Multi-Agent Reinforcement Learning: Transfer and Algorithms

Hussein Mouzannar
American University of Beirut
hmm46@aub.edu.lb

Abstract

This manuscript details some of the literature in transfer learning for reinforcement learning tasks and multi-agent systems. In addition, we will explore a new decentralized scalable algorithm for multi agent reinforcement learning. The algorithm is an online actor-critic with a modular action-value function learned using agent influence measures and a decentralized policy. We evaluate the approach and compare to single agent methods on a multi agent particle environment domain. This work was done during a semester long research course.

Introduction

Reinforcement Learning (RL) is concerned with situations where an agent acts in a sequential environment mapping his observations to actions aiming to maximize a numerical reward given by the environment [50]. An example of a reinforcement learning task could be a game of chess: on each turn the player observes the board and then decides on which piece to move in order to win the game.

These tasks are usually modeled as Markov Decision Processes (MDP), the agent interacts with the outside world, the environment, at a set of discrete time steps. The setup is as follows: at each time step the agent observes the environment state and then decides on an action and performs it, in response to his action, the environment presents him with a reward and transitions to a new state. Therefore our end goal is learning a policy that maps states in actions which uses knowledge gained from previous experiences in order to maximize the agent's reward.

One unique challenge in reinforcement learning is the trade-off between exploitation: repeating actions that have lead to good reward in the past, and exploration: trying new actions never selected before. An effective method for solving MDPs must balance between the two.

Recently, traditional RL methods have combined with Deep Learning algorithms to solve problems where the input is a high dimensional such as in vision or speech, this method is called Deep Q-Learning [27] based on Q-Learning [60] and uses experience replay to help train deep neural networks from raw pixels. DQN was successful at playing Atari games at human level and was combined with regular artificial intelligence methods namely tree search to defeat the world champion at the game of Go [46]. This has caused a massive surge of interest in RL, however the uses of RL go beyond games. RL technologies have reduced Google data centre cooling bill by 40% [7], optimized the treatment policies for chronic illnesses [45] and numerous applications to robotics [16]. However, currently the interest in RL still does not quite match its practical applications.

Single agent RL makes the unrealistic assumption that agents learn in the presence of non-learning agents and without adversarial competition. The presence of the other learning agents breaks the Markovian assumption made by RL methods which states that the current state description constitutes complete information for decision making. Most environments in the real world are not static and contain other learning agents that are also trying to maximize their rewards, examples include self-driving cars [43], labor-markets [56] and more [36]. Thus the need to directly model other agents during learning, to model communication with other agents and to learn joint policies arises. This has led to the insurgence of multi agent reinforcement learning [42] which aims to learn agents policies in potentially competitive or cooperative environments [55] [14].

Another issue with RL that constitutes the main cause of its little adoption in practical applications is it's sample inefficiency. RL requires considerably more samples than supervised learning to arrive at good policies: the agent has to spend a millions of time steps acting in the environment until he figures out how to maximize his reward. There are two approaches to combat this: create more sample efficient RL algorithms [6] [59], or benefit from previous experience of related environments and tasks, this is referred to as transfer learning [53]. RL agents start out tabula-rasa, acting randomly in the environments and learning from trial and error which is very expensive when acting in the real world e.g. robots falling while learning to walk.

This paper attempts to construct a multi-agent reinforcement learning (MARL) algorithm to handle cooperative or competitive environments that is comparatively sample efficient. We learn decentralized policies for each agent that only rely on local observations and communicated information from neighboring agents. Multi-agent systems can be composed of a big number of agents interacting, however for a specific agent not all agents will have a big influence on his policy and thus do not need to be taken into consideration. An example of this is a multi-agent team surveying an unknown territory, an agent only needs to observe nearby agents that are working near him. Another example which is usually used as a benchmark for MARL is robotic soccer, the central defender only has to model the opposing team attackers. Thus our agent learns not only his policy but also which agents to take into consideration that affect his rewards the most, this allows the learning algorithm not to depend on the total number of agents in the environment which would be expensive otherwise.

Our algorithm is an online on-policy algorithm, we do not separate between training and execution. It learns modular Q-values with discrete actions in continuous state spaces for a given subset of agents that influence the actor the most based on two proposed importance measures. Given these Q-values it trains a decentralized policy that relies only on local information.

A primer in Reinforcement Learning

Setup

Reinforcement Learning tasks are formulated as MDPs [50] which consist of the tuple (S, A, T, γ, R). At each time step t, the agent is in state $s \in S$ and executes an action $a \in A$ and consequently transitions to state $s' \sim T(s, a, s')$ with probability $p(s'|a, s)$ and receives reward $r \sim R(s, a, s')$. The agent's goal is to maximize his discounted reward over his life in the environment: $\sum_{i=1}^{T} \gamma^i r_i$ where γ is the discount rate $0 \leq \gamma \leq 1$ which determines the value of future rewards..

The MDP framework is abstract and flexible to be applied to many different problems: the time steps need not be actual time intervals but successive stages of decision making, the actions can be low-level voltage on robotic arm, or high level (deciding to write a good report or not) [50]. The only assumption is to have a state description that captures all the history of the environment and is sufficient for decision making.

The agent's acts in the environment according to a policy: a mapping from states to probabilities of selecting each possible action. We denote by $\pi(a|s)$ the probability that the agent chooses action a given that he is in state s.

Two important functions widely used in RL algorithms are the the value function and the action-value function. The value function $v_\pi(s) = \sum_{a \in A} \pi(a|s) \sum_{s',r} p(s'|a, s) * (r + v_\pi(s))$ is the expected reward when following policy π starting at state s, it tells us how good is it to be in state s. The action-value function $q_\pi(s, a)$ is the expected reward the agent would receive if in state s he performed action a and then followed with policy π.

Dynamic Programming

When the state space S and action space A are finite and the environment transition is known, we can solve for the optimal policy π^*. This policy is guaranteed to exist and it leads to optimal value and action-value functions. In fact, the two functions satisfy what is called the Bellman optimality condition:

$$q^*_\pi(s,a) = \sum_{s',r} p(s'|a,s) * (r + \gamma \max_{a'} q^*_\pi(s,a'))$$

$$v^*_\pi(s) = \max_a \sum_{s',r} p(s'|a,s) * (r + \gamma v^*_\pi(s))$$

Using this fact, one approach to find π^* will be to iteratively apply the bellman equation for v^* in a dynamic programming fashion for all states until convergence. Now using the optimal value-function, we obtain a deterministic policy $\pi(s) = arg\max_a \sum_{s',r} p(s'|a,s) * (r + \gamma v^*(s))$, this is named value iteration. Another approach is to obtain the value-function for an initial policy, then try to greedily improve on this policy and repeat until the policies converge: this is policy iteration.

Temporal Difference (TD) Learning

When the dynamics of the environment are not known, two approaches can be applied: estimate the transition function from experience and then apply dynamic programming, this class of methods are *model-based*, or directly try to estimate the value function in a *model-free* fashion.

TD-Learning [49] is a prediction algorithm that learns the value function $v(s)$ for a given policy in the following manner: the agent acts in the environment and adjusts the value function according to: $v(s) \leftarrow v(s) + \alpha(r + \gamma v(s') - v(s))$ where α is a learning rate and the quantity $r + \gamma v(s') - v(s)$ is referred to as the TD-error.

One of the key early algorithms inspired by TD is Q-learning [60]. The algorithm tries to learn the optimal Q^* irrespective of the policy being followed, this approach is called *off-policy* . The Q-value is learned iteratively by bootstrapping from previous estimates using the following equation:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

Function Approximation and DQN

When dealing with continuous state representations, i.e. a vector of real numbers, it is infeasible to calculate $|S| \times |A|$ Q-values. Instead we approximate the Q-values with a parameterized function mapping from states and actions to Q-values. Thus the action-value function is parameterized by a parameter θ and denoted as $Q(s,a|\theta)$. This parameterization could be linear: $Q(s,a|\theta) = \theta^T(S+A)$ or a neural network where the weights of the network θ parameterize the Q-value function. Now our update is with respect θ and a loss function with the aim of minimizing the TD-error. With the squared loss our updates will be:

$$\theta = \theta + \alpha \nabla_\theta (r + \gamma \max_{a'} Q(s',a'|\theta) - Q(s,a|\theta))^2$$

DQNs [28] have popularized the use of neural networks as approximators with the introduction of a replay buffer D that stores tuples of experience (s,a,r,s') to be re-optimized over and with adaptive learning rate.

