LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

INSTITUT FÜR INFORMATIK
LEHRSTUHL FÜR DATENBANKSYSTEME
UND DATA MINING

Project Thesis
in Computer Science

Multi-Agent Reinforcement Learning in StartCraftII with the PyMARL-Framework

Patrick Matthäi

Aufgabensteller:	Prof. Dr. Matthias Schubert
Betreuer:	Sabrina Friedl
Abgabedatum:	99.99.9999

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.
This paper was not previously presented to another examination board and has not been published.

München, 99.99.9999

.....
Patrick Matthäi

Abstract

Contents

Chapter 1

Introduction

Chapter 2

Motivation

Most reviewed papers concerning MARL algorithms do not provide detailed experiments with huge groups of agents. This raises the question how agent policies behave in terms of metrics and visible wrong behavior when watching these agents play. Since the joint action as well as the communicated state increase with the number of agents, it is interesting how already developed algorithms perform. PyMARL is a collection of algorithms running on the SMAC environment which allows for multi-agent micro-management learning in StarCraftII. This is a novelty since most algorithms perform on different environments and are therefore not suitable for comparison. PyMARl combines already five algorithms (IQL,VDN,QMIX,COMA,QTRAN) while others have forked PyMARL or SMAC to run experiments (CKL, SMIX). Use of local and global rewards is not always consistent (f.e. some papers seem to use local rewards with algorithm A while others f.e. PyMARL implement algorithm A with global reward)

Chapter 3

Presets

3.1 PyMARL - A multi-agent reinforcement learning framework

In order to setup PyMARL(fork) with extensions on local and combined rewards, the Readme in the following repository `https://github.com/PMatthaei/pymarl` contains a step by step guide. The main repository of PyMARL can be found at `https://github.com/oxwhirl/pymarl`. [6]

3.2 Related work

3.2.1 Q-Learning in single agent environments

Q-Learning is an algorithm to find a suitable policy to maximize an agent's accumulated long term reward without explicit information about dynamics of the environment including the reward scheme. Q-learning estimates the value of an action in a particular state of the environment.[10]

$$Q_{\text{new}}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}}\Big)}^{\text{temporal difference}}$$

3.2.2 DQN - Deep Q-Networks

[4]

3.2.3 IQL - Independent Q-Learning

Each agent deploys an independent Q-learning algorithm to learn an individual action-value function solely based on their own actions.

Recently DQN have been used to estimate the value-function especially in environments with large input [9]. In a multi-agent setup, the taken action is determined by the corresponding DQN for all agents separately, therefore the environment is the only interface for interaction between agents. This results in an easy to setup architecture which is decentralized, stable across tasks and computational efficient. The setup by [9] uses local rewards and the game state (screen capture of the game itself) is fully observable shared between the agents. This results in the update rule for a single agent i:

$$Q^{\mathrm{i}}_{new}\left(s^i_t, a^i_t\right) \leftarrow Q^i\left(s^i_t, a^i_t\right) + \alpha \cdot \left(r^i_t + \gamma \cdot \max_a Q\left(s_{t+1}, a\right) - Q^i\left(s^i_t, a^i_t\right)\right.$$

Although this implementation is straight forward and simple, simultaneous learning and exploring agents create a non-stationarity in the environment. This can lead to non-convergence of policies and inaccurate representation of interactions between agents [5].

The implementation of IQL in PyMARL is similar to the previous setup. The state for this setup is a (partial) copy of the game‘s state as opposed to image data as in above setup.

3.2.4 JQL - Joint Q-Learning

JQL or joint action learning (JAL) only differs from IQL in terms of the provided action. While IQL used just the action of the currently training agent a_i, JQL learns Q-values based on the joint action $\mathbf{a}$ over all agents in the environment or group. In order to deal with the non-stationarity of the environment, the relative value of an individual action can maintain beliefs about policies of other agents.

$$Q^{\mathrm{i}}_{new}\left(s^i_t, \mathbf{a}_t\right) \leftarrow Q^i\left(s^i_t, \mathbf{a}_t\right) + \alpha \cdot \left(r^i_t + \gamma \cdot \max_a Q\left(s_{t+1}, a\right) - Q^i\left(s^i_t, \mathbf{a}_t\right)\right.$$

