Reinforcement Learning in Two Player Simultaneous Action Games

Patrick Phillips [1], **Jing Bi**[2], **Jing Shi**[2], **Chenliang Xu**[2]
[1]Author, [2]Supervisors
250 Hutchison Rd, Rochester, NY 14620
Correspondence to pphill10@u.rochester.edu

Abstract

Two player simultaneous action games are common both in video games and in life. We first introduce the fundamental concepts of reinforcement learning in this setting and discuss the challenges this type of multiplayer game poses. Then we introduce two novel agents that attempt to handle these challenges by using joint action Deep Q-Networks (DQN). The first agent, called the Best Response AgenT (BRAT), builds an explicit model of its opponents policy using imitation learning, and then uses this model to find the best response to exploit the opponents strategy. The second agent, Meta-Nash DQN, builds an implicit model of its opponent's policy in order to produce a context variable that is used as part of the Q-value calculation. An explicit minimax over Q-values is used to find actions close to Nash equilibrium. We find empirically that both agents converge to Nash equilibrium in a self-play setting for simple matrix games, while also performing well in games with larger state and action spaces. These novel algorithms are evaluated against vanilla RL algorithms as well as recent state of the art multi-agent and two agent algorithms. This work combines ideas from traditional reinforcement learning, game theory, and meta learning.

Introduction

An important class of games to consider in reinforcement learning is two player simultaneous action games. Common examples include Rock-Paper-Scissors, Pong, predator-prey games, iterated matrix games, and many real life scenarios such as business competition or war. Furthermore, as we show later, many team games in which agents on two teams share the same objective can also be reformulated into this setting. These types of games include sports such as soccer and basketball as well as video games such as DOTA or Overwatch.

Two player simultaneous action games, and more generally multi-agent games, pose some unique challenges. The most immediate and fundamental challenge is that from the view of any particular agent, the environment is non-stationary. This is known as the moving-target problem [Hernandez-Leal, Kartal, and Taylor 2019; Al-Shedivat et al. 2018; Bansal et al. 2018]. Another difficulty is that of credit assignment; how can any particular agent know if it was their good actions or just their opponents mistakes that lead to the reward they received? A more universal challenge in reinforcement learning and machine learning in general is the curse of dimensionality. This difficulty is often exacerbated by multi-agent games since more agents tend to lead towards larger state and action spaces. One final challenge is that of policy robustness. It is difficult to ensure that any particular action or policy will work well regardless of the other agent's behavior [Hernandez-Leal, Kartal, and Taylor 2019]. In some games it is quite often the case that a particular action will only be good against a particular opponent. Agents can stick to playing a Nash Equilibrium strategy, however this can often lead to overcautious behavior that has little chance of doing better than breaking even. Particularly in two player games, where much more effort can be put into modelling one's opponent, finding and acting on opponent's weaknesses is often viable and effective. Learning robust policies, while also exploiting the weaknesses of opponents is a key motivation for the algorithms we present.

MDP Formalization

A single agent Markov Decision Process (MDP) is defined by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$, where $\mathcal{S}$ represents some finite set of states and $\mathcal{A}$ represents a finite set of actions. The transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ gives the probability of the transition from state $s \in \mathcal{S}$ on action $a \in \mathcal{A}$ to next state $s' \in \mathcal{S}$. The reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathrm{R}$ gives the (possibly stochastic) reward that an agent receives from being in state s taking action a and ending in state s'. Finally $\gamma \in [0, 1]$ is a discount factor that indicates how much less important future rewards are than immediate rewards.

MDPs can be simply extended to multi-agent MDPs that are defined as the tuple $\langle \mathcal{S}, \mathcal{N}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$. Here the key distinctions are that now the action space, rewards, and transition function are given for the joint set of agents, formally this means $\mathcal{A} = \mathcal{A}_1 \times \mathcal{A}_2 \times ... \times \mathcal{A}_N$, $\mathcal{R} = \mathcal{R}_1 \times \mathcal{R}_2 \times ... \times \mathcal{R}_N$, and $\mathcal{T} = \mathcal{S} \times \mathcal{A}_1 \times \mathcal{A}_2 \times ... \times \mathcal{A}_N$.

We consider a particular type of multi-agent MDPs; two player MDPs where $\mathcal{N} = 2$ and $\mathcal{R}_1 = -\mathcal{R}_2$ (the game is zero sum). Note that certain games with more agents can be reformulated as a two player zero sum MDP if there are always only two distinct rewards $\mathcal{R}_1$ and $\mathcal{R}_2$ received by all the agents and $\mathcal{R}_1 = -\mathcal{R}_2$. This includes games such as multi-agent soccer, DOTA, and more. The reformulation can

be achieved by setting the action space for the first agent to be the joint action $\mathcal{A}'_1 = \mathcal{A}_1 \times \mathcal{A}_2 \times ... \times \mathcal{A}_N$ for all agents who receive rewards $\mathcal{R}_1$ and similarly setting the joint action $\mathcal{A}'_2 = \mathcal{A}_1 \times \mathcal{A}_2 \times ... \times \mathcal{A}_N$ for agents who receive the rewards $\mathcal{R}_2$. Sometimes handling this class of multi-agent MDPs with joint actions can be beneficial, as it helps to better coordinate a team's actions, however sometimes the joint action space of size $|A|^{\mathcal{N}}$ is far too large to consider as it grows exponentially with the number of agents.