Policy Gradient

Another approach for solving MDPs with a continuous state space is by directly trying to optimize the end policy itself in order to increase the reward.

Policy gradient (pf) methods [51] are a class of RL algorithms that do not require having a value function but instead learn by optimizing a parameterized policy. The policy is denoted as $\pi(a|s,\theta)$ with our parameter $\theta \in \mathbb{R}^{d'}$ and denotes the probability of selection action a at time step t given that the agent is in state s with parameter θ.

Thus to learn, we define a performance measure $J(\theta) \doteq v_{\pi_\theta}(s_0)$ that is the true value of the initial state under our policy, and we will maximize this measure with respect to θ using for example gradient ascent.

First of all let us see how we parameterize our policy: we denote $h(s,a,\theta) \in \mathbb{R}$ as our preference for the state-action pair, the higher this value is, the more advantageous it is to select this action in the given state. Our preference function can be computed using a neural network with input a state representation and thus our θ becomes the weights of the network. Now if we have a discrete action set, our policy can now be parameterized as:

$$\pi(a|s,\theta) = \frac{exp(h(s,a,\theta))}{\sum_b exp(h(s,b,\theta))}$$

Given our policy parameterization, the policy gradient theorem allows us to write the gradient of our policy as:

$$\nabla_\theta J(\theta) \propto \mathbb{E}_\pi[\sum_a q(s,a)\nabla_\theta \pi(a|s,\theta)]$$

The REINFORCE algorithm [61] estimates this gradient by using sample estimates and thus the gradient of our policy becomes:

$$\nabla J(\theta) = \mathbb{E}_\pi[G_t \frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}]$$

where G_t is the Monte Carlo return, and the updates subsequently become:

$$\theta_{t+1} \doteq \theta_t + \alpha G_t \frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}$$

One problem with REINFORCE is the high variance of the updates, to counteract this a common solution is to subtract a baseline from the action-value function that does not vary with the action. A natural baseline is an estimate of the state value $\hat{v}(S_t, \mathbf{w})$ where $\mathbf{w} \in \mathbb{R}^m$ thus the update will be:

$$\theta_{t+1} \doteq \theta_t + \alpha(G_t - b(S_t)\frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}$$

We can also transform REINFORCE to an online algorithm using multi-step returns or just a one-step return as in $TD(0)$. This approach is referred to as Actor-Critic [17] as computing the state-value requires bootstrapping from previous estimates. The one-step Actor Critic updates are:

$$\theta_{t+1} \doteq \theta_t + \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))\frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \tag{1}$$

Policy gradient methods can be extended to cases where the action space is also continuous via Deterministic Policy Gradients (DPG) [47]. With a deterministic policy we can write the policy gradient as:

$$\nabla_\theta J(\theta) = E[\nabla_\theta \pi_\theta(a|s)\nabla_a Q^\pi(s,a)|_{a=\pi(s)}]$$

Deep deterministic policy gradient (DDPG) [22] is a variant of DPG where the critic and and the actor are both approximated by deep neural networks, it is an off-policy algorithm that samples trajectories from a replay buffer.

For a general survey of deep reinforcement learning please refer to [21].

Transfer in Reinforcement Learning

One approach to speed up learning in RL tasks is to transfer knowledge from other related tasks to the one at hand [53]. This is the *transfer learning* (TL) paradigm [32]s, which has proved to be successful in supervised learning especially in image recognition tasks [18].

We make the distinction between the *source* task, on which the learner has previously acquired knowledge on, and the *target* task which is to be learned. The learner must first choose an appropriate source task to transfer from, then learn how the tasks are related and different through inter-task mappings and finally transfer information from the source task to speed up learning.

We review some selected papers that discuss transfer in RL.

Single Task

In [20], the authors study transfer between tasks with similar dynamics but with different reward and transition functions. The idea is to store transition samples (s, a, s', r) from the source task, and learn an estimate for their true reward $\hat{R}$ and transition function $\hat{T}$ to reuse in the target task. It adopts the optimism in the face of uncertainty heuristic by having the reward estimate modified with the addition of a confidence interval $\hat{R} \leftarrow \hat{R} + CI$, they also extend this idea to continuous state tasks using confidence bands.

Additionally, a recent line of work in transferring between tasks with differing reward functions has been explored using what's called *successor features*. The setup is based on decomposing a learned reward function in a given tasks in such way to show the tasks decomposition into multiple subtasks [3] [25].

To transfer between tasks with different state spaces but similar action spaces, [15] contribute a new transfer algorithm. After learning a policy π^S for the source task, the agent in the target tasks maps his current state to a state in the source tasks using an Unsupervised Manifold Alignment technique: $s_t^S = \chi^+(s_t^T)$, using the source policy we obtain the action $a_t^S = \pi^S(s_t^S)$ which we assume has an equivalent a_t^T in target domain, and simulate in the source task to transition to s_{t+1}^S, this state is then mapped to the target domain via $s_{t+1}^T = \chi^-(s_{t+1}^S)$. Now this data is used to train the target task policy.

Multiple Task Transfer

The setup in the following works is to transfer from a library of source tasks which have already been learned to a single target task. We make the distinction between multiple task transfer and multi-task learning which learns multiple related tasks jointly.

Evolving Neural Architectures

PathNet is a neural network algorithm that has embedded agents that determine which layers of the network to re-use for new tasks [8]. A PathNet is a modular neural network, where each layer consists of modules which themselves are neural networks. A genetic or a reinforcement learning algorithm determines which modules are active in each layer, this is what called a pathway. Concretely a PathNet consists of L layers each with M modules, a pathway is $N \times L$ boolean matrix which sets the output of a module to zero or not. We first learn on the source task using gradient descent and find out the optimal pathway according to a loss function, afterwards the weights on the optimal path are fixed and the rest are reset, learning on the target task repeats the same process. PathNet can be thought of as evolutionary dropout, where networks are dropped instead of units.

Similar to this, progressive neural networks [39] solves the problem of catastrophic forgetting when learning tasks sequentially. A progressive neural network starts with one single column which itself is a neural network with parameters Θ_1 and learns these weights on the first task using back propagation. Once a new task is presented, parameters Θ_1 are frozen and a new layer is added to the network with parameters Θ_2 which are then also learned. This process extends to learning multiple tasks.

Finally attentive deep architectures for transfer (A2T) [35], first learns K neural networks for tasks $T_1, \cdots, T_K$, for transfer on the target task, the algorithm learns a convex combination of the weights of the K networks to build the target network.

Deep Multi-task Learning

DQN

The following paper [33] attempts to learn one DQN network able to play multiple games (Atari) at once benefiting from similarity across tasks and able to transfer knowledge on new tasks. The approach consists of first training an expert agent E for each task $S_1, \cdots, S_N$ which are used to acquire training data from sampling from the networks. Afterwards, the global or target network objective is to match the experts policy guided by cross-entropy and the experts feature representation of the states with a squared loss: this is called the actor mimic objective. They are able to show the convergence of this method under some constraints, the network is able to reach expert level on many games and occasionally exceed the expert.

Another approach [38] is to transfer knowledge from multiple agents, action policies, to an untrained DQN which is smaller and more efficient. Their approach is based on policy distillation, given experts in multiple tasks the aim is to try to match their Q-values. First a dataset is generated of tuples (s, a, Q) from the source tasks and afterwards regression on a neural network is performed with three different tried losses: Negative likelihood loss, mean-squared error and Kullback–Leibler (KL) divergence. Experiments on 10 Atari games show that the distilled network is able to match performance with a significantly smaller network size.

