Intro to Machine Learning - Final Project

Ben Giacalone Quynh Duong Marie Mellor Mohammad Bayat

December 2023

1 Machine Learning Task

1.1 Task Definition

The task is to create agents that can work in cooperative, competitive, and mixed cooperative-competitive environments. Multi-agent environments are applicable in various problem spaces, demonstrating success in domains such as multi-robot control [2], multiplayer games [11], and traffic routing [8].

In this context, it is crucial to understand the concept of reinforcement learning (RL). Reinforcement learning can be viewed as a way of solving Markov Decision Processes (MDPs) [1], a flexible way of describing sequential problems. On each timestep until termination, an algorithm must take in a *state* (also described as an observation), output an *action*, and collect a *reward* [12]. RL algorithms must therefore learn to convert states to actions in such a way that they maximize their expected return, i.e. the sum of all rewards from the current timestep. This function that maps states to actions constitutes a *policy*.

The challenge with multi-agent RL is that the optimality of a policy does not depend solely on an individual agent — instead, agents must take into account the behavior of other agents as well. Tasks in multi-agent RL are typically categorized in three ways: pure competition, pure cooperation, and mixed-sum. In pure competition settings, agents play against each other to obtain desired rewards (*e.g.* they are placed in a zero sum game). In pure cooperation, agents collaborate to achieve shared rewards. In mixed-sum settings, agents engage in both competition and cooperation with each other to achieve the desired rewards.

1.2 Evaluation Protocol

Agent Rewards We measure the rewards collected by agents as they train. Seeing raw rewards allows us to identify not only whether or not high-scoring policies were learned, but the optimality of policies throughout the training process. We display these as reward curves.

Task Specific Metrics Given the diversity of environments, each environment has its own distinct definition of success. For example, in `simple_spread`, we measure the average number of collisions per episode to evaluate the agents' ability to cover landmarks and avoid collisions. We use these metrics to identify if policies improved or worsened. These metrics can be seen in the tables containing our results.

1.3 Data

Rather than using a dataset as standard in other machine learning tasks, reinforcement learning uses *environments* that agents repeatedly sample from. We plan on using the Multi Agent Particle Environments [7] released by OpenAI in 2017. This package contains a number of environments that test cooperative and competitive behavior in agents. Before running our experiments, we decided to remove `simple_crypto` and `simple_reference`, since these environments would not have been affected by our architectural changes. We describe the environments we did use below.

`simple_spread` A cooperative environment where agents work together to cover all landmarks while avoiding colliding with each other.

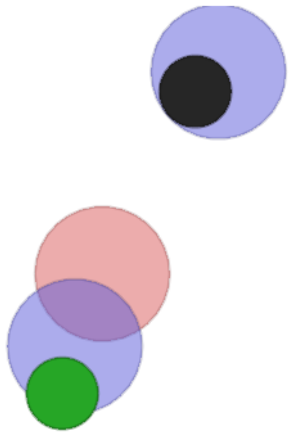


Figure 1: A visualization of the `simple_adversary` task. Blue agents must cover the green landmark, while ensuring the red agent (the adversary) cannot figure out the correct landmark.

`simple_adversary` A mixed-sum setting where good agents must work together to be as close to the target landmark as possible, while distracting and preventing the adversary from getting close to the target landmark.

`simple_tag` A mixed-sum setting where an agent attempts to avoid being touched by a team of adversary agents. Large landmarks in this environment provide the opportunity for cover and choke points.

2 Machine Learning Model

In this section, we explain the algorithm used to train our models, MADDPG. We then explain how we use a permutation invariant architecture to more naturally handle agent information in a multi-agent setting.

2.1 MADDPG

Deep Q-Networks (DQNs) [6] extend the Q-learning framework described in Watkins and Dayan [14] for reinforcement learning in discrete action spaces by replacing the Q-value lookup table with a neural network. To stabilize the network, they introduce *experience replay*, which allows the model to gradually integrate new experience into its learning regime, preventing catastrophic forgetting. They also introduce *target networks*, which mitigate the moving target problem by only updating the Q-target network after hundreds of steps have passed.

The Deep Deterministic Policy Gradient (DDPG) algorithm [4] uses findings from Silver et al. [10] to allow the use of continuous action spaces. Two models are now used, a *policy network* that takes in state observations and outputs continuous actions, and a *value network* that is parameterized by both a state and continuous action, returning the predicted Q-value[1]. Rather than employing an expensive search for the action that would maximize the critic's Q-value, the policy is directly trained to maximize the critic's output for given states.

