\pi_i(a_i|o_i)Q_i^{\pi}(x, a_1, ..., a_N)]$$

Here $Q_i^{\pi}(x, a_1, ..., a_N)$ is a centralized action-value function that takes as input the actions of all agents, $a_1, ..., a_N$, in addition to some state information x, and outputs the Q-value for agent i. The complete MADDPG alogrithm is detained in 1

Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient for N agents

for *episode=1 to M* **do**
- Initialize a random process N for action action exploration;
- Receive initial state x;
- **for** *t=1 max-episode-length* **do**
  - for each agent i, select action $a_i = \mu_{\theta_i}(o_i) + \mathbb{N}_t$ w.r.t. the current policy and exploration. Execute actions $a = (a_1, ..., a_N)$ and observe reward r and new state x_0 Store (x, a, r, x_0) in replay buffer D;
  - x $\leftarrow x'$
  - **for** *agent i=1 to N* **do**
    - Sample a random minibatch of S samples (x_j, a_j, r_j, x'_j) from D;
    - Set $y^j = r_i^j + \gamma Q_i^{\mu}(x'^j, a'_1, ...a'_N)|_{a'_k = \mu'(o_k^j)}$;
    - Update Critic by minimizing Loss L(θ)
    - Update actor using the sampled policy gradient: ;
    - $\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}(o_i^j)\nabla_{a_i} Q_i^{\mu}(x^j, a_1^j, ..., a_i, ...a_N^j)$
  - **end**
  - Update Target Network Parameters:
$$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$$
- **end**

end

Where $L(\theta) = \frac{1}{S}\sum_j (y^j - Q_i^{\mu}(x^j, a_1^j, ..., a_i, ...a_N^j))^2$

Knowing the observations and policies of other agents is not a particularly restrictive assumption, this information is often available to all agents in communicative scenarios. However, this assumption can be relaxed by learning the policies of other agents from observations.

III. Experiments and Results

A. SMARTS Simulator

Scalable Multi-Agent RL Training School i.e. SMARTS[11] is a simulation platform that supports the training, accumulation, and use of diverse behaviour models of road users. These models are then used to create increasingly more realistic and diverse road interactions that enable deeper and broader research on multi-agent interaction. The Key elements of SMARTS for supporting Multi-Agent Reinforcement Learning (MARL) research are as follows:

Observation, Action and Reward: The observation space for an agent is specified as a configurable subset of available sensor types that include dynamic object list, bird's-eye view occupancy grid maps and RGB images, ego vehicle states, and road structure etc. An agent's action space is determined by the controller chosen for it. SMARTS supports four types of controllers: ContinuousController, ActuatorDynamicController, TrajectoryTrackingController and LaneFollowingController. The mapping between the controllers and the action space is table I

TABLE I: Controller to action space mapping Table

Controller	Action Space Type	Control Command Dimensions
LaneFollowingController	Mixed	Target speed lane change (+1 or 0 or -1)
TrajectoryTrackingController	-	trajectory
ActuatorDynamicController	Continuous	throttle brake steering rate rad
ContinuousController	Continuous	throttle brake steering

The **observation space** is a stack of three consecutive frames, each containing: 1) relative position of goal; 2) distance to the centre of lane; 3) speed; 4) steering; 5) a list of heading errors; 6) at most eight neighbouring vehicle's driving states (relative distance, speed and position); 7) a bird's-eye view gray-scale image with the agent at the centre.

The **Action space** is a 4 dimensional vector of discrete values, for longitudinal control, keep lane and slow down; and lateral control—turn right and turn left

The **Performance** of a policy is judged depending on the average collision rate and completion rate of the agent population on 10 episodes under two different background traffic settings