Policy Gradient

To solve multi-task learning in situations where tasks can be naturally decomposed, [2] develops a framework for multi-task RL where tasks are annotated with a sketch of subtasks required for the main task completion. The approach learns a modular sub policy for each subtask by a decoupled actor-critic objective where the baseline is task specific, then the task policy is simply the concatenation of the sub policies in order.

Formally, each MDP task τ has its own R_τ where R the reward function and p_τ the initial distribution across states. Each task is annotated with a sketch K composed of high level actions $b_1, b_2, \cdots$ drawn from fixed vocabulary β. For each high level action b we have a subpolicy π_b: subpolicy chooses actions $\in A$ or *STOP* action. Thus given a sketch for task, our policy Π_τ is the concatenation of the sub-policies:

Execute actions from π_{b_i} until the action *STOP* is generated, then pass the control to $\pi_{b_{i+1}}$

Additionally the learner goes through a curriculum learning scheme to avoid over-fitting by learning easier tasks first. The method was evaluated on a maze environment and a crafting environment.

To handle problems in continuous action spaces, [62] proposes a new algorithm (multi-DDPG) based on DDPGs. The method retains one critic for all actors, thus an actor for each task, which has to output action values to guide all others, thus the critic loss is the sum of losses for each actor. They evaluate their method using a set of robot movement based control tasks.

Finally, [54] develops a new approach for multi-task learning based on policy gradient. Workers attempt to learn their own tasks while constrained to be close to the global "distilled policy" by a KL penalty.The global policy is trained to be the centroid of all task policies.

Multi-Task Learning Framework

Different from the previous section, we now consider the multi-task (MT) learning framework popularized first by [63] and [19] that has resulted in a rich literature on MT in supervised and RL settings.

In the MT reinforcement learning (MTRL) the agent faces a series of m MDP tasks $T_1, \cdots, T_m$ each characterized by the tuple $(S_t, A_t, T_t t, \gamma_t, R_t)$. The goal is for the agent to learn optimal policies $\Pi = (\pi_1, \cdots, \pi_m)$ each parameterize by a vector θ_t.

The major assumption in this framework is that there exists $k < m$ latent basis tasks such that all tasks can be represented as linear combination of these basis tasks denoted [19]. This allows us to enables us to write the parameters of each task t as $\theta_t = L \cdot s_t$, where L is the common knowledge-base and s_t is the task specific state description. Furthermore the representation s is forced to be sparse so as to only rely on a small number of tasks.

Thus now we can formulate our MTRL objective with $J(\theta_t)$ denoting the expected return in task t with policy parameterized by θ_t [1]:

$$\min_{L, s_1, \cdots, s_m} -\frac{1}{m} \sum_{t=1}^{T} (J(L \cdot s_t) + \mu ||s_t||_1) + \lambda ||L||_F$$

This objective is inefficient to solve because of the explicit dependence on all available trajectories and evaluation of a single candidate L that requires the optimization of all s. [1] solves this by first plugging in the second taylor order expansion of the expected return and splitting the objective for each task and relying on distributed solvers.

Moving to the Muli-agent setting

We now move to the multi-agent (MA) setting where the cooperative and competitive tasks are now formulated as stochastic games. A stochastic game is a tuple (n, S, A, T, γ, R) [44] in which at each time step t, the n agents are in the environment state $s_t \in S$ choose actions $\mathbf{a} = a_1, \cdots, a_n \in A^n$ that cause the environment to transition to state s_{t+1} according to the transition function $T : S \times A^n \times S \to [0, 1]$. Each agent receives a reward according to the reward function $R : S \times A^n \times S \to \mathbb{R}^n$, that is $R = (r_1, \cdots, r_n)$. A stochastic game can be considered as the combination of an MDP and a matrix game.

If we consider the partially observable setting, we define a partially observable stochastic game as the tuple $(n, S, A, T, \gamma, R, \mathbb{O}, O)$. Each agent will have a stochastic policy $\pi_i : \mathbb{O}_i \times A_i \to [0, 1]$ where $\mathbb{O}_i$ is the set of observations available to agent i where these observations are drawn from the observation probability function $O : S \times A \to \mathbb{O}$.

Solving Discrete Stochastic Games

An important concept for solving stochastic games is that of the Nash-Equilibrium, we want to have each agent playing rationally meaning his strategy is optimal with respect to the other agents policies. The Nash-Equilibrium is a collection of joint-policies such that:

$$V_i(s, \pi_1^*, \cdots, \pi_i^*, \cdots \pi_n^*) \geq V_i(s, \pi_1^*, \cdots, \pi_i, \cdots \pi_n^*) \; \forall i : 1, , n, \pi_i \in \Pi_i, s \in S$$

The above equation tells us if the agent deviates from his optimal strategy then his reward will decrease.

To extend Q-values to the multi-agent (MA) case it is necessary to consider joint-actions instead of individual actions, thus the Q-value for each agent becomes $Q(s, a_1, \cdots, a_n)$. Nash Q-Learning [14] defines Nash Q-values (NashQ) as the expected reward for agent i when all agents follow the Nash equilibrium, and defines an analogous update to Q-Learning for each agent Q-values with the NashQ value. To calculate the NashQ values requires the Q-values of all others agents, thus needing to maintain n Q-value tables. Multiple other algorithms exist for solving general-sum stochastic games which include Fried or Foe Q-learning [23] and WoLF policy hill-climbing [5], see [42] for a complete survey.

Single Agent methods in MA setting

Let us first consider applying traditional single agent RL methods to the MA domain, we refer to these approaches as Independent Learning (IL). With Independent Q-learning (IQL) [52], we have our agent i learn his own Q_i conditioning only on his own observations and actions. One major issue with this approach is that if we have other agents independently learning then the world would become non-stationary as $P(s'|s, a_1, \cdots, a_i, \cdots, a_n) \neq P(s'|s, a_1', \cdots, a_i, \cdots, a_n)$, fixing our agent's action might lead to different results depending on the other agents actions.

While this approach is scalable and successful in tabular cases, to apply it in continuous domains algorithms like DQN require techniques such as experience replay buffers to effectively learn. However, since the environment is non-stationary the replay buffers become obsolete over time and may hurt the learning process. To fix this issue, [10] proposes two methods: the first is multi-agent importance sampling which weights each experience sample (s, a, s', r, t) by the degree of which other agents policies have changed from time-step t to the present. The drawback to this method is requiring the training to be centralized to have access to other agents policies. To eliminate non-stationarity, we must condition on other agent's actions as:

$$P(s'|s, a_1, \cdots, a_n|\pi_1', \cdots, \pi_n') = P(s'|s, a_1, \cdots, a_n|\pi_1, \cdots, \pi_n) \forall \; \pi i \; and \; \pi_i'$$

On the other hand we have independent policy gradient (IPG) algorithms and accordingly independent actor critic methods. Policy gradient usually have high variance updates, this is exaggerated in the multi-agent setting as the agent's reward also depends on other agents actions which are not taken into consideration furthermore increasing variance of gradient estimates. [24] show that the probability of taking a gradient step in the correct direction with IPG decreases exponentially with the number of agents

For simplicity, from this point we will assume that we are in a fully observable environment unless noted otherwise and then extend the treatment to the partially observable setting. A fully observable stochastic game is a tuple (n, S, A, T, γ, R) defined as above.

Related Multi-Agent Algorithms

Decentralized policy

These approaches are characterized by learning a decentralized policy $\pi(a_i|s_i)$ for each agent. The first thing about these methods are that they assume independence of the global policy, we are inherently making the assumption that $\pi(\mathbf{a}|s) = \prod_{i=1}^{n} \pi(a_i|s)$. Within this class we separate two sub-approaches:

Individual reinforcement learning (IRL) ignores the presence of other agents and treats them as part of the environment. One can apply single agent reinforcement learning algorithms as is such as DQN, AC and more. This approach has many problems highlighted previously, however it works reasonably well in practice. Its main advantages are decentralized training and execution, scalability as it is invariant to the number of agents in the environment and being simple to train.