The estimated value of the individual action a_i performed by agent i can be retrieved via:

$$EV\left(a^i\right) = \sum_{a^{-i} \in A_{-i}} Q\left(a^{-i} \cup \{a^i\}\right) \prod_{j \neq i} \left\{ \mathrm{Pr}^i_{a^{-i}[j]} \right\}$$

[2]

3.2.5 VDN - Value Decomposition Networks

The main idea behind VDN arises from the assumption that joint-action-value functions can be decomposed into d single-agent value functions which are then composed via summation into the join-action-value function.

$$Q((h^1, h^2, ..., h^d), (a^1, a^2, ..., a^d)) \approx \sum_{i=1}^{d} Q^i(h^i, a^i)$$

Thus each agents value-function $\tilde{Q}_i$ depends solely on it's local observation. The history of agent i is denoted as h^i while a^i corresponds to the agent's current action. The history $h_t = a_1 o_1 r_1, ..., a_{t-1} o_{t-1}, r_{t-1}$ of an agent includes it's action, observation and reward at a given timestep up until a the last registered timestep $t-1$.

The agent's value-function is learned via back-propagation of the gradients resulting from applying the Q-learning rule on the joint reward and the local observations. $\tilde{Q}_i$ is therefore independent of specific rewards.

As a result each agent independently performs greedy actions based on it's own $\tilde{Q}_i$. [7]

"However, VDN severely limits the complexity of centralised action-value functions that can be represented and ignores any extra state information available during training. [5]"

$$Q_{tot}(h, \mathbf{a}; \theta) = \sum_{i=1}^{N} \alpha_i Q^i \left(h^i, a^i; \theta^i\right), \alpha_i \geq 0$$

[11]

3.2.6 QMIX - Monotonic Value Function Factorization

QMIX is another value-based method which trains decentralized policies in a centralized end-to-end learning step. Therefore a DQN estimates action-values as complex non-linear combination of per-agent values. The estimation uses local observations instead of a global fully-observed state during execution.

Since the joint-action value is strictly monotonic per agent, off-policy learning can benefit from maximisation of the joint-action value which results in consistency between centralized and decentralized policies. [5]

$$\frac{\partial Q}{\partial Q_i} \geq 0, \forall i$$

- agent networks representing their respective Q_i-value, DRQNs that receive the current individual observation s_t^i and the last action a_{t-1}^i as input at each time step

- mixing network that combines agent values into Q, feed-forward neural network that takes the agent network outputs as input and mixes them monotonically, producing the values of Q
- enforcing monotonicity constraint of by restricting the mixing network to have positive weights
- set of hypernetworks
- result:represent complex centralised action-value functions with a factored representation that scales well in the number of agents and allows decentralised policies to be easily extracted via linear-time individual argmax operations.
- for consistency we only need to ensure that a global argmax performed on Qtot yields the same result as a set of individual argmax operations performed on each Qa :

$\mathbf{a}$ is joint action and $\mathbf{h}$ is joint action-observation history.

$$\underset{\mathbf{a}}{\text{argmax}} Q(\mathbf{h}, \mathbf{a}) = \begin{pmatrix} \text{argmax}_{a^1} Q_1\left(\mathbf{h}^1, a^1\right) \\ \vdots \\ \text{argmax}_{a^n} Q_n\left(\mathbf{h}^n, a^n\right) \end{pmatrix}$$

b is batch size of samples from the replay buffer

$y^{\text{tot}} = r + \gamma \max_{\mathbf{u}'} Q_{\text{tot}}\left(\boldsymbol{\tau}', \mathbf{u}', s'; \theta^-\right)$ and θ^- are targets of the target network.

loss function:

$$\mathcal{L}(\theta) = \sum_{i=1}^{b} \left[\left(y_i^{tot} - Q(\mathbf{h}, \mathbf{a}, s; \theta)\right)^2 \right]$$

3.2.7 SMIX - Enhanced Centralized Value Functions

$$G_t^{\lambda} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

$$G_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^n \mathbb{E}_{\boldsymbol{\pi}} Q\left(\boldsymbol{\tau}_{t+n}, \mathbf{a}_{t+n}; \theta^-\right)$$

θ^- are parameters of the target network.