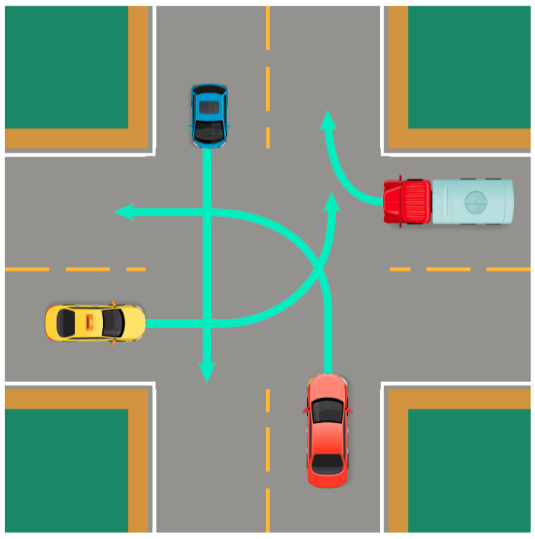

Fig. 1: 4-lane intersection environment with 4 agents

And finally, SMARTS also provides a **Behavioural Analysis** tool, that enable us to analyse the behavioural difference of the algorithms under different traffic scenarios.

B. Ray-distributed DL and RL

Ray[12] provides a simple API for building distributed applications by providing simple primitives for building and running distributed applications, enabling parallelization of single machine code and including a large ecosystem of applications, libraries, and tools to enable complex applications. RLlib[13] is an open-source library for reinforcement learning that offers both high scalability and a unified API for a variety of applications. RLlib natively supports TensorFlow[14], TensorFlow Eager, and PyTorch[15]. Tune is a Python library for experiment execution and hyperparameter tuning at any scale.

C. Experiment Setup

We use ray, rllib and tensorflow to conduct our experiments on our personal Laptop equipped with 16GB VRAM and GTX 1050 with 4GB graphics memory. In all the following experiments, we use two CPU nodes- one worker-node to collect experience and the other node to train. We conducted experiments to gain insight into performance of MADDPG with respect to batch size, experience collection by using prioritized replay and/or by using distributed MADDPG. In the experiments that follow, we have shown individual rewards for agents as opposed to combined episodic reward, which can be misleading. An example would be, very high reward by two agents and meagre rewards by the remaining agents as opposed to decent(no so high reward) by all the agents. Clearly, the second case is favourable.

In all our experiments, both critic and actor are fully connected networks of size 64 x 64 with reLU activations. The critic learning rate used was 1e-3 and that of actor was 1e-4. we used a batch size of 1024 unless otherwise stated. All the experiments were performed on the 4-lane intersection environment with 4 agents as shown in Figure 1

D. Comparison to PPO

We first compare MADDPG to PPO, implemented for multi-agents with their agents polices sharing weights. We observed that training multi-agent PPO took 3-3.5 times the duration taken by MA-DDPG. However, PPO produced superior results fort all the agents-0 to 3 as shown in Figure 2. But in cases, where the traffic density is high, in those cases large-scale PPO may suffer from the credit assignment problem as it would get increasingly difficult for the policy to assign blame or reward to specific agents causing that negative or positive change respectively.