However, we can also integrate information from other agents while maintaining a decentralized policy. What this means is that we allow communication between agents during training or to follow the centralized learning and decentralized learning framework adopted by [24](MADDPG) and [9](COMA). Let us focus here on actor critic (AC) algorithms. AC takes steps in the direction of $\nabla J(\theta) = E_\pi[\sum_a A(s,a)\nabla\pi(a|s,\theta)]$ where A is the advantage function which is based on $Q(s,a)$. We can instead learn for each agent a value function that conditions on the observations and actions of all agents: $Q(s, a_1, \cdots, a_n)$. The motivation of this approach is that conditioning on the actions of other agents the environment becomes stationary and thus more easily solvable.

In COMA [9] the authors introduce a new MARL method called counterfactual multi-agent (COMA) policy gradients based on actor-critic methods for cooperative MARL tasks with global reward. The algorithm proposed, follows the centralized learning and decentralized execution framework, the learned policies only use local information at execution and learning does not assume any structure on communication or the environment.

COMA uses a centralized critic which conditions on the true state or the action-observation histories of all agents and individual actors that condition on their own action-observation history. But instead of naively letting each actor follow the TD error estimate which relies on the global reward that does not reflect the agent's contribution to that reward. COMA relies on a counterfactual baseline inspired by difference rewards. The centralized critic outputs action values for each state $Q(s, \mathbf{a})$ where $\mathbf{a} = (a_1, \cdots, a_n)$ and the advantage function for agent i compares the Q-value for his action a_i compared to a baseline which marginalizes his action while keeping the actions of other agents a_{-i} fixed:

$$A^i(s, a_i) = Q(s, a_i) - \sum_{a_i'} \pi_i(a_i'|s) Q(s, (a_i', a_{-i}))$$

Having solved the problem of having proper rewards, the evaluation of the critic is expensive if it is a deep NN as the output of the network would have A^n nodes: all possible actions for all agents. To fix this, the COMA critic inputs the actions of all other agents and outputs a Q-value for each of the agent's actions. Thus we need a single forward pass for the critic and actor to compute the advantage. However, a slight caveat is that the network's input are now linear in the number of agents and actions. Furthermore, COMA converges to a locally optimal policy under the conditions of single-agent actor-critic algorithms. The algorithm is tested on micromanagement tasks in StarCraft where the objective is to position individual units and attack enemy units.

To simplify the learning of the joint Q-value, [48] decomposes the joint Q into an additive combination of individual Q-values in the following manner:

$$Q(s, a_1, \cdots, a_n) \approx \sum_{i=1}^{n} Q(s, a_i)$$

The setup is a global reward, thus to learn these individual Q-values the reward has be to decomposed, however here this is done implicitly with the summation.

MADDPG [24] presents an adaptation of actor-critic methods for the multi-agent setting in cooperative and competitive environments. Unlike COMA, MADDPG maintains a centralized critic for each agent instead of for all agents, the environments have explicit communication between agents and the algorithm is capable of learning continuous policies instead of discrete policies.

To extend AC to the multi-agent setting, the critic is augmented with information about other agents policies. We let the N agents have their policies parameterized by $\theta = (\theta_1, \cdots, \theta_N)$, we now have a centralized action-value function $Q_i^\pi(s, a_1, \cdots, a_N)$ and thus we can write the gradient of the expected return J as:

$$\nabla_\theta J(\theta) = E[\nabla_\theta \pi_\theta(a|s) \nabla_a Q^\pi((s, a_1, \cdots, a_N)|_{a=\pi(s)}]$$

The algorithm is evaluated on a multi-agent particle environment and is show to have better performance than IRL methods.

[64] study the problem of MARL in the collaborative setting with individual rewards. The policies are decentralized as having a centralized controller which receives all rewards and observations is not scalable or robust to attacks. The authors consider that the agents are connected by a time-varying communication network, at each time step the agent receives information from his neighbors and using local information acts on the environment. The algorithms proposed are actor-critic methods, the actor is individual for each agent without the need for other agents policies and the critic uses values from the agent's neighbor to get a consensual estimate.

The authors define a networked multi-agent MDP by the tuple (S, A, P, R, G) where G is an undirected graph with n (agents) vertices and an edge set for each time step. They consider their objective to maximize the sum of the individual rewards:

$$\max_\theta J(\theta) \; \mathbb{E}[\sum_{t=0}^{T-1} \frac{1}{N} \sum_{i=1}^{n} r_{t+1}^i]$$

The actor-critic algorithm proposed consists of two steps: the critic step where the parameters of the agent's critic are averaged with it's neighbors to create a consensus update and the actor step is as usual for AC.

Both of these approaches require having other agents policies or learning them and additionally both depend linearly on the number of agents n courtesy of learning the global Q-value function.

Joint Policy

We can instead learn a joint policy $\pi(\mathbf{a}|s)$, an immediate consequence of this is centralized execution and training. Learning a joint policy requires to also have a global Q-value $Q(s, \mathbf{a})$, implying a global reward R shared by all agents. A global reward R does not allow for competition, to have competition we must have each team of agents learning a joint policy however their Q-values must condition on actions of all agents to satisfy stationarity.

Learning a joint-policy effectively reduces the MARL task to a single agent task whose action space is the joint action space and whose observation space is the joint-observation space of all agents. Therefore, with this we can use single agent RL algorithms to learn a joint policy.

Thus it is straightforward to define multi-agent extensions, [11] propose DDPG and DQN extensions with both centralized and decentralized policies. They experiment with factoring the joint policy, parameter sharing and concurrent training. They introduce three cooperative multi agent environments: pursuit-evasion, water-world, and coordinating bipedal walkers.

A line of work in MARL has focused on the StarCraft combat games [57] which focus on the low level short-term control of a group of agents during a combat against the enemy members. One work formulates these games as zero-sum stochastic games [34]. Agents communicate using the proposed bidirectionally-coordinated net (BiC-Net) and agent learning and control is done using a single multi-agent actor critic algorithm whose update averages out Q-values from each agent.

The disadvantages of joint learning are an exponentially growing action and state space : $\mathbb{A}^n$, this means that the algorithms now need an equivalent increase in the number of samples to converge.

Simple Extension of Actor-Critic to Multi-agent domain

We propose a simple extension of Actor-Critic to the multi-agent discrete action domain with decentralized policies is to update agent's i policy $\pi(a_i|s_i)$ using the joint action-value function $Q_i(s, a_1, \cdots, a_n)$. This is very similar to the approach of MADDPG however with non-deterministic policies. We refer to this approach as Multi-Agent Actor-Critic (MAAC), the algorithm for execution and training is shown below.