Relevant Game Theory

The first key notion from Game Theory that we rely heavily on is that of Nash Equilibrium. A set of policies $\boldsymbol{\pi}(s) = \{\pi_i(s)\}_{i \in N}$ form a Nash Equilibrium if unilateral deviation from this equilibrium by a single agent cannot improve the value of that agent's sum of discounted rewards. The sum of discounted rewards for agent i is defined as

$$\mathcal{R}_i\left(s; \pi_i, \boldsymbol{\pi}^*_{-i}\right) = \mathrm{E}\left[\sum_{t=0}^{\infty} \gamma^t r_i(s_t, \pi_i(s_t), \boldsymbol{\pi}_{-i}(s_t)\right] \quad (1)$$

Now formally, a collection of policies $\boldsymbol{\pi}(s)$ forms a Nash equilibrium if

$$\mathcal{R}_i\left(s; \pi_i, \boldsymbol{\pi}^*_{-i}\right) \leq \mathcal{R}_i\left(s; \pi^*_i, \boldsymbol{\pi}^*_{-i}\right) \quad (2)$$

for all states s and admissible policies π_i and for all $i \in N$. In Eq. 2 we use π^* to denote that the policies are that of a Nash Equilibrium, and the notation π_{-i} to denote the set of policies for all agents excluding the i-th agent.

One particular type of game that is of significant importance to subsequent sections are matrix games, and more specifically zero sum matrix games. A matrix game is defined uniquely by a payoff matrix $\mathbf{A}$ such as

$\mathbf{A} =$

		Player 2		
		A	B	C
Player 1	A	(x, y)	(x, y)	(x, y)
	B	(x, y)	(x, y)	(x, y)
	C	(x, y)	(x, y)	(x, y)

In the matrix game Player 1 chooses the a row, and Player 2 chooses a column. They then receive rewards according to the entry corresponding to the choice of row and column, where Player 1 receives reward x and Player 2 receives y. The game is zero sum if $x = -y$ for all entries, and in this case the game can be specified by only a single entry at each location of the matrix. Each player is allowed to use a mixed strategy which is a probability distribution over available actions. The extension to of matrix games to multiple agents is straightforward, however unnecessary for this work.

John Nash proved that any game with a finite number of players each allowed to use a mixed strategy over a finite set of actions has at least one Nash Equilibrium [Nash 1950]. However, it has since been proved that finding if there exists a second equilibrium point is NP-Complete [Daskalakis, Goldberg, and Papadimitriou 2009]. There are many algorithms to find the Nash Equilibria of matrix games, however they all take exponential time in the worst case. One particularly useful algorithm for our purposes however is the Lemke-Howson algorithm which efficiently finds *one* equilibrium point (it also takes worst case exponential time, but practically runs much faster than this) [Lemke and Howson]. We use the Lemke-Howson algorithm to compute Nash Equilibrium of matrix games throughout this work. For a simple example of Nash Equilibrium, consider the Prisoner's Dilemma where each player can either cooperate (Co) or defect (Def).

		Player 2	
		Co	Def
Player 1	Co	$(-1, -1)$	$(-3, 0)$
	Def	$(0, -3)$	$(-2, -2)$

Consider the two strategies in which each player plays *Def*; if either agent were to unilaterally deviate from this strategy they would be worse off. Thus this is the Nash Equilibrium.

Iterated matrix games, which are used as the environment for some of our experiments, consist of repeatedly playing a matrix game. The state in these iterated matrix games is encoded as just the most recent set of actions. In multi-stage games such as iterated matrix games, there are many more complex notions of equilibrium points that go beyond Nash Equilibrium since unilateral deviation is now a bit of an unreasonable basis. The most fundamental equilibrium point, called the Subgame perfect Nash Equilibrium, describes a set of strategies in multi-stage games that are Nash Equilibrium strategies in every subgame, where a subgame is any smaller part of the multi-stage game [Harsanyi, Selten et al. 1988]. For games with only a single Nash equilibrium point such as the Prisoner's Dilemma, the Subgame perfect Nash Equilibrium is again to always defect, even though this makes both players much worse off than cooperating, especially for games with many iterations. However, since the games we consider are zero-sum, we do not have to contend with handling counter-intuitive equilibria like this one.