3.2.8 QTRAN -

3.2.9 CKL - Common Knowledge Learning

3.2.10 COMA - Counterfactual Multi-Agent Policy Gradients

[3]

"At the other extreme, we can learn a fully centralised state-action value function Qtot and then use it to guide the optimisation of decentralised policies in an actor-critic framework, an approach taken by counterfactual multi-agent (COMA) policy gradients (Foerster et al., 2018), as well as work by Gupta et al. (2017). However, this requires on-policy learning, which can be sample-inefficient, and training the fully centralised critic becomes impractical when there are more than a handful of agents. COMA (Foerster et al., 2018) uses a centralised critic to train decentralised actors, estimating a counterfactual advantage function for each agent in order to address multi-agent credit assignment[5]"

3.3 StarCraftII Multi Agent Challenge Environment

PyMARL's multi-agent reinforcement learning algorithms are running in the SMAC environment `https://github.com/oxwhirl/smac`. [6] The following sections are disecting the construction of the reward in SMAC.

3.3.1 Reward function

The StarCraftII Multi Agent Challenge (SMAC) environment supports sparse and dense reward functions. This can be controlled via the configuration parameter `reward_sparse`. Per default sparse rewards are set to `False`, resulting in dense rewards for all agents interacting with the environment. In the following other important configuration parameters regarding the reward function and their default values are listed:

- `reward_death_value: -10`
  Reward for death of a unit.
- `reward_defeat: 0`
  Reward for losing a round.
- `reward_negative_scale: 0.5`
  Scale negative rewards to count just half as much as positive rewards.
- `reward_only_positive: True`
  Do not reward unit death.
- `reward_scale_rate: 20`
  Scale rate to normalize rewards. (20 is the maximum achievable reward)

- `reward_win: 200`
  Reward for winning a round.

The function `reward_battle` instantly returns zero if `reward_sparse` is set to `True` since battle actions are only used for sparse rewarding. Rewards concerning overall win and loss are added during the step implementation of the environment. In the following description of the reward construction only alive units are considered.

The sparse (battle) reward is constituted by the accumulative hit/shield point damage dealt to all enemy units as well as the `reward_death_value` for each killed enemy. In case only positive rewards are configured, allied casualties are taken into account. Hereby the damage dealt to ally units and `reward_death_value` per ally unit killed are scaled by `reward_negative_scale` and afterwards deduced from the reward. Therefore received damage of allies and their death is penalized.

Since the reward is accumulated, the contribution of each unit/agent can not be deduced afterwards. The python code corresponding to the global reward function can be found in the appendix in section A.1.

3.4 Global and local reward functions

The global reward function $R^{global}(s,a) = \frac{1}{n}\sum_{i=1}^{n} R_i\left(s^{(i)}, a^{(i)}\right)$ in a multi-agent setup is defined as a function returning the total reward for a group of n agents. Each agent is provided with this reward as result of his own action contributing to the joint action.

On the other side, a local reward function $R_i^{local}\left(s^{(i)}, a^{(i)}\right)$ should reward an action taken by a single agent i within a group. Accumulating all local rewards of agents in the same group should result in the global reward calculated for this group. [1]

It should be noted that the default PyMARL reward function is not dividing it‘s global reward by the amount of agents contributing to it. A corrected global reward function is used in the following experiments.

3.4.1 Rewards in Multi-Agent Reinforcement Learning

[1] Shows that most algorithms "will need to see $\tilde{\Omega}(n)$ trajectories in the worst case to achieve good performance". An important definition arising from this paper is the neighborhood of an agent i, which describes the subset of agents that are directly influenced by agent i‘s state.

Whitehead 1991 Tan: n agents observing everything about each other decrease required learning time at rate of omega(1/n)

3.4.1.1 Credit Assignment Problem

In case a group of agents is producing a joint or collective action in an environment, a single reward signal loses information about each agents contribution to the reward. In this scenario, the credit assignment problem is targeting the question which agents should receive more credit for positively contributing to the global reward signal. [8]

For learning algorithms relying on the TD error computed on the global reward - such as VDN - the obtained gradient for each agent does not provide information about how this agents contribution to the reward. With increasing amount of agents, the gradient becomes noisy due to the fact that some agents might be exploring. [**?**]

3.4.2 Challenges in reward shaping

Chapter 4

Dynamic reward composition

4.1 Idea

Since the global reward function can de decomposed into a sum of agent-specific local reward functions we can choose arbitrary mixins of these functions. In other words, there is the possibility to choose which reward signal is predominantly influencing the agents policy. Agents which rely more on the local reward may suffer less from the credit assignment problem but are prone to acting selfishly in situations where team work is necessary.