Algorithm 1: Multi-Agent Actor-Critic for n agents

```
Initialize policy neural network $\pi_{\theta_i}(a|s)$, input is observation and output is probability distribution over action space
Initialize critic neural network $Q_{w_i}(s, a)$, input is action and observations and output is Q-value.
Initialize replay buffer $\mathcal{D}$ with capacity $m$
**for** episode $= 1$ to $M$ **do**
  Reset replay buffer $\mathcal{D}$
  Receive initial state s
  **for** $t = 1$ to max-episode-length (while not terminal state) **do**
    for each agent $i$, sample action $a_i$ with $Pr = 1 - \epsilon$ from $\pi_{\theta_i}(a|s)$ and with $Pr = \epsilon$ from $A$
    Execute actions $a = (a_1, \ldots, a_n)$ and observe reward $r = (r_1, \ldots, r_n)$ and new state $\mathbf{s}'$
    Store $(\mathbf{s}, a, r, \mathbf{s}')$ in $\mathcal{D}$
    Sample a random minibatch of $S$ samples $\{(\mathbf{s}^j, a^j, r^j, \mathbf{s}'^j)\}_{j=0}^{S}$ from $\mathcal{D}$
    **for** agent $i = 1$ to $n$ **do**
      **Critic:**
      Compute $y_i^j = r_i^j + \gamma\, Q_i(\mathbf{s}'^j, a'_1, \ldots, a'_N)$, where $a'_i$ are the predicted actions for each agent given state $s'$
      Update critic by minimizing the loss $\mathcal{L}(w_i) = \frac{1}{S}\sum_j \left(y_i^j - Q_i(\mathbf{s}^j, a_1^j, \ldots, a_N^j)\right)^2$
      **Actor:**
      Update actor network in the direction of:
        $\nabla \log \pi_{\theta_i}(a|s) \cdot (Q_i(\mathbf{s}^j, a_1^j, \ldots, a_N^j) - V_i(s^j))$
    **end for**
  **end for**
**end for**
```

Assumption 1. Agents policies are independent:

$$\pi(a_1, \cdots, a_n|s) = \prod_{i=1}^{n} \pi_i(a_i|s)$$

Theorem 1. The MAAC gradient update is:

$$g = \mathbb{E}_{\pi_i}\left[\nabla \log \pi_{\theta_i}(a|s) \cdot (Q_i(\mathbf{s}, a_1, \ldots, a_N) - V_i(s))\right]$$

MAAC converges to a local maximum of the expected reward.

Proof. First let us check the expected contribution of the baseline where $\mu_i(s)$ is the state distribution under π_i [49]:

$$\begin{aligned}
b &= \mathbb{E}_{\pi_i}\left[-\nabla \log \pi_{\theta_i}(a|s) \cdot V_i(s)\right] \\
&= -\sum_s \mu_i(s) \sum_a \nabla \log \pi_{\theta_i}(a|s) \cdot V_i(s) \\
&= -\sum_s \mu(s) \cdot V_i(s) \nabla \sum_a \log \pi_{\theta_i}(a|s) \\
&= -\sum_s \mu(s) \cdot V_i(s) \nabla 1 \\
&= 0
\end{aligned}$$

Thus we can only consider the update as:

$$\begin{aligned} g &= \mathbb{E}_{\pi_i}\left[\nabla \log \pi_{\theta_i}(a|s)\cdot Q_i(\mathbf{s}, a_1, \ldots, a_N)\right] \\ &= \mathbb{E}_{\pi_i}\left[\nabla \log \pi_{\theta_i}(a|s)\cdot Q_i(\mathbf{s}, a_1, \ldots, a_N)\right] + \mathbb{E}_{\pi_i}\left[\sum_{j\neq i} \nabla_{\theta_i} \log \pi_{\theta_j}(a|s)\cdot Q_i(\mathbf{s}, a_1, \cdots, a_N)\right] \end{aligned}$$

The term on the right side evaluates to zero thus it does not effect the gradient, continuing on:

$$\begin{aligned} g &= \mathbb{E}_{\pi_i}\left[\sum_{j=1}^{n} \nabla_{\theta_i} \log \pi_{\theta_j}(a|s)\cdot Q_i(\mathbf{s}, a_1, \ldots, a_N)\right] \\ &= \mathbb{E}_{\pi_i}\left[\nabla_{\theta_i} \log \prod_{j=1}^{n} \pi_{\theta_j}(a|s)\cdot Q_i(\mathbf{s}, a_1, \ldots, a_N)\right] \\ &= \mathbb{E}_{\pi}\left[\nabla_{\theta_i} \log \pi_{\theta}(\mathbf{a}|s)\cdot Q_i(\mathbf{s}, \mathbf{a})\right] \end{aligned}$$

Thus the gradient update reduces to the update in the single agent case [17] and is guaranteed to converge to a local maximum of the return $J(\theta_i)$. $\square$

The above algorithm assumes that every agent has access to all other agent's policies to be able to update the critic network. This assumption can be met in these three situations: a centralized training environment, the existence of a communication network between agents where agents can request from other agents their actions for a given state or that each agent models all other agents policies either perfectly via a common broadcast of action, reward and state from all agents or observations.

These assumptions are perhaps too strong and can be computationally infeasible especially when n is large:

- A centralized environment establishes a distinction between training and execution that may not be realistic, it is also not robust to failures while training.
- Communication between agents to obtain predicted actions causes a major slow-down in training, need $n \times S$ messages for each agent and are also prone to failures.
- Modeling other agents creates a heavy computational burden on each agent of training $2 \times n$ networks.

Proposed Approach:

The main issue with MAAC is the need for each agent to model all other agents policies in order to have a decentralized algorithm. When interactions between agents are either sparse agent-wise or sparse between certain groups of agents, learning the joint action-value function $Q_i(s, a_1, \cdots, a_n)$ wouldn't offer as much as an advantage versus learning a single or group action values. By sparse interactions we mean that actions of other agents don't affect either each agent's observations or reward and hence actions. This can happen when the underlying environments is large which would imply that the n agents may have to dispersed across the environment and thus not interacting with each other, when n is large or when the agents have different non competing tasks.

We propose to have each agent model only a subset of size k of other agents who influence his actions and reward the most. Moreover, across time-steps this subset k is due to change and thus the approach adapts to changing influence from all agents. This means that each agent i will maintain an action-value function $Q_i(s, a_i, a_{j_1}, \cdots, a_{j_k})$ where $\{j_1, \cdots, j_k\} \in \{1, \cdots, n\} - \{i\}$

Concretely, agent i maintains an importance array Imp_i of size $n-1$ where $Imp[j]$ represents how much agent j affects agent i reward and actions. We first initialize Imp_i with a heuristic which could be the distance between agent i and agent j. The algorithm starts by selecting the top k indices in Imp as agents to model. After a certain number of time steps to be determined later, the agent now has to decide again which are the top k agents to model.

For all other agents besides the modeled k the importance will only be a function of the heuristic and the old value. However for the currently modeled k agents we can get an estimate value for their importance and influence:

1. **Influence in terms of reward**: we sample m instances from the replay buffer $\{a_i, a_{j_1}, \cdots, a_{j_k}, s\}_1^m$, and compute $Imp_i[j]$ as:

$$Imp_i[j] = \frac{1}{m}\sum_{k=1}^{m} |(Q_i(s, a_i^k, a_j^k, a_{-j}) - \frac{1}{|A|}\sum_{a \in A} Q_i(s, a_i^k, a, a_{-j})\ |^2 \tag{2}$$

   What this expression is computing is that in the m experiences, how much did agent's j chosen action influence the expected reward versus any action agent j could have taken i.e. the average. We would think that if whatever action j took didn't matter then he doesn't influence i, but it could also be the case that all of agent's j actions have similar powerful influence. The time complexity of this approach is $O(k \times |A| \times m)$

2. **Influence in terms of action:** here we try to see if agent j and agent i policies are independent. If they are then it is clear that j has no influence on i. To check for independence we now have to also learn the joint policy between i and the k agents: $\pi(a_i, a_j|s)\ \forall j \in top\ k$. If agent i and j are independent then:$\pi(a_i, a_j|s) = \pi(a_j|s) \times \pi(a_j|s)\ \forall a_i, a_j \in A\ and\ s \in S$. Thus we sample m instances average difference between the previous two quantities:

$$Imp_i[j] = \frac{1}{m}\sum_{k=1}^{m}(\pi(a_i, a_j|s) - \pi(a_j|s) \times \pi(a_j|s))^2 \tag{3}$$

   While this approach is computationally cheaper to check $O(m)$, it requires learning k further policy networks with inputs of size $|A|^2$ each and k critic networks if both are not combined. We also have to make an additive assumption on the reward of agents i and j, in particular our targets become:

$$y_{i\&j} = (r_i \pm r_j) + \gamma Q_{i\&j}(s, a_i, a_j, a_{-j}) \tag{4}$$

Now that all agents influence are computed, we now allow ourself to only swap one of our current top k agents for someone outside. We make a random choice with probability τ for a random agent or with probability $1 - \tau$ for the k'^{th} most influential agent. The randomness here comes from our imperfect knowledge of the influence, we have to balance exploitation and exploration also in terms of agents to model.