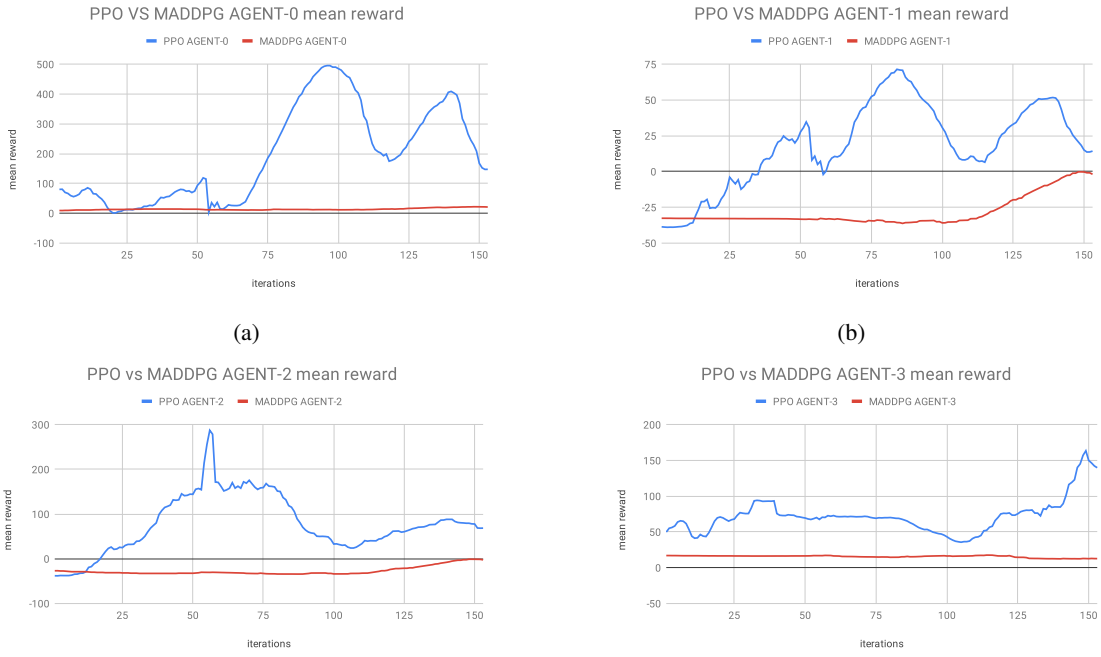

Fig. 2: mean episode reward for 4 agents for MADDPG with vs PPO with shared weights

E. Effect of Prioritised Experience Replay

We now compare MADDPG with typical experience collection against that which uses prioritised experience replay(PER). As shown in Figure 3, PER greatly improves the stability and causes an almost monotonic increase in the performance during training. The reason is attributed to the sampling of experiences that have the maximum TD-error and also the rank term used in PER ensures that states that have low TD-error are also sampled leading to high rate of learning and consequently low variance in performance.

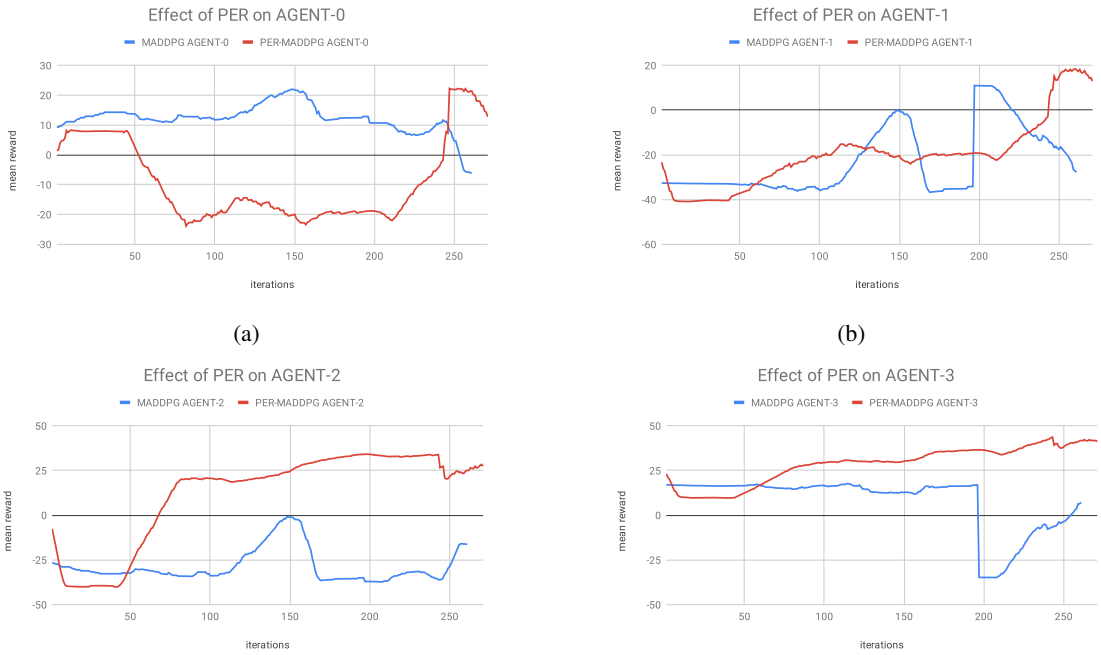

Fig. 3: mean episode reward for 4 agents for MADDPG with Prioritised Experience Replay vs uniform random sampling

F. Effect of batch-size

To observe the effect of batch size, we run MADDPG with a batch-size of 64. To our surprise, we observe better performance for all the agents as shown in Figure 4. The reason may be that smaller batch sizes are noisy imparting a regularising effect and lower generalisation error as was shown in [16]