Agents focusing more on global rewards for their training procedure are denoted as "altruistic" or "selfless" since their actions will be choosen on the intention of increasing profit for their team.

Agents concentrating on local rewards for their training procedure are denoted as "selfish" since their actions will be choosen in a way to increase their own profit.

The agents reward focus can be set via a combination weight $c \in [0, 1]$. A combination weight of $c = 1$ constitutes a totally "self-interested" agent while choosing $c = 0$ observes team performance only.

4.2 Implementing a local reward function

In order to prove the effectiveness of dynamic reward composition, a local reward function is needed in the SMAC environment. Therefore the current global reward function has to be analyzed to identify reasonable decompositions. It should be mentioned that reward shaping is a difficult topic and most algorithms run on simple global reward functions within their environment to prove independence of reward shaping since their target is to solve the problem without encoding the solution completely into the reward function. The im-

plementation of a local reward function in this paper is targeting to be close to the global reward function of the SMAC environment to allow for comparsion of already conducted experiments. Therefore the following decisions have been made.

Each agent receives only his own health difference to the previous step as positive reward. This is very close to the global implementation where all allied units are considered.

During action selection, each agent which executed an attack action on an enemy is registered. Based on this targeting data, the total dealt damage and kill reward in a step is evenly split between all attacking agents. Passive agents which do not attack will not receive reward. Since the environment is deterministic in its attack damage calculation, the same amount of damage can be expected by the same unit in a step. This assumption does not hold with heterogeneous team compositions which is one reason to exclude them from the following experiments.

Natural global rewards such as round loss or win are evenly distributed between all agents.

The python code corresponding to the local reward function can be found in the appendix in section A.2.

4.3 Combining local and global rewards

In a group of n cooperating agents, each agent i chooses an action $A = (a_1, a_2, ..., a_i), i = n$ and is assigned his own reward combination weight $W_{reward} = (w_1, w_2, ..., w_i)$. Since the global reward is composed of the local rewards as depicted in section 3.4, local and global rewards can be linearly mixed. Via the combination weight w_i the combined reward can be shaped to reinforce local or global rewards. Thus the combined reward for an agent i is

$$R_i^{combined}\left(s^{(i)}, a^{(i)}\right) = w_i * R_i\left(s^{(i)}, a^{(i)}\right) + \frac{1}{n} * \sum_{j=1}^{n}(1 - w_j) * R_j\left(s^{(j)}, a^{(j)}\right) \quad (4.1)$$

¸where w_i is the combination weight for the currently observed agent and w_j are weights from group members of agent i. The recombined global reward is therefore:

$$R_{global}^{combined} = \frac{1}{n} * \sum_{j=1}^{n}(1 - w_j) * R_j\left(s^{(j)}, a^{(j)}\right) \quad (4.2)$$

The following experiments have been conducted with a uniform weight for all agents resulting in $w_1 = w_2 = ... = w_i = w$. Therefore $R_{global}^{combined}$ can be rewritten to

$$\begin{aligned} R_{global}^{combined} &= \frac{1}{n} * \sum_{j=1}^{n} (1 - \mathbf{w}) * R_j \left(s^{(j)}, a^{(j)}\right) \\ &= (1 - w) * \frac{1}{n} * \sum_{j=1}^{n} * R_j \left(s^{(j)}, a^{(j)}\right) \\ &= (1 - w) * R^{global}(s, a) \end{aligned} \tag{4.3}$$

Resulting in the final form:

$$R_i^{combined}\left(s^{(i)}, a^{(i)}\right) = w * R_i\left(s^{(i)}, a^{(i)}\right) + (1 - w) * R^{global}(s, a) \tag{4.4}$$

Equation 4.1 is offering non-uniform combination weights in order to contribute to learning of different policies per agent. This could be helpful in heterogeneous agent compositions.