In what we have discussed so far, we have assumed to have access to at least the k agents policies to update our Q-values and that at the time of agent swapping the Q-values have approximately converged. It is difficult to determine whether the Q-network has converged since it might not even converge in some cases, it is possible to devise tests to check whether the Q-values are accurate. One possible test is to check for a certain interval of time-steps whether the difference between the Q-value at the end and start of the interval is equal to the sum of reward accumulated during the interval. That is when:

$$Q_i(s_{end}, a_{end}) - Q_i(s_{start}, a_{start}) \approx \sum_{i=start}^{end} r_i$$

This measure is problematic and we refrain from checking for convergence but simply allow the algorithm a determined number of steps as a parameter. When approximately modeling other agent's policies we also have to check for accuracy before computing the importance. One way is to predict over a fixed number of steps other agents actions and check against learned policy.

Extension to Partially observable domains

In a partially observable world, the true state s is no longer known for the agents. Each agent relies on his observations o_i, thus now our modular Q-value takes the form of: $Q(o_i, o_{j_1}, \cdots, o_{j_k}, a_i, a_{j_1}, \cdots, a_{j_k})$ and our policy becomes $\pi_i(a_i|o_i)$ leading to bigger critic and actor networks. Our former treatment follows naturally with these two new definitions. One thing to note is that in partially observable domains, networks with memory such as long short term memory (LSTM) neural networks usually perform better than feed-forward networks [12].

Improving Sample Efficiency

We present a list of approaches proposed in the RL literature to improve sample efficiency and reduce training time for common algorithms.

As presented earlier, experience replay is an efficient way to decrease training time, we replay episodes from previous time-steps. A list of papers focus on prioritized experience replay and other novel methods such as dual-Q learning, distributional RL and multi-step rewards to improve sample efficiency and run-time [13].

A similar notion to this is presented in asynchronous advantage actor-critic (A3C) [26], where the approach is to asynchronously execute multiple agents in parallel, on multiple instances of the environment. The idea is that different agents will explore different parts of the environment and each parallel agent experience will be uncorrelated with the others. Agents gradient updates $d\theta$ are accumulated and are then asynchronously used to update the global network parameter θ.

Other methods focus on changing the optimization algorithm instead of using stochastic gradient descent. One of the problems is with setting the step-size, as the distribution over rewards is changing over time, it is hard to choose a certain step-size or step-size decay schedule. Trust Region Policy Optimization (TRPO) [40] presents a practical optimization algorithm to handle this issue. Our previous approximation of θ is accurate locally thus we can only trust our policy in a local region around it. Consider our objective function where θ_{old} is the parameter before the update, thus equivalently we can write the PG update with the trust region constraint:

$$\max_{\theta} \mathbb{E}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{Q}_t]$$

$$subject\ to\ \mathbb{E}_t[KL[\pi_{\theta_old}(.|s_t), \pi_\theta(.|s_t)] \leq \delta$$

Or using Lagrange multipliers, we can turn the constraint into a penalty for some β:

$$\max_{\theta} \mathbb{E}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t - \beta KL[\pi_{\theta_old}(.|s_t), \pi_\theta(.|s_t)]]$$

Instead of solving the constrained optimization problem, Proximal policy optimization (PPO) [41] proposes to use regular gradient descent algorithms with the KL penalty, however now the problem is with choosing β. PPO suggests to vary it dynamically according to the size of the KL penalty: if KL is large, reduce β, if it small, increase β.

Another approach is to use transfer learning, however little work has explored transfer learning in MARL.

[4] is the first work that addresses transfer in multi-agent learning. The contribution is a novel method: Bias Transfer and an extension of Q-value reuse to multi-agent setting.

A simple extension of single agent methods is problematic, the authors restrict the treatment to tasks with only homogeneous agents that are cooperative. The first step is to map knowledge between similar states and actions between source and target tasks. In the multi-agent setting, the knowledge is distributed among agents, thus the need for a mapping from the joint actions of an n-agent task to an m-agent task. Also a similar mapping between states can be defined, but in the MA setting states may include the agent parameters.

The two approaches for mapping are: static agent mapping a $1-1$ constant mapping between agents, but there are also usually multiple such mapping that will lead to different behaviors. The second isa dynamic or context agent mapping that removes the constant restriction, the specific mapping will be context related e.g. in a grid world map to closest agent in source task. This knowledge is incorporated in the learning as a bias in the initial action value function.

We now present our modular decentralized multi agent actor critic algorithm:

Algorithm 2: Modular Decentralized Multi Agent Actor Critic (MDMAAC)

```
Initialize policy neural network $\pi_{\theta_i}(a|s)$, input is observation and output is probability distribution
over action space
Initialize critic neural network $Q_{w_i}(s,a)$, input is action and observations and output is Q-value.
Initialize replay buffer $\mathcal{D}$ with capacity $m$
Set $\epsilon$, $\tau$ and $\alpha$
Receive initial state $\mathbf{s}$
Initialize $Imp_i$ with freely available heuristic e.g. distance to agents
Initialize $topk_i$ sets of size $k$
for $t = 1$ to max-episode-length (while not terminal state) do
  Update $Imp_i$ for those in $\overline{topk_i}$, as $Imp_i[j] = \alpha Imp_i[j] + (1-\alpha)dist(i,j)$
  if $t$ mod $\frac{1}{1-\alpha} = 0$ then
    Update $Imp_i$ for $topk_i$ according to (2) or (3)
    With $Pr = 1-\tau$ swap in $k$'th highest $Imp_i$ score, for lowest in $topk_i$, and with $Pr = \tau$
    randomly from $n\ agents$
    Reset replay buffer $D$
  end if
  Sample action $a_i$ with $Pr = 1-\epsilon$ from $\pi_{\theta_i}(a|s)$ and with $Pr = \epsilon$ from $A$
  Execute action $a_i$ and observe reward $r_i$ and new state $\mathbf{s}'$ or $\mathbf{o}'$
  Broadcast tuple $(i, a_i, r_i, s)$ through communication network
  Store $(\mathbf{s}, \{a_i, a_{j_1}, \cdots, a_{j_k}\}, \{r_i, r_{j_1}, \cdots, r_{j_k}\}, \mathbf{s}')$ in $\mathcal{D}$
  Sample a random minibatch of $S$ samples $\{(\mathbf{s}^c, \{a_i^c, a_{j_1}^c, \cdots, a_{j_k}^c\}, \{r_i, r_{j_1}, \cdots, r_{j_k}\}, \mathbf{s}'^c)\}_{c=0}^{S}$
  from $\mathcal{D}$
  for agents $l \in topk_i$ do
    Critic:
    Compute $y_l^c = r_l^c + \gamma\, Q_l(\mathbf{s}'^c, a_l', a_i')$, model other agents as simple networks
    Update critic by minimizing the loss $\mathcal{L}(w_i) = \frac{1}{S}\sum_j \left(y_i^c - Q_i(\mathbf{s}^c, a_l^c, a_i^c)\right)^2$
    Actor:
    Update actor network in the direction of:
        $$\nabla \log \pi_{\theta_l}(a|s) \times (Q_l(\mathbf{s}^c, a_j^c, a_i^c) - V_l(s))$$
  end for
  Self Critic:
  Compute $y_i^c = r_i^c + \gamma\, Q_l(\mathbf{s}'^c, a_i^c, a_{j_1}^c, \cdots, a_{j_k}^c)$, model other agents as simple networks
  Update critic by minimizing the loss $\mathcal{L}(w_i) = \frac{1}{S}\sum_j \left(y_i^c - Q_i(\mathbf{s}^c, a_i^c, a_{j_1}^c, \cdots, a_{j_k}^c)\right)^2$
  Self Actor:
  Update actor network in the direction of:
    $$\nabla\left(\log \pi_{\theta_l}(a_i^c|s) \times (Q_l(\mathbf{s}^c, a_i^c, a_{j_1}^c, \cdots, a_{j_k}^c) - V_l(s)) - \beta KL[\pi_{\theta_{t-1}}(.|s^c), \pi_{\theta_l}(.|s^c)]\right)$$
end for
```

Multi-Task Multi-Agent RL

In situations when data for simulation is scarce or when sample efficiency is also a constraint, learning multiple related tasks jointly can lead to an improved performance. Consider m tasks $T_1, \cdots, T_m$ each formulated as a stochastic game (n, S, A, T, R, γ). We assume the agents communication mode is modeled by an undirected graph G with the set of vertices $V = (1, \cdots, n)$ and an edge set E.