Deep Q-Network (DQN)

The explosion of reinforcement learning has been largely inspired by the 2015 paper by Mnih et. al. that introduced Deep Q-Networks (DQN) [Mnih et al. 2015]. DQN learns to simply minimize the loss function

$$L_i(\theta_i) = \mathrm{E}_{(s,a,s',r)\sim D}\left[\left(Q(s, a; \theta_i) - r + \gamma \max_{a'} Q(s', a', \theta_i^-)\right)\right] \quad (3)$$

where θ_i denotes the parameters of the Q-Network at iteration i, and θ_i^- denotes the parameters of a target Q-network that is periodically updated to the current Q-Network. This loss function, sometimes called the TD(0) loss, is a one step bootstrap that was first inspired by Bellman's steady state equation for the Q-function. The term

$$r + \gamma \max_{a'} Q(s', a', \theta_i^-) \quad (4)$$

is sometimes referred to as the target or TD-target, since this is the target that we are chasing with our updates to parameters θ. During optimization of the loss function, minibatches

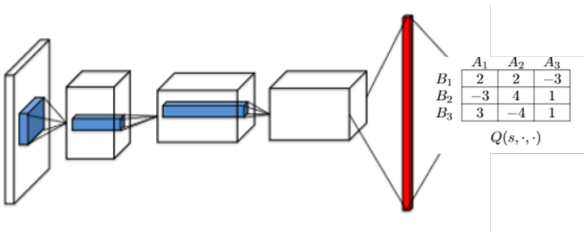

Figure 1: Minimax-DQN architecture

of experience which consist of transitions (s, a, s', r) are drawn from the replay buffer D. The use of this replay buffer from which random samples are drawn stands in contrast to prior work in which updates to the Q function were done online (in sequential order of experience). Using a replay buffer to decorrelate transitions, and also using a target network to compute targets to avoid a non-stationarity problem were two pivotal contributions of Mnih et. al.'s DQN.

Minimax-DQN

A Minimax Q-Learning algorithm was proposed very early by Littman in his 1994 work for two player zero sum games. This algorithm explicitly calculates min-max values of matrix games and does exact tabular updates [Littman 1994]. Recent work has extended Littman's algorithm to make use of function approximation similarly to DQN [Fan et al. 2020] in an algorithm called Minimax-DQN. The Minimax-DQN algorithm is the building block of our RL agents.

The key insight that Littman had in his 1994 work was that at any particular state $s \in \mathcal{S}$ the Q-function $Q(s, \cdot, \cdot)$ can be thought of as a matrix game where the entries represent the values of being in state s, and having the first agent select a row action and the second agent a column action. Since the first agent is trying to maximize his rewards, and the second agent is trying to minimize the first agent's rewards (maximize his own rewards), we can take the Nash Equilibrium value of $Q(s, \cdot, \cdot)$ with a min-max calculation, and then use this value to find the TD-target. This yields the intuitive extension of DQN algorithms to simply adapt the TD(0) loss by replacing target values of Eq. 4 with

$$y_i = r_t + \gamma \cdot \max_a \min_b E\left[Q_\theta^-(s_{t+1}, a, b)\right]. \tag{5}$$

While Littman originally proposed this TD-target for tabular values, it is also useful when using function approximators like neural networks. More specifically, a Deep-Q-Network typically takes in the state and outputs the value for taking any given action from that state. Thus to use this minimax-target, we use a Minimax-DQN that outputs a matrix of values as shown in Figure 1. The loss is then given analogously to Equation 3, by simply substituting in the TD-target given in Equation 4. Algorithm 1 shows the full pseudocode for Minimax-DQN utilizing these target values.

Best Response AgenT (BRAT)

The first agent we introduce is the Best Response AgenT BRAT. BRAT uses Minimax-DQN updates to learn a Q-function while simultaneously building a model of its opponent's policy using imitation learning. Ideally a few shot or one shot imitation learning algorithm would be used, however for simplicity we implement the imitation learning using a simple classification NN that is trained on (state, action) pairs from its opponent.

This naive form of imitation learning, often called behavioral cloning, suffers from compounding error caused by covariate shift [Ross and Bagnell 2010]. Essentially, once the imitation learner makes one mistake, it will now be working with state spaces that come from outside the distribution of data it trained on. For the small state spaces that we run experiments on behavioral cloning proves sufficient.

Pseudocode for BRAT is given in Algorithm 2. The first difference between the Minimax-DQN algorithm and BRAT is that we train against a particular opponent instead of in self-play. We thus construct a classification model of this particular opponent's policy, π_ϕ^{opp}, and periodically update it using log-loss. We use π_ϕ^{opp} to exploit our opponent by choosing actions greedily as

$$a_t = \operatorname*{argmax}_{a_t} \ \pi_\phi^{opp}(s_t) Q_\theta(s_t, a_t, \cdot). \tag{6}$$

Here the policy $\pi_\phi^{opp}(s_t)$ is a vector of probabilities for each action given by our classification model, and we multiply by whichever column of the Q value matrix $Q_\theta(s_t, \cdot, \cdot)$ that will yield the highest expected value. Figure 2 shows an example of this vector-matrix product for some 3-action game.