While DDPG works well in the single agent setting, it underperforms in environments where agents interact with each other. Lowe et al. [5] identify that this is due to nonstationarity in the environment when multiple agents are introduced — the predicted value of an agent's action is determined not only on its own policy, but the policy of other agents as well. This is demonstrated in the paper with the speaker-listener

[1]The policy network is often called the *actor* and the value network the *critic* in reinforcement learning literature. We use both sets of terms interchangeably.

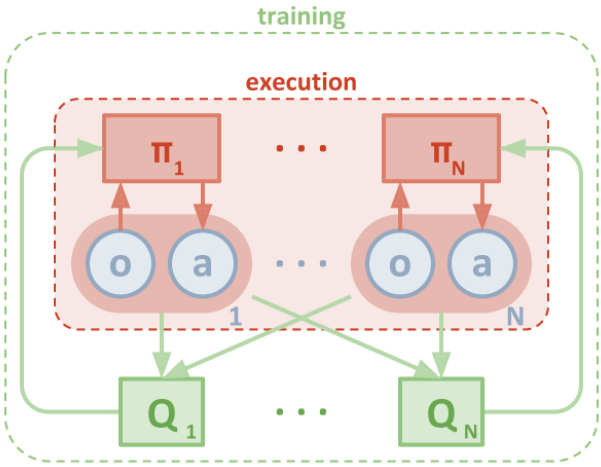


Figure 2: Each policy (denoted as $\pi_1..\pi_N$) learns a centralized critic (denoted as $Q_1..Q_N$) that is conditioned on observations and actions from other agents. During execution, only the policies are used. Figure taken from Lowe et al. [5].

environment, where one agent broadcasts a message to another agent indicating where a landmark is, while the other agent listens to the message and moves towards the landmark. This is a cooperative task where both agents receive the same reward. Since the speaker's reward is based on the listener's movements, it can be punished even when broadcasting the correct landmark.

To fix this, the authors propose the Multi-Agent Deep Deterministic Policy Gradient (MADDPG) algorithm. As shown in Figure 2, MADDPG learns *centralized critics* for each policy, which include the observations and actions of other agents in the environment. By accounting for the behavior of other agents, critics have an easier time of performing credit assignment.

For completeness, we will briefly summarize the MADDPG algorithm, which defines the loss function for the networks. As with most RL algorithms, it is iterative, going back and forth between sample collection and training. On each iteration, the policy networks ingest their respective agent's current observation (i.e. state) and output an action, causing the environment to return a reward. These state, action, reward tuples are saved into a buffer. The critic is trained to better approximate the expected return via mean squared error (i.e. $Critic(S_t, A_t) \leftarrow r_t + \gamma * Critic(S_{t+1}, Policy(S_{t+1}))$ for all agents' states S, all agents' actions A, the reward r, and policy and critic networks $Critic$ and $Policy$ at timestep t), and the actor is trained via gradient ascent to maximize the critic's expected return[2]. When computing the predicted return, the critic takes into account *every* agent in the environment.

2.2 Permutation Invariant Architectures

The original MADDPG experiments and codebase use simple architectures — 2-layer multi layer perceptrons (MLPs) — to make their point. Recall that this algorithm requires two networks, a policy network that is used at runtime to convert the current state into a set of (continuous) actions, and a critic network used only during training that nudges the policy network towards actions leading to higher returns. While the policy network only has to take in a single agent's observations, the critic must take the observations and actions of *every* agent in the environment. In order to fulfill this requirement, the original paper concatenates every agent's observations and actions into a single, large vector, and passes this directly to the critic.

This decision makes sense for a relatively small number of agents with unique observation and action spaces. However, there are a couple issues with this approach for larger problems:

- Many multi-agent reinforcement learning problems feature large numbers of homogeneous agents. Controlling a drone swarm, for example, requires taking in a uniform set of observations (e.g. speed, battery

[2]The algorithm also uses target networks and noise to improve stability and reduce overfitting; these details have been omitted for brevity.

levels, distance to other drones) and outputting a uniform set of actions (e.g. power sent to each motor) over as many as 100 agents. This can quickly lead to an explosion of features for the critic.

- Additionally, if observations and actions truly are uniform across agents, the network will learn redundant features, in the same way a feedforward network will learn redundant features when operating on images.

- Finally, there are environments where the number of agents vary over the course of an episode, e.g. by being killed by another agent. A fixed architecture either ends up masking the majority of its inputs due to dead agents or cannot generalize to a higher number of agents than seen during training.

These issues can be mitigated by using architectures that can operate on entire sets of agents. Many permutation invariant architectures exist for operating on batches of same-sized vectors.