In single-agent MT learning, we assume that all tasks can be represented as a linear combination of a smaller subset of tasks. With n agents we assume that for each agent i $\exists$ $k^i < m$ latent basis tasks such that all tasks can be represented as linear combination of these basis tasks for each agent.

Thus now, for agent i in task t his parameters can be written as $\theta_t^i = L^i \cdot s_t^i$ where $L^1, \cdots, L^n$ are common knowledge tasks. However, each agent state representation in a given task will share some elements with the other agents thus without constraining L^i's to be close to each other we will not allow the agents to share information in each task. Therefore, we add a penalty term constraining the L's to be close to each other with respect to the frobenius norm.

Our goal remains to learn optimal policies $\Pi^n = ((\pi_1^1, \cdots, \pi_1^n), \cdots, (\pi_m^1, \cdots, \pi_m^n))$, while encouraging the agent's state representation to be sparse to encourage sharing of information between tasks. Thus one formulation of an objective function for MT MARL is:

$$\min_{L^1,\cdots,L^n} \sum_{t=1}^{m} \sum_{i=1}^{n} -J_t^i(L_i \cdot s_t^i) + \mu||s_t^i||_1 + \lambda||L^i||_F + \beta \left| L^i - \frac{1}{n}\sum_{j=1}^{n} L^j \right|_F$$

The optimization takes place over the graph G, we can make simplify the dependence on all trajectories by approximating with the second order Taylor expansion similar to [1].

A similar objective has been explored in [37], however the terminology of multi-agent in the mentioned paper refers to workers in a distributed system rather than agents in a stochastic game.

Additionally, [30] take a different approach to solving MT MARL. They first solve single task MARL for all tasks and then use policy distillation to combine each task network to produce one policy that is able to act in all tasks equally well.

Evaluation Environments

Many benchmarks have been adopted in the literature to evaluate and compare RL algorithms. We describe some of the benchmarks that can be used to evaluate our proposed approach.

RL

One of the most popular reinforcement learning toolkits is OpenAI Gym that contains multiple benchmark problems. It encompasses multiple episodic environments that include:

- Classic control: classic tasks from RL literature such as cart-pole, mountain car and more
- Atari: classic Atari games, with screen images or RAM as input

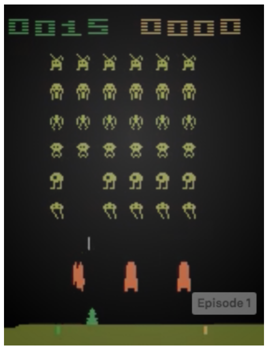

Figure 1: Space Invaders Atari game

- MuJoCo: continuous control tasks, such as making a 3D humanoid stand up, walk and more.
- Robotics: goal based tasks for Fetch and ShadowHand robots

Transfer Learning

OpenAI has recently launched a new contest for transfer learning in RL [29]. Each contestant is given a training set of different levels from the Sonic The Hedgehog$^{\text{TM}}$ series of games and is then evaluated on a test set of custom levels. The goal is to be able to generalize with few-show learning on new tasks. The Sonic games have a discrete set of actions, the reward is the horizontal distance from the initial start position where the player has to travel through the map jumping, swinging to get to the end level.

Multi-Agent RL

Pommerman is a multi agent game similar to Bomberman, the famous game from Nintendo. One each game are four agents, divided into two groups, each group must defeat its opponents by laying bombs on the grid. Each agent can move on the grid and communicate with his teammate and receives as observation the board, its position, its enemies, bombs in the map and more.

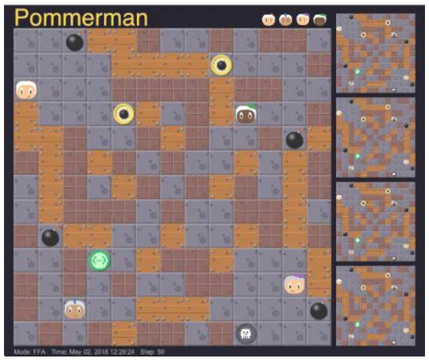

Figure 2: Pommerman visualization

The game is figured as a contest in the $NIPS$ 2018 Competitions track. First place winner has a cash prize of $ 4k USD and an Nvidia GPU, the contest submission deadline is November 26.

The Multi-Agent Particle Environment [24] is a simple multi-agent particle world with a continuous observation and discrete action space where agents engage in competitive and cooperative tasks.

In *Cooperative navigation*, n agents are rewarded based on how far any agent is from each of n landmarks. Agents are penalized if they collide with other agents. Agents have to cover all landmarks while avoiding collision.

In *Predator-prey* environment, good agents try to avoid being hit by adversaries. Adversaries are slower than the good agents, in addition obstacles block the way.

Finally in *covert communication* Alice must send a private message to Bob over a public channel where Alice and Bob are rewarded based on how well Bob reconstructs the message. However, an adversary Eve is present, thus Alice and Bob are negatively rewarded if Eve can reconstruct the message. Alice and Bob have a private key which they must learn to use to encrypt the message.

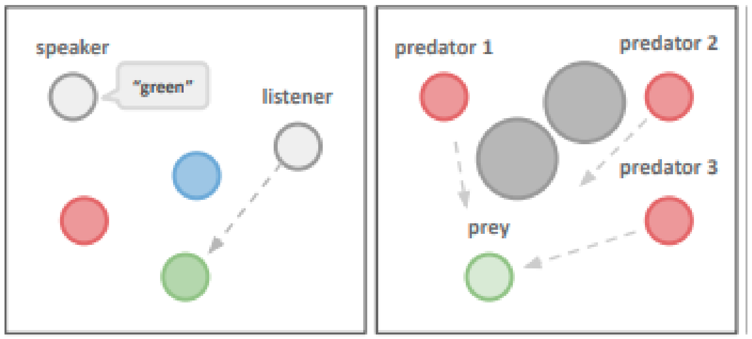

Figure 3: Left: Cooperative Communication, Right: Predator-prey.

Finally, in the StarCraft II Learning Environment [58] the objective of the game is to beat an opponent by shooting,however the player must also carry out and balance a number of sub-goals, such as gathering resources or building structures. The game is also split into mini-games of those sub tasks which are each used as benchmarks.

Figure 4: Starcraft II mini-games

Open-Problems and Future Work

We list some interesting open-problems in RL as mentioned by OpenAI in [31] which target some of the open-gaps in the literature related to our discussion:

- Multi-Task RL with continuous actions: train a single neural network that can simultaneously solve a collection of MuJoCo environments in OpenAI Gym
- Better sample efficiency for TRPO: modify TRPO implementation so that it would converge on all of Gym's MuJoCo environments using at least 3x less experience
- Parameter Averaging in Distributed RL: explore the effect of parameter averaging schemes on sample complexity and amount of communication in RL algorithms
- Regularization in Reinforcement Learning: Experimentally investigate the effect of different regularization methods on an RL algorithm

Now for our proposed approaches, we would first like to experimentally evaluate MDMAAC on the multi-agent particle environment versus IRL and MADDPG. Second, we want to explore situations in which communication between agents is unreliable or unfeasible as in a communication radius constraint to see how will MDMAAC adapt to these situations. Finally, we want to formulate MT MARL as a distributed optimization problem where the workers are now the agents.