In Algorithm 2, the last notable difference is that the TD-

Algorithm 1 Minimax-DQN

Input: A two player zero-sum Markov game $(\mathcal{S}, \mathcal{A}, \mathcal{B}, P, R, \gamma)$, replay buffer $\mathcal{D}$, minibatch size n, set of opponent policies π_i, and architecture of minimax deep Q-network $Q_\theta : \mathcal{S} \to \mathcal{A} \times \mathcal{B}$.
Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value network Q_θ with random weights θ
Initialize target network Q_θ^- = Q_θ
while not done **do**
  for $k = 1, K$ **do**
    Set $a_t, b_t = \operatorname*{argmax}_{b_t} \operatorname*{argmin}_{a_t} Q_\theta(s_t, a_t, b_t)$
    Execute actions a_t and b_t and observe reward r_t and state s_{t+1}
    Store transition $(s_t, a_t, b_t, r_t, s_{t+1})$ in $\mathcal{D}_i$
  end for
  Sample minibatch of transitions $(s_t, a_t, b_t, r_t, s_{t+1})_{i \in [n]}$ from $\mathcal{D}_i$
  Set target
  $$y_i = r_{t,i} + \gamma \cdot \max_a \min_b \left[Q_\theta^-(s_{t+1,i}, a, b)\right]$$
  using target network Q_θ^-. Then optimize parameters using loss
  $$L(\theta) = \frac{1}{n} \sum_{i \in |n|} \left(y_i - Q_\theta(s_{t,i}, a_{t,i}, b_{t,i})\right)^2$$
end while

Algorithm 2 Best Response AgenT (BRAT)

Input: A two player zero-sum Markov game $(\mathcal{S}, \mathcal{A}, \mathcal{B}, P, R, \gamma)$, replay buffer $\mathcal{D}$, minibatch size n, opponent policy π^{opp}, and architecture of minimax deep Q-network $Q_\theta : \mathcal{S} \to \mathcal{A} \times \mathcal{B}$.
Initialize opponent policy model to π_ϕ^{opp}
Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value network Q_θ with random weights θ
Initialize target network $Q_\theta^- = Q_\theta$
while not done **do**
  for $k = 1, K$ **do**
    Select actions
      $a_t = \arg\max_{a_t} \ \pi_\phi^{opp}(s_t) Q_\theta(s_t, a_t, \cdot)$
      $b_t = \pi^{opp}(s_t)$
    Execute actions a_t and b_t and observe r_t and s_{t+1}
    Store transition $(s_t, a_t, b_t, r_t, s_{t+1})$ in $\mathcal{D}_i$
  end for
  Sample minibatch of transitions $(s_t, a_t, b_t, r_t, s_{t+1})_{i \in [n]}$ from $\mathcal{D}_i$
  Set the target
  $y_i = r_{t,i} + \gamma \cdot \max_a \left[\pi_\phi^{opp}(s_{t+1,i}) Q_\theta^-(s_{t+1,i}, a, \cdot) \right]$
  OR
  $y_i = r_{t,i} + \gamma \cdot \max_a \min_b \left[Q_\theta^-(s_{t+1,i}, a, b) \right]$
  using target network Q_θ^-. Then optimize parameters with loss:
    $L(\theta) = \frac{1}{n} \sum_{i \in |n|} \left(y_i - Q_\theta(s_{t,i}, a_{t,i}, b_{t,i}) \right)^2$
  Update ϕ using log-loss for (state, action) pairs:
  $L(\phi) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{M} \mathrm{I}(a_j = a_{t,i}) \log \pi_\phi^{opp}(s_{t,i})$
end while

B_1	B_2	B_3
1/3	2/3	0

$\pi_{opp}(s)$

Player 2

Player 1	A_1	A_2	A_3
B_1	2	2	-3
B_2	-3	4	1
B_3	3	-4	1

$Q(s, \cdot, \cdot)$

Figure 2: Policy times $Q(s, \cdot, \cdot)$ value matrix in 3-action game.

target is set as either

$$y_i = r_{t,i} + \gamma \cdot max_a \left[\pi_\phi^{opp}(s_{t+1,i}) Q_\theta^-(s_{t+1,i}, a, \cdot) \right]$$
$$OR$$
$$y_i = r_{t,i} + \gamma \cdot \max_a \min_b \left[Q_\theta^-(s_{t+1,i}, a, b) \right].$$

The first option is more greedy; we assume the value of being in subsequent state s_{t+1} is dependent on our model of our opponent's policy, and act accordingly. The second option assumes that our opponent will act optimally from the subsequent state. We found empirically that the second option is generally more effective, although computing the min-max value of the game can be much more computationally expensive, especially for large action spaces.

Meta-Nash DQN

The second agent we introduce builds an *implicit* model of the opponent using a GRU to create a context variable C which is in turn used to compute matrix values $Q(C, s, \cdot, \cdot)$. To select actions, this agent directly computes a Nash-Equilibrium policy for the particular output $Q(s, \cdot, \cdot)$.

The implicit model of the opponent's policy is constructed using (state, action) pairs, similar to BRAT. During execution, the pairs (S_{t-1}, a_{t-1}) are continually fed into the GRU to produce a context variable C_{t-1}. This context variable is fed into the DQN, along with the current state, to produce matrix values $Q(C_{t-1}, s, \cdot, \cdot)$. The Nash Equilibrium set of policies are explicitly calculated with the Lemke-Howson algorithm, and an action is sampled from the mixed Nash Equilibrium policy for the Meta-Nash Agent. The contiguous trajectories are stored in the replay buffer along with the context variables C and hidden states h which are needed to learn the parameters of the GRU.