The Transformer [13] operates on a set of fixed-length vectors, incorporating positional embeddings to allow treating the set as a list. "Query", "'key", and "value" representations are each computed for each input vector, and information is combined by taking the dot product of each vector's "key" vector and each vector's "value" vector to parameterize a softmax distribution, then taking the dot product of this softmax distribution and the value vectors. These combined vectors are summed with the original input vectors, resulting in a set of vectors with the same dimension and count as the original set.

Graph neural networks (GNNs) [9], in particular, the graph convolutional networks (GCNs) proposed by Kipf and Welling [3], operate on a set of node features. Information from neighboring nodes are aggregated via message passing, by convolving each node's features with its neighbors. After this message passing operation, the number of nodes and node features stays the same.

For this paper, we are most interested in the DeepSet architecture proposed by Zaheer et al. [15]. As indicated by the name, this architecture is explicitly designed to be used in set-based settings, showing good performance on set expansion and image tagging tasks. A linear layer (or set of linear layers) individually process each feature vector in a given set, producing a set of computed features. These computed features are then pooled together to produce a single vector. Unlike the Transformer, DeepSets are much less computationally expensive, and unlike GCNs, they do not require a notion of neighboring nodes.

3 Experiment

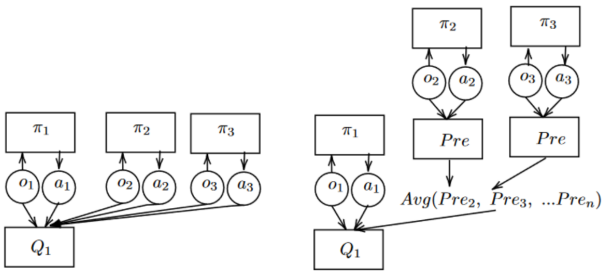


Figure 3: Left: The original network architecture used by MADDPG. Observations and actions from all agents are concatenated together before being passed to the critic. Right: Our proposed modification, where a preprocessing network (Pre) independently processes the observations and actions of all other agents before combining them into a single, compact representation through averaging.

3.1 Design

We aim to test the following hypothesis: **Our simple permutation invariant architecture will not perform worse than the baseline.** For our independent variable, we test whether or not we choose to use our modified architecture. If an environment is mixed-sum, we also check the effects of only replacing adversaries or good agents, to see how one team's performance changes when only one of them is modified.

Our dependent variables are the same as in the original MADDPG paper, the task specific metrics defined by each environment.

In our multi-agent system design, the critical component is the preprocessing of agent observation and actions. The deep-set block design goal is to process the data agents' data (observation and actions). In the block of Deep Set block there are 3 different layers designed to extract features from the agents' obsevations and actions. The output of these convolutional layers is then averaged. The averaging of the output takes away any permutation invariance, which ensures that the output does not depend on the order of input data.

The data segregation of observation and action depends on the agent group. This segregation is crucial for feeding the data into the correct data set block. Furthermore, for the processing steps, each segregated data is passed through its respective deep-set block. In this step, the convolutional layers process the data and the permutation-ivariant features are extracted.

3.2 Methodology

In our experiments, we modify the original code's architecture, which is a 2-layer MLP with 128 hidden units. As shown in Figure 3, we take each agent's observations and actions, and we run them through a DeepSet style preprocessing network. We define two preprocessing networks per environment, one to handle adversaries, and one to handle good agents. This is due to the fact that these "teams" have different observation and/or action spaces, leading to different sized vectors. Each preprocessing network consists of 3 1D convolutional networks with a kernel size of 1 to simulate independently running an MLP over each vector. We do not increase the number of hidden units to mitigate the possibility of increasing performance solely due to more parameters. Once the feature vectors have been preprocessed, they are averaged, then each "team"'s contextualized feature vectors are concatenated, then passed to the baseline architecture.

We initialize two Deep Set block preprocessors. These blocks are designed to handle inputs from different agent groups (e.g., adversaries and non-adversaries) separately. The decision to use two separate blocks is based on the hypothesis that different agent groups may exhibit distinct behavioral patterns that could be better captured with dedicated processing units.

Each agent is assigned a preprocessor based on its group. In our implementation, the first group (adversaries) is assigned the first Deep Set block, and the second group (non-adversaries) is assigned the second block. This assignment is managed using the preproc idx list, which maps each agent to the appropriate preprocessor. During training, we split the observation and actions of the agents based on their assigned preprocessors. Each set of observations and actions is then processed using the specific assigned processors.