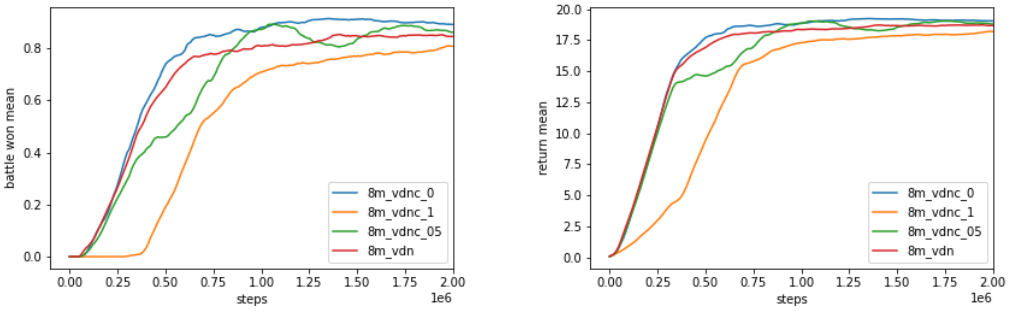
The python code corresponding to the combined reward function (with non-uniform weights) can be found in the appendix in section A.3.

Chapter 5

Evaluation

5.1 Experiments

VDN deployed in the 8m environment is already performing nearly perfectly although the learned policy differs noticebly from QMIX. Setting the combination weight to $c = 0$ results in the best performance, indicating that a global reward signal is influencing a more effective policy and most prominent a faster learning. Agents trained with a combination weight of $c = 1$ reach about a 80 percent win rate, but start learning when other algorithms already converge. Experiments conducted with $c = 0.5$ show a fluctuating learning process, which nonetheless seems to converge to a similar performance.

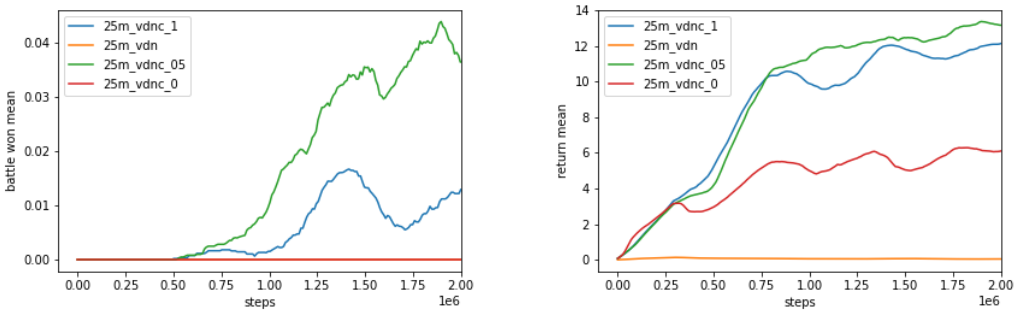


(a) Battle won mean

(b) Return mean

Figure 5.1: VDN on 8 marines vanilla vs. combined rewards c=0.0,0.5,1.0.

As expected, environments with a larger group of agents profit from local rewards and gain even more from combined rewards. Vanilla VDN and combined reward VDN with $c = 0$ are not able to win any rounds. Providing the agents with local rewards only ($c = 1$) allows for at least some wins.



(a) Battle won mean

(b) Return mean

Figure 5.2: VDN on 25 marines vanilla vs. combined rewards c=0.0,0.5,1.0.

All models and experiment metrics can be found at `https://github.com/PMatthaei/pymarl-results`. Further experiment permutations than the ones included in this documentation can be evaluated via the visualization script in Google Colaboratory.

Chapter 6

Conclusion

Problem: Environment → only homogeneous agents and symmetric groups. heterogeneous groups require more selfless behaviour f.e. tanking, exploding, being the bait.

Chapter 7

Future work

The current implementation only used uniform unchangeable combine weights. Depending on the situation a agent might want to vary its combination to encourage a more selfish or more selfless behaviour.

Furthermore each agent should hold its own combination weight since their state may differ from other agents and requires a corresponding reward combination.

The current local reward function does not support heterogeneous team compositions since it is assumed every unit inflicts the same damage per step. This renders our implementation useless for a majority of environments.

Hyperparameter tuning not conducted.