During training, random chunks of the trajectories are sampled of length equal to a *GRU-length* hyperparameter, and fed into the GRU along with the correct hidden state obtained from the replay buffer to produce context C_{t-1}. TD-targets are computed using the min-max Nash Equilibrium values similar to Minimax-DQN, but now also using the context variable from the replay buffer. The TD-error is both a function of the parameters ϕ of the GRU and the parameters θ of the DQN:

$$L(\theta, \phi) = \mathrm{E}_{(s,a,b,s',r) \sim D} \Bigg[\left(Q_\theta(C^\phi, s, a, b) - r + \gamma \max_{a'} \min_{b'} Q_\theta(C^\phi, s', a', b') \right)^2 \Bigg].$$

The full pseudocode for the Meta-Nash Agent is given in Algorithm 3.

This algorithm is called ***Meta***-Nash DQN because there is a nice parallel between opponent modelling using a GRU and the work of Meta-Q-Learning [Fakoor et al. 2020]. If we consider each opponent we train against as a distinct task, then Fakoor et. al.'s Meta-Q-Learning would feed in transitions (s, a, s', r) into a GRU to produce a context variable which in turn is used to produce Q-values. Unlike Meta-Q-Learning, we only need the context variable to contain information about our opponent's policy, and not the reward or transition functions and thus we only feed in (state, action) pairs into the GRU. Furthermore, we want to be able to model our opposing agent as a nonstationary entity instead of part of the environment, and therefore handle a matrix of Q-values which is of course different than Fakoor et. al's Meta-Q-Learning.

Related Work

Literature on multi-agent reinforcement learning (MARL) and competitive games has recently surged in popularity. Early attempts approached multi-agent problems with single

Algorithm 3 Meta Learning Nash-DQN

Input: A two player zero-sum Markov game $(\mathcal{S}, \mathcal{A}, \mathcal{B}, P, R, \gamma)$, replay buffer $\mathcal{D}$, minibatch size n, set of opponent policies π_i, and architecture of minimax deep Q-network $Q_\theta : \mathcal{S} \times \mathcal{A} \times \mathcal{B} \to \mathcal{R}$.
Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value network Q_θ with random weights θ
Initialize target network $Q^*_\theta = Q_\theta$
Initialize context GRU with parameters ϕ and set *GRU-Length*
while not done **do**
  Sample batch of policies π_i
  for each π_i **do**
    for $k = 1, K$ **do**
      Select opponent action $a_t \sim \pi_i$
      Select own action
        $b_t \sim \max \min Q_\theta(C_{t-1}, S_t, a_t, b_t)$
      Execute actions a_t, b_t and observe R_t, S_{t+1}
      Feed transition through GRU
        $h_t, C_t = GRU_\phi(h_{t-1}, (S_t, a_t))$
      Append transition
      $(h_{t-1}, C_t, S_t, a_t, b_t, R_t, S_{t+1})$
      To replay buffer $\mathcal{D}_i$
    end for
    Sample minibatch of transitions from $\mathcal{D}_i$
    Compute adapted paramters by setting the target
      $y_i = R_t + \gamma \cdot \max_a \min_b [Q^*_\theta(C_{t,i}, S_{t+1,i}, a, b)]$
    and then updating parameters
      $\theta, \phi \leftarrow \arg\min \sum_{i \in [n]} (y_i - Q_\theta(C_{t-1,i}, S_{t,i}, a_{t,i}, b_{t,i}))$
  end for
end while

agent learners that treat other agents as part of the environment. However, this has been found not to work well in practice, likely due to the nonstationarity of each agent's environment [Matignon, Laurent, and Le Fort-Piat 2012]. There has been extensive work focusing explicitly on two player games in strategic turn based setting such as Go, Chess, and Backgammon [Tesauro 1994; Silver et al. 2018]. There has been much less focus on the two player simultaneous action games that we address.

An important algorithm in the history of MARL developed by Lowe et. al. is Multi-Agent Deep Deterministic Policy Gradients (MADDPG) [Lowe et al. 2017]. MADDPG uses a centralized Q-value $Q^{\boldsymbol{\pi}}_i(s, a_1, ..., a_n)$ function for each agent i which takes as input the state and actions of each agent, and outputs the Q-value for agent i. Lowe et al. derive an actor critic method similar to DDPG to learn a policy from this value function in continuous action spaces. However, to update the Q-value function requires the policy of other agents. Lowe et al. suggest using maximum likelihood with entropy regularization to model other agents' policies when they are unknown.

Recently Li. et al. proposed an extension to MADDPG called Minimax Multi-Agent Deep Deterministic Policy Gradients (M3DDPG). This algorithm adds the assumption that all other agents are acting adversarially, and thus takes a minimum over all opponent actions in Q-value updates. However, they note that finding the min over all other agents actions is computationally intractable, especially as they work with continuous action spaces. Thus they use a linear approximation of the Q-function and take one gradient step towards the action that minimizes the Q-value. This algorithm is a nice extension of the Minimax-DQN algorithm from [Fan et al. 2020] to continuous action spaces with multiple agents. However, like the Minimax-DQN algorithm, M3DDPG forces the agent to act conservatively with no way to exploit a suboptimal opponent.