The output features from each block are concatenated to form a unified representation of the environment's state. This representation is then fed into subsequent layers of our model.

We run our experiments on the `simple_adversary`, `simple_tag`, and `simple_spread` environments. A description of these tasks are present in Section 1. We list out the individual metrics here in more detail:

- `simple_spread`: The number of collisions between agents, and the average distance to each landmark. The goal of the agents is to cover all landmarks while minimizing collisions.
- `simple_adversary`: Across all timesteps, the percentage of time a good agent is covering the target landmark, and the percentage of time the adversary is covering the landmark.
- `simple_tag`: The average number of times an adversary touches a good agent. The better the good agent, the less times they are touched.

As a baseline, we use the original architectures proposed by the paper, which come with the codebase by default.

4 Results and Discussion

4.1 `simple_spread`

Based on Table 1, the number of collisions produced by modified architecture on the simple spread model is lower than the baseline architecture and the average distance average agent distance from the landmark

Agent π	# collisions	Average dist.
Linear	5.771	0.927
Preprocessed	0.019	2.102

Table 1: Average # of collisions per episode and average agent distance from a landmark in the cooperative navigation task. The linear model is the baseline model.

Agent π	Adversary π	AG succ %	ADV succ %	Δ succ %
Linear	Linear	92.7%	70.7%	22%
Preprocessed	Preprocessed	53.1%	31.1%	22%
Preprocessed	Linear	93.2%	72.1%	21.1%
Linear	Preprocessed	93.2%	32.2%	61%

Table 2: Success (succ %) for agents (AG) and adversaries (ADV) is if they are within a small distance from the target landmark. The linear model is the baseline model.

produced by modified architecture on the simple spread model is higher than the baseline architecture. This result is because in the modified architecture, the agents are focusing on running away from each other hence the lower collision rate rather than focusing on minimizing the distance to the landmark.

4.2 `simple_adversary`

Based on Table 2, the modified architecture underperformed compared to the baseline architecture. We tried to only change either the agent or the adversary to the modified architecture to see how they would perform compared to the baseline architecture and modified architecture. Based on the results from above, the adversary performed worst using the preprocessor. Therefore, any agent using a preprocessor will perform worse than the baseline architecture.

4.3 `simple_tag`

In Table 3, we see that using the modified architecture causes worse performance over the baseline. In the baseline case, there was an average of about 1 touch per episode. When we switched the adversary to the modified model, this number dropped to about 0.05, barely ever touching the good agents. When we modify the good agents to use the permutation invariant architecture, the number of times the adversary is able to touch the agents skyrockets, at about 9 touches per episode. Therefore, the preprocessing block inhibits agent performance.

4.4 Conclusion

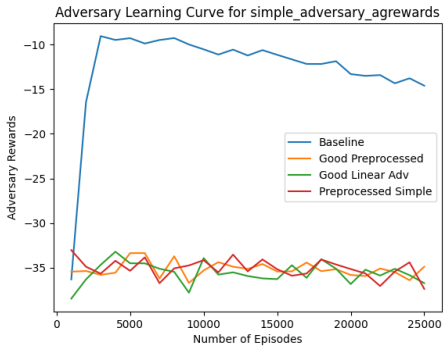
Across all of our experiments, the trend is clear: using our proposed architecture worsens performance. This contradicts our hypothesis.

We believe one reason why the permutation invariant architecture underperformed is to the fact that we combine the target agent's observations with all other agent observations and rewards, rather than isolating it out. The averaging operation diffuses information from in the gradient across all agents, weakening the policy training process. One direction we could look at in future work, then, is to explicitly separate out the target agent's observations and actions from the rest, to see if that helps with policy training.

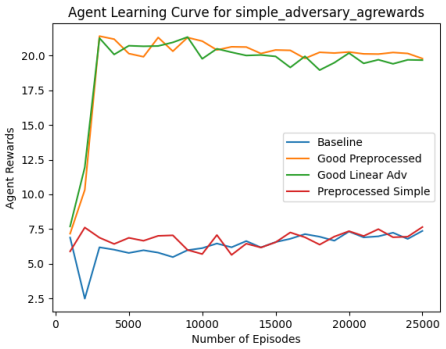
Another possible reason for our modified architecture underperforming is that we do not increase the number of hidden units in our deep-set blocks, instead keeping them at the number of agent features. As mentioned previously, we did this to mitigate the chance of any performance improvements coming solely from the increased number of parameters. However, this limits the expressivity of the network; intermediate features may have a difficult time moving across the averaging gap. Future work could see how doing something like doubling the number of hidden units translates to improvements in performance.