Acknowledgments

We thank Yara Rizk for her numerous insightful comments and discussions. We also would like to thank Julia El Zini for discussions on multi-task learning. Finally, we thank Mariette Awad for her guidance in choosing the research topics.

References

[1] H. B. Ammar, E. Eaton, P. Ruvolo, and M. Taylor. Online multi-task learning for policy gradient methods. In *International Conference on Machine Learning*, pages 1206–1214, 2014.

[2] J. Andreas, D. Klein, and S. Levine. Modular multitask reinforcement learning with policy sketches. In *International Conference on Machine Learning*, pages 166–175, 2017.

[3] A. Barreto, W. Dabney, R. Munos, J. J. Hunt, T. Schaul, H. P. van Hasselt, and D. Silver. Successor features for transfer in reinforcement learning. In *Advances in neural information processing systems*, pages 4055–4065, 2017.

[4] G. Boutsioukis, I. Partalas, and I. Vlahavas. Transfer learning in multi-agent reinforcement learning domains. In *European Workshop on Reinforcement Learning*, pages 249–260. Springer, 2011.

[5] M. Bowling and M. Veloso. Rational and convergent learning in stochastic games. In *International joint conference on artificial intelligence*, volume 17, pages 1021–1026. Lawrence Erlbaum Associates Ltd, 2001.

[6] C. Dann, N. Jiang, A. Krishnamurthy, A. Agarwal, J. Langford, and R. E. Schapire. On polynomial time PAC reinforcement learning with rich observations. *CoRR*, abs/1803.00606, 2018.

[7] R. Evans and J. Gao. Deepmind ai reduces google data centre cooling bill by 40%. *DeepMind blog*, 20, 2016.

[8] C. Fernando, D. Banarse, C. Blundell, Y. Zwols, D. Ha, A. A. Rusu, A. Pritzel, and D. Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017.

[9] J. Foerster, G. Farquhar, T. Afouras, N. Nardelli, and S. Whiteson. Counterfactual multi-agent policy gradients. *arXiv preprint arXiv:1705.08926*, 2017.

[10] J. Foerster, N. Nardelli, G. Farquhar, P. Torr, P. Kohli, S. Whiteson, et al. Stabilising experience replay for deep multi-agent reinforcement learning. *arXiv preprint arXiv:1702.08887*, 2017.

[11] J. K. Gupta, M. Egorov, and M. Kochenderfer. Cooperative multi-agent control using deep reinforcement learning. In *International Conference on Autonomous Agents and Multiagent Systems*, pages 66–83. Springer, 2017.

[12] M. Hausknecht and P. Stone. Deep recurrent q-learning for partially observable mdps. *CoRR, abs/1507.06527*, 2015.

[13] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*, 2017.

[14] J. Hu and M. P. Wellman. Nash q-learning for general-sum stochastic games. *Journal of machine learning research*, 4(Nov):1039–1069, 2003.

[15] G. Joshi and G. Chowdhary. Cross-domain transfer in reinforcement learning using target apprentice. *arXiv preprint arXiv:1801.06920*, 2018.

[16] J. Kober, J. A. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[17] V. R. Konda and J. N. Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[19] A. Kumar and H. Daumé III. Learning task grouping and overlap in multi-task learning. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, pages 1723–1730. Omnipress, 2012.

[20] R. Laroche and M. Barlier. Transfer reinforcement learning with shared dynamics. In *AAAI*, pages 2147–2153, 2017.

[21] Y. Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[22] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[23] M. L. Littman. Friend-or-foe q-learning in general-sum games. In *ICML*, volume 1, pages 322–328, 2001.

[24] R. Lowe, Y. Wu, A. Tamar, J. Harb, O. P. Abbeel, and I. Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems*, pages 6382–6393, 2017.

[25] C. Ma, J. Wen, and Y. Bengio. Universal successor representations for transfer reinforcement learning, 2018.

[26] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[28] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[29] A. Nichol, V. Pfau, C. Hesse, O. Klimov, and J. Schulman. Gotta learn fast: A new benchmark for generalization in rl. *arXiv preprint arXiv:1804.03720*, 2018.

[30] S. Omidshafiei, J. Pazis, C. Amato, J. P. How, and J. Vian. Deep decentralized multi-task multi-agent reinforcement learning under partial observability. In *International Conference on Machine Learning*, pages 2681–2690, 2017.

[31] OpenAI. Requests for research. *OpenAI blog*, 2018.

[32] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[33] E. Parisotto, J. L. Ba, and R. Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*, 2015.

[34] P. Peng, Q. Yuan, Y. Wen, Y. Yang, Z. Tang, H. Long, and J. Wang. Multiagent bidirectionally-coordinated nets for learning to play starcraft combat games. *arXiv preprint arXiv:1703.10069*, 2017.

[35] J. Rajendran, A. Lakshminarayanan, M. M. Khapra, P. Prasanna, and B. Ravindran. Attend, adapt and transfer: Attentive deep architecture for adaptive transfer from multiple sources in the same domain. 2016.

[36] Y. Rizk, M. Awad, and E. W. Tunstel. Decision making in multi-agent systems: A survey. *IEEE Transactions on Cognitive and Developmental Systems*, pages 1–1, 2018.

[37] M. Rostami, S. Kolouri, K. Kim, and E. Eaton. Multi-agent distributed lifelong learning for collective knowledge acquisition. *arXiv preprint arXiv:1709.05412*, 2017.

[38] A. A. Rusu, S. G. Colmenarejo, C. Gulcehre, G. Desjardins, J. Kirkpatrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, and R. Hadsell. Policy distillation. *arXiv preprint arXiv:1511.06295*, 2015.

[39] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

[40] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[41] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[42] H. M. Schwartz. *Multi-agent machine learning: A reinforcement approach*. John Wiley & Sons, 2014.

[43] S. Shalev-Shwartz, S. Shammah, and A. Shashua. Safe, multi-agent, reinforcement learning for autonomous driving. *arXiv preprint arXiv:1610.03295*, 2016.

[44] Y. Shoham and K. Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008.

[45] S. M. Shortreed, E. Laber, D. J. Lizotte, T. S. Stroup, J. Pineau, and S. A. Murphy. Informing sequential clinical decision-making through reinforcement learning: an empirical study. *Machine learning*, 84(1-2):109–136, 2011.

[46] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[47] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *ICML*, 2014.

[48] P. Sunehag, G. Lever, A. Gruslys, W. M. Czarnecki, V. Zambaldi, M. Jaderberg, M. Lanctot, N. Sonnerat, J. Z. Leibo, K. Tuyls, et al. Value-decomposition networks for cooperative multi-agent learning. *arXiv preprint arXiv:1706.05296*, 2017.

[49] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[50] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[51] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[52] M. Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337, 1993.

[53] M. E. Taylor and P. Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(Jul):1633–1685, 2009.

[54] Y. Teh, V. Bapst, W. M. Czarnecki, J. Quan, J. Kirkpatrick, R. Hadsell, N. Heess, and R. Pascanu. Distral: Robust multitask reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 4499–4509, 2017.

[55] G. Tesauro. Extending q-learning to general adaptive multi-agent systems. In *Advances in neural information processing systems*, pages 871–878, 2004.

[56] L. Tesfatsion. Agent-based computational economics: modeling economies as complex adaptive systems. *Information Sciences*, 149(4):262–268, 2003.

[57] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning.

[58] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, et al. Starcraft ii: a new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[59] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas. Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224*, 2016.

[60] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[61] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pages 5–32. Springer, 1992.

[62] Z. Yang, K. Merrick, H. Abbass, and L. Jin. Multi-task deep reinforcement learning for continuous action control. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 3301–3307, 2017.

[63] J. Zhang, Z. Ghahramani, and Y. Yang. Flexible latent variable models for multi-task learning. *Machine Learning*, 73(3):221–242, 2008.

[64] K. Zhang, Z. Yang, H. Liu, T. Zhang, and T. Başar. Fully decentralized multi-agent reinforcement learning with networked agents. *arXiv preprint arXiv:1802.08757*, 2018.