There have also been many efforts in MARL that use a central value network (either V(s) or Q(s, a)) which can be decomposed into individual value functions for each agent. However, these methods have only been used in cooperative settings [Sunehag et al. 2018; Rashid et al. 2018]. One of the main motivations of such an approach in multi-agent scenarios is that again the joint action space grows exponentially in the number of agents, and thus planning with a classic central value network is intractable for many problems. These methods have proved effective in cooperative games such as team searching and fetching and Starcraft unit micromanagement.

Michael Bowling and Manuela Veloso introduced two algorithms for handling two player simultaneous action games using tabular Q-value functions and policies [Bowling and Veloso 2001]. The first simple algorithm called policy hill climbing (PHC) is sort of tabular actor critic method that uses mixed strategies. The second algorithm called Win or Lose Fast PHC (WoLF PHC) adds a variable learning rate to the PHC algorithm which is large while WoLF PHC is losing and small when WoLF PHC is winning, where winning/losing is evaluated by comparing the most recent returns to average returns.

A more recent paper that handles two player simultaneous action games is Learning with Opponent Learning Awareness (LOLA) [Foerster et al. 2018]. LOLA is a policy gradient algorithm that uses the parameters of their opponent to make policy updates based on a forecast of their opponent's learning. To do so, Foerster et. al assume that they have access to their opponents policy parameters, which the authors admit is a unrealistic assumption. They develop a weaker version of LOLA which does not make this assumption, and instead uses a maximum likelihood estimate to infer their opponents policy, similar to MADDPG.

Experiment Methodology

We consider tests in two different environments: (1) a simple iterated matrix game of Matching Pennies, and (2) a predator-prey game. The first environment consists of repeatedly playing the matrix game shown in Figure 3. The second environment takes place in a gridworld as depicted in Figure 4, where some agents are predators, and other are prey. The predators get +1 reward whenever they catch a prey, and the prey get -1 reward whenever they are caught. The discount factor γ is important in this setting so that predators are incentivized to quickly catch the prey, and the prey are incentivized to survive for as long as possible.

		Player 2 (P2)	
		H	T
Player 1 (P1)	H	$1, -1$	$-1, 1$
	T	$-1, 1$	$1, -1$

Figure 3: Payoff Matrix for Matching Pennies Game

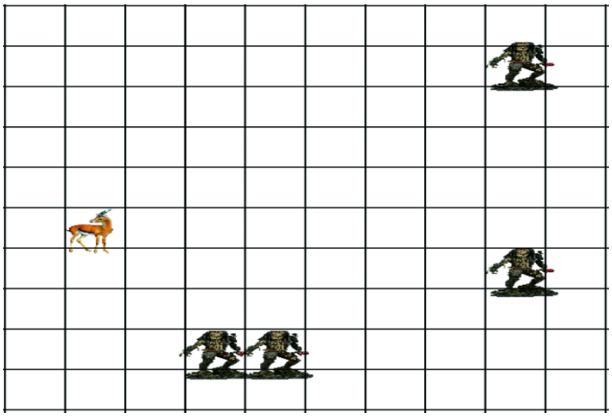

Figure 4: Sample environment for predator-prey game

In both environments we set up a tournament that consists of multiple agents competing in a round-robin setting. We consider six agents; the Best Response AgenT (BRAT), the Meta-Learning NashDQN (META-NASH), Win or Lose Fast Policy Hill Climbing (WoLF-PHC), single agent Q-Learning (Q-Learning), single agent Policy Gradient (PG), and Learning with Opponent Learning Awareness (LOLA). During the round robin tournament, all pairs of agents play some number of matches/episodes against eachother. For the Iterated Matching Pennies environment, the matches/episodes consist of 100 transitions. A total of 50 episodes are run consecutively. This is done for 25 different random trials (different parameter initialization).

For the predator-prey game we use a similar tournament setup, however the number of transitions per episode generally does not reach the maximum of 100 transitions, as the prey is caught sooner. Since the episodes are typically shorter, we run for a total of 200 different episodes. We experiment with both a 1v1 predator-prey game and a 3v1 predator-prey game where we take the joint action space of the 3 predators to reduce the problem to a two agent MDP.

Before the round robin tournaments there is the option to pretrain the agents. Notice that the algorithms we develop, and in particular the Meta-Nash DQN are designed to train against many different agents in order to be able to quickly adapt to playing a new agent (the simple behavioral cloning of BRAT does not actually use/require this setup, but ideally a few shot imitation learning algorithm that did utilize this pretraining would be used instead of behavioral cloning). Other algorithms typically do not have a pretraining step like this. For the Iterated Matching Pennies tournament we do not pretrain, but for the predator-prey environment we do pretrain each agent by allowing all pairs of agents to play for 50 episodes before results are started to be recorded (only a single set of parameters for each agent are used during this step, while during the tournament each agent has a different set of parameters for each match).

Results

The average rewards for the Iterated Matching Pennies tournament are plotted in Figure 5. On the y-axis the reward is plotted averaged over the 6 different games it is playing, one with each opposing agent, as well as averaged over 10 different random parameter initializations. The x-axis is just the episode number. Notice that the META-NASH algorithm consistently receives rewards near 0 for the average reward in each episode. This is likely due to the explicit minimax calculation over Q-values which forces the agent to play closely to Nash Equilibrium. On the other hand, the BRAT agent slowly improves performance, which we speculate is due to the fact that the BRAT agent is gradually building a more accurate model of its opponent.