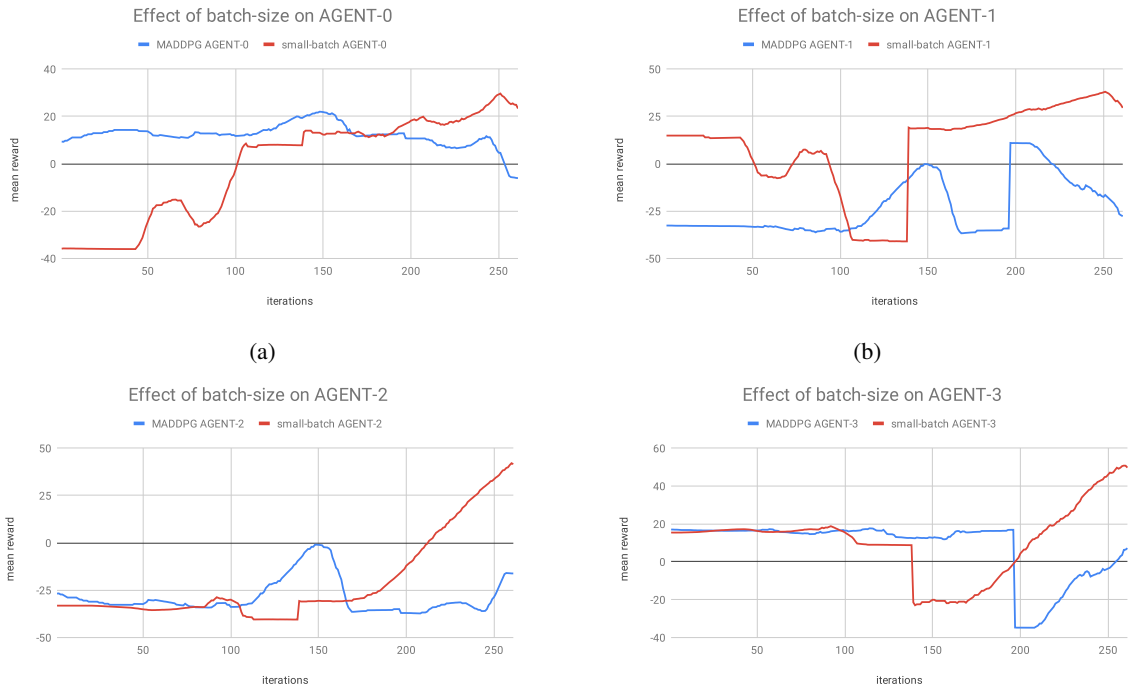

Fig. 4: mean episode reward for 4 agents for MADDPG with Batch Size = 64(small batch vs batch size =1024 (original)

G. Comparison with distributed MADDPG

Finally we compare MADDPG training over single node that does both experience collection and training vs one node that does experience collection and two other nodes that aid in training and update weights asynchronously. As shown in Figure 5, we observe a much higher performance in distributed MADDPG. The reason is often attributed to the effect of exploration in parameter space due to asynchronous update of neural-network weights. This result is similar to that observed in a3c as compared to a2c.