Agent π	Adversary π	# touches
Linear	Linear	1.025
Preprocessed	Preprocessed	0.216
Preprocessed	Linear	9.773
Linear	Preprocessed	0.054

Table 3: Average # of touches for agents (AG) and adversaries (ADV). The linear model is the baseline model.

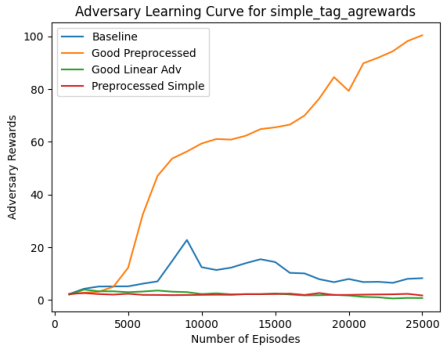


(a) Comparison of the learning curve of `baseline_simple_adversary_agrewards` to the preprocessed adversaries created from changing the architecture.



(b) Comparison of the learning curve of `baseline_simple_adversary_rewards` to the preprocessed agents created from changing the architecture.

Ultimately, we still believe permutation invariance is a necessary consideration for multi-agent reinforcement learning. As more systems are developed that take advantage of this paradigm, we will no doubt see exactly what does and does not matter when incorporating permutation invariance.



(a) Comparison of the learning curve of `baseline_simple_tag_agrewards` to the preprocessed adversaries created from changing the architecture.

(b) Comparison of the learning curve of `baseline_simple_tag_rewards` to the preprocessed agents created from changing the architecture.

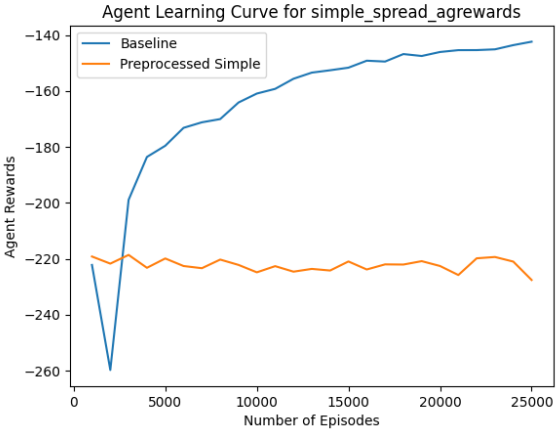


Figure 4: Comparison of the learning curve of `baseline_simple_spread_rewards` to the preprocessed agents rewards created from changing the architecture. Note that no adversaries exist for this environment.

References

[1] Richard Bellman. A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684, 1957. ISSN 0022-2518.

[2] Yixin Huang, Shufan Wu, Zhongcheng Mu, Xiangyu Long, Sunhao Chu, and Guohong Zhao. A multi-agent reinforcement learning method for swarm robots in space collaborative exploration. In *2020 6th International Conference on Control, Automation and Robotics (ICCAR)*, pages 139–144, 2020. doi: 10.1109/ICCAR49639.2020.9107997.

[3] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016. URL `http://arxiv.org/abs/1609.02907`.

[4] 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. *arXiv preprint arXiv:1509.02971*, 2015.

[5] 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. URL `http://arxiv.org/abs/1706.02275`.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[7] Igor Mordatch and Pieter Abbeel. Emergence of grounded compositional language in multi-agent populations. *arXiv preprint arXiv:1703.04908*, 2017.

[8] Anum Mushtaq, Irfan Ul Haq, Muhammad Azeem Sarwar, Asifullah Khan, Wajeeha Khalil, and Muhammad Abid Mughal. Multi-agent reinforcement learning for traffic flow management of autonomous vehicles. *Sensors*, 23(5), 2023. ISSN 1424-8220. doi: 10.3390/s23052373. URL `https://www.mdpi.com/1424-8220/23/5/2373`.

[9] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605.

[10] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. Pmlr, 2014.

[11] Joseph Suarez, Yilun Du, Phillip Isola, and Igor Mordatch. Neural MMO: A massively multiagent game environment for training and evaluating intelligent agents. *CoRR*, abs/1903.00784, 2019. URL `http://arxiv.org/abs/1903.00784`.

[12] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018. ISBN 0262039249.

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL `http://arxiv.org/abs/1706.03762`.

[14] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[15] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabás Póczos, Ruslan Salakhutdinov, and Alexander J. Smola. Deep sets. *CoRR*, abs/1703.06114, 2017. URL `http://arxiv.org/abs/1703.06114`.