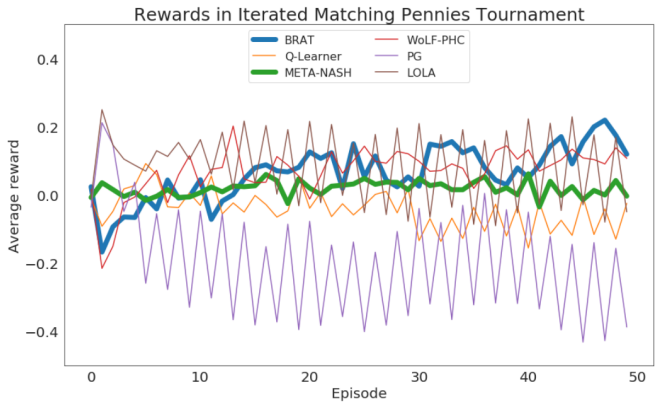

Figure 5: Graph for rewards in IMP

In Figure 6, we again plot the average rewards for the IMP tournament, this time as a bar graph of the cumulative reward over all 50 episodes.

The average rewards for the 1v1 predator-prey tournament are plotted in Figure 7. Again on the y-axis we are plotting the average reward per episode which is averaged over the 6 different games the agent is playing versus all the other agents as well as over the 25 different random initializations.

The average rewards for the 3v1 predator-prey tournament are plotted in Figure 8 in the same manner.

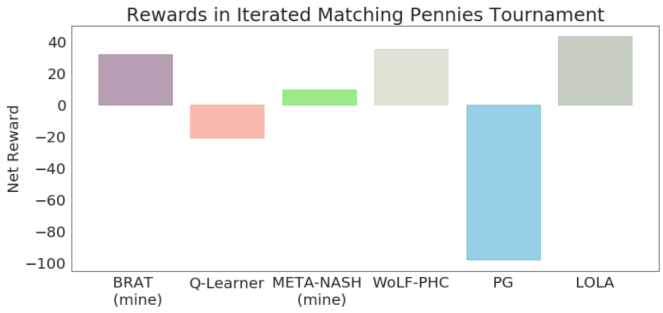

Figure 6: Bar graph for cumulative rewards in IMP

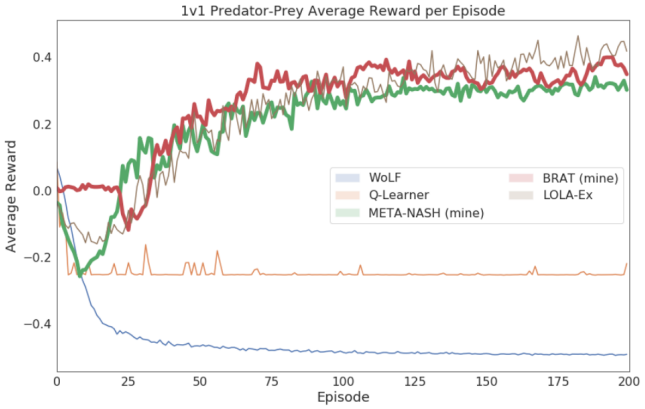

Figure 7: Results from 1v1 predator-prey tournament

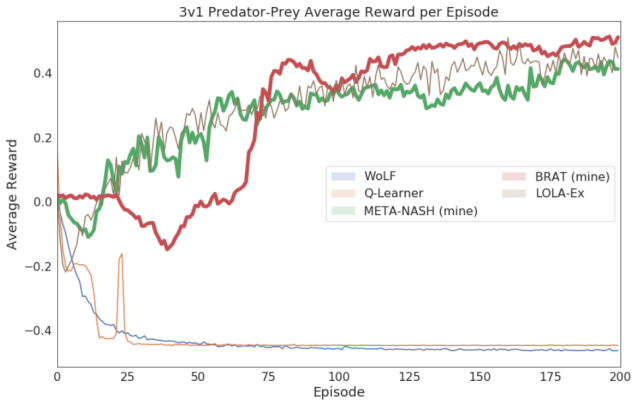

Figure 8: Results from 3v1 predator-prey tournament

We found empirically that the algorithms we developed did not scale well to games with larger state and action spaces such as the soccer environment Gfootball.

Acknowledgements

Thanks to Ph.D. students Jing Bi and Jing Shi for helping supervise my project. Thanks to the NSF REU grant for funding my research. Lastly, thanks to Professor Xu for asking questions and pushing and supporting my research.

References

Al-Shedivat, M.; Bansal, T.; Burda, Y.; Sutskever, I.; Mordatch, I.; and Abbeel, P. 2018. Continuous Adaptation via Meta-Learning in Nonstationary and Competitive Environments. *arXiv:1710.03641 [cs]* URL http://arxiv.org/abs/1710.03641. ArXiv: 1710.03641.