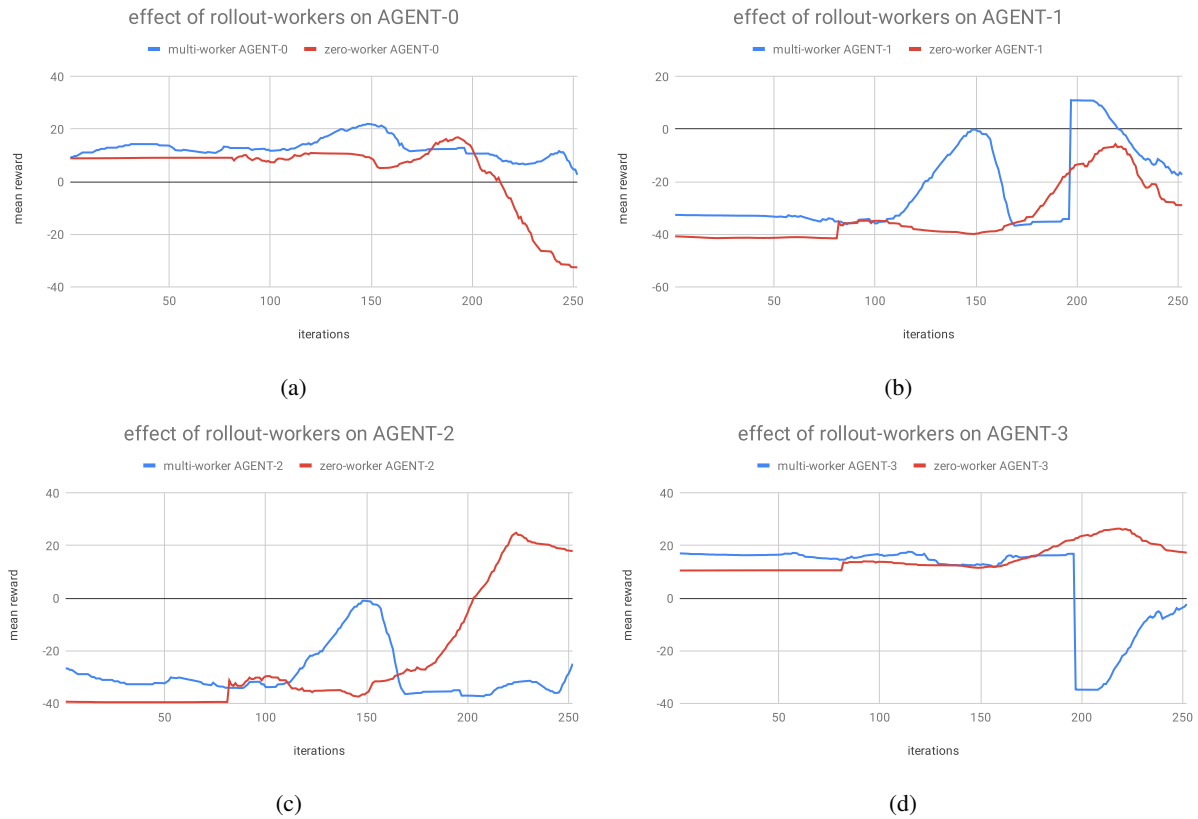

Fig. 5: mean episode reward for 4 agents for single-CPU MADDPG VS distributed MADDPG

IV. Discussion and Future Work

In this project, we gained insight into two class of multi-class algorithms namely single agent RL applied to many agents with shared weights(like multi-agent PPO) and centralised critic that took in info of the actions of each agent(MADDPG). we furthermore saw Prioritised experience replaat can boost the performance and stability of MADDPG. In the future, we would like to continue this endeavour and compare the performance and stability of MA-PPO and MA-DDPG with increasing density of traffic. Instead of directly feeding actions of all agents in the central critic, an ensemble of mapper functions can be learnt in an online function that can infer actions of other agents from observing them.

V. Acknowledgement

We would like to thank Prof Michael Yip and the TA for the ECE276C, Reinforcement Learning for Robotics Course and there guidance that helped us to carry out this project.

References

[1] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[2] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[3] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *CoRR*, abs/1706.02275, 2017.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[5] Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *In Proceedings of the Eleventh International Conference on Machine Learning*, pages 157–163. Morgan Kaufmann, 1994.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[7] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[8] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 387–395, Bejing, China, 22–24 Jun 2014. PMLR.

[9] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[10] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.

[11] Ming Zhou, Jun Luo, Julian Villella, Yaodong Yang, David Rusu, Jiayu Miao, Weinan Zhang, Montgomery Alban, Iman Fadakar, Zheng Chen, Aurora Chongxi Huang, Ying Wen, Kimia Hassanzadeh, Daniel Graves, Dong Chen, Zhengbang Zhu, Nhat Nguyen, Mohamed Elsayed, Kun Shao, Sanjeevan Ahilan, Baokuan Zhang, Jiannan Wu, Zhengang Fu, Kasra Rezaee, Peyman Yadmellat, Mohsen Rohani, Nicolas Perez Nieves, Yihan Ni, Seyedershad Banijamali, Alexander Cowen Rivers, Zheng Tian, Daniel Palenicek, Haitham bou Ammar, Hongbo Zhang, Wulong Liu, Jianye Hao, and Jun Wang. Smarts: Scalable multi-agent reinforcement learning training school for autonomous driving, 2020.

[12] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 561–577, USA, 2018. USENIX Association.

[13] Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning, 2018.

[14] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[15] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[16] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks, 2018.