Appendix A

Appendix

A.1 SMAC - global reward function

```
def global_reward(self):
  if self.reward_sparse:
    return 0

  reward, delta_deaths, delta_ally, delta_enemy = 0

  neg_scale = self.reward_negative_scale
  # update deaths
  for al_id, al_unit in self.agents.items():
    if not self.death_tracker_ally[al_id]:
    # did not die so far
    prev_health = (self.previous_ally_units[al_id].health      +
                                  self.previous_ally_units[al_id
                                 ].shield)
  if al_unit.health == 0:
    # just died
    self.death_tracker_ally[al_id] = 1
    if not self.reward_only_positive:
      delta_deaths -= self.reward_death_value * neg_scale
      delta_ally += prev_health * neg_scale
    else:
    # still alive
      delta_ally += neg_scale * (prev_health - al_unit.health -
                                  al_unit.shield)

  for e_id, e_unit in self.enemies.items():
    if not self.death_tracker_enemy[e_id]:
      prev_health = (
      self.previous_enemy_units[e_id].health
      + self.previous_enemy_units[e_id].shield
      )
    if e_unit.health == 0:
      self.death_tracker_enemy[e_id] = 1
      delta_deaths += self.reward_death_value
      delta_enemy += prev_health
    else:
      delta_enemy += prev_health - e_unit.health - e_unit.
                                  shield
  # shield regeneration
  if self.reward_only_positive:
    reward = abs(delta_enemy + delta_deaths)
  else:
    reward = delta_enemy + delta_deaths - delta_ally

return reward
```

A.2 SMAC with combined rewards - local reward function

```
def local_reward(self, a_id, target_id):

  if self.reward_sparse:
    return 0

  neg_scale = self.reward_negative_scale
  delta_death, delta_self, delta_enemy = 0
  agent_unit = self.get_unit_by_id(a_id)
  # If the unit with id a_id is still alive
  if not self.death_tracker_ally[a_id]:
    # Fetch its previous health
    prev_health = self.previous_ally_units[a_id].health + self.
                                previous_ally_units[a_id].
                                shield
    # If it just died
    if agent_unit.health == 0:
    # Reward death negatively if configured
    if not self.reward_only_positive:
      delta_death -= self.reward_death_value * neg_scale
    # Remember lost health (= damage from enemies) to reward
                                negatively later
      delta_self += prev_health * neg_scale
  # If still alive
  else:
    current_health = agent_unit.health + agent_unit.shield
    health_reward = prev_health - current_health
    # Reward the delta health (shield regeneration, heal
                                through MMM)
    delta_self += neg_scale * health_reward

  # Search for the target (if it exists) which a_id attacked
  if target_id is not None:
    e_id, e_unit = next((e_id, _) for e_id, _ in self.enemies.
                                items() if target_id == e_id)
    if not self.death_tracker_enemy[e_id]:
      # Reward dealt attack damage + kill
      delta_enemy += self.local_attack_r_t

  if self.reward_only_positive:
    reward = abs(delta_enemy + delta_death)
  else:
    reward = delta_enemy + delta_death - delta_self

  return reward
```

A.3 Generalized reward combination function

```
def combine_local_rewards(self, rs):
  ws = self.local_reward_weights
  # Pair local rewards with their weight
  rs_ws = list(zip(rs, ws))
  # Calculate weighted global reward
  r_global_combined = np.mean([(1.0 - w_j) * r_j for r_j, w_j
                               in rs_ws])
  # Calculate each agents recombined reward
  rs_combined = [w_i * r_i + r_global_combined for r_i, w_i in
                               rs_ws]

  return rs_combined
```

Bibliography

[1] Drew Bagnell and Andrew Y. Ng. On local rewards and scaling distributed reinforcement learning. In Y. Weiss, B. Schölkopf, and J. C. Platt, editors, *Advances in Neural Information Processing Systems 18*, pages 91–98. MIT Press, 2006.

[2] Caroline Claus and Craig Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. *AAAI/IAAI*, 1998(746-752):2, 1998.

[3] Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. *arXiv preprint arXiv:1705.08926*, 2017.

[4] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[5] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder De Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. *arXiv preprint arXiv:1803.11485*, 2018.

[6] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philiph H. S. Torr, Jakob Foerster, and Shimon Whiteson. The StarCraft Multi-Agent Challenge. *CoRR*, abs/1902.04043, 2019.

[7] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z Leibo, Karl Tuyls, et al. Value-decomposition networks for cooperative multi-agent learning. *arXiv preprint arXiv:1706.05296*, 2017.

[8] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[9] Ardi Tampuu, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente. Multiagent cooperation and competition with deep reinforcement learning. *PloS one*, 12(4):e0172395, 2017.

[10] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[11] Chao Wen, Xinghu Yao, Yuhui Wang, and Xiaoyang Tan. Smix (λ): Enhancing centralized value functions for cooperative multi-agent reinforcement learning. In *AAAI*, pages 7301–7308, 2020.