Bansal, T.; Pachocki, J.; Sidor, S.; Sutskever, I.; and Mordatch, I. 2018. Emergent Complexity via Multi-Agent Competition. *arXiv:1710.03748 [cs]* URL http://arxiv.org/abs/1710.03748. ArXiv: 1710.03748.

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

Daskalakis, C.; Goldberg, P. W.; and Papadimitriou, C. H. 2009. The complexity of computing a Nash equilibrium. *SIAM Journal on Computing* 39(1): 195–259.

Fakoor, R.; Chaudhari, P.; Soatto, S.; and Smola, A. J. 2020. Meta-Q-Learning. *arXiv:1910.00125 [cs, stat]* URL http://arxiv.org/abs/1910.00125. ArXiv: 1910.00125.

Fan, J.; Wang, Z.; Xie, Y.; and Yang, Z. 2020. A Theoretical Analysis of Deep Q-Learning. *arXiv:1901.00137 [cs, math, stat]* URL http://arxiv.org/abs/1901.00137. ArXiv: 1901.00137.

Foerster, J. N.; Chen, R. Y.; Al-Shedivat, M.; Whiteson, S.; Abbeel, P.; and Mordatch, I. 2018. Learning with Opponent-Learning Awareness. *arXiv:1709.04326 [cs]* URL http://arxiv.org/abs/1709.04326. ArXiv: 1709.04326.

Harsanyi, J. C.; Selten, R.; et al. 1988. A general theory of equilibrium selection in games. *MIT Press Books* 1.

Hernandez-Leal, P.; Kartal, B.; and Taylor, M. E. 2019. A Survey and Critique of Multiagent Deep Reinforcement Learning. *Autonomous Agents and Multi-Agent Systems* 33(6): 750–797. ISSN 1387-2532, 1573-7454. doi:10.1007/s10458-019-09421-1. URL http://arxiv.org/abs/1810.05587. ArXiv: 1810.05587.

Lemke, C. E.; and Howson, J. T. ???? EQUILIBRIUM POINTS OF BIMATRIX GAMES 11.

Littman, M. L. 1994. Markov games as a framework for multi-agent reinforcement learning. In *Machine Learning Proceedings 1994*, 157–163. Elsevier. ISBN 978-1-55860-335-6. doi:10.1016/B978-1-55860-335-6.50027-1. URL https://linkinghub.elsevier.com/retrieve/pii/B9781558603356500271.

Lowe, R.; Wu, Y. I.; Tamar, A.; Harb, J.; Abbeel, O. P.; and Mordatch, I. 2017. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in neural information processing systems*, 6379–6390.

Matignon, L.; Laurent, G. J.; and Le Fort-Piat, N. 2012. Independent reinforcement learners in cooperative Markov games: a survey regarding coordination problems. .

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.;

Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540): 529–533. ISSN 0028-0836, 1476-4687. doi:10.1038/nature14236. URL http://www.nature.com/articles/nature14236.

Nash, J. F. 1950. Equilibrium Points in n-Person Games. *Proceedings of the National Academy of Sciences of the United States of America* 36(1): 48–49. URL http://www.jstor.org/stable/88031.

Rashid, T.; Samvelyan, M.; de Witt, C. S.; Farquhar, G.; Foerster, J.; and Whiteson, S. 2018. QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning. *arXiv:1803.11485 [cs, stat]* URL http://arxiv.org/abs/1803.11485. ArXiv: 1803.11485.

Ross, S.; and Bagnell, D. 2010. Efficient reductions for imitation learning. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 661–668.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362(6419): 1140–1144.

Sunehag, P.; Lever, G.; Gruslys, A.; Czarnecki, W. M.; Zambaldi, V. F.; Jaderberg, M.; Lanctot, M.; Sonnerat, N.; Leibo, J. Z.; Tuyls, K.; et al. 2018. Value-Decomposition Networks For Cooperative Multi-Agent Learning Based On Team Reward. In *AAMAS*, 2085–2087.

Tesauro, G. 1994. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation* 6(2): 215–219.

Appendix A: Code Breakdown

To run any of the code first use a package manager to install the packages from *requirements.txt*. The other files exposed in the main folder include the code for the META-NASH agent and the BRAT agent, as well as the code to run the tournaments. The *run_tournament* file is currently set up to run the Iterated Matching Pennies (IMP) tournament for 25 randomly initialized trials each consisting of 50 episodes, each episode consisting of 100 transitions. The IMP tournament was run with $\gamma = 1.0$ and using Adam optimizer with $lr = 0.005$. These were the same settings used to generate the results in this paper.

To run tournaments for the predator-prey environment, there is a conflicting dependency for the *gym* package. I believe the only change that has to be made is to replace gym version .17.0 with .10.0, and then the predator-prey environment from the folder *ma-gym* should be available.

To plot the results from the tournaments, look in the folders *matching-pennies_tournament* and *predator-prey_tournament*. In each folder there is data saved in a folder with the environment name, a .ipynb with the code to load the data and plot, and pdf images of the results.

The file *CNN_context_dqn_nash_eq* contains a version of the META-NASH algorithm suitable for environments that have a pixel state space such as the atari environments. No tests of this environment were included in this report.

The *common* folder includes some functionality that is useful to various algorithms implemented such as the replay buffer code.