Reinforcement learning algorithms for the Untangling of Braids

Abdullah Khan*, Alexei Lisitsa†, Alexei Vernitski *

* University of Essex, Colchester CO4 3SQ, United Kingdom. *(ak20749,asvern)@essex.ac.uk*

†University of Liverpool, L69 3BX, United Kingdom. *A.Lisitsa@liverpool.ac.uk*

Abstract

We use reinforcement learning algorithms (Q-Learning and Deep Q-Learning) to tackle the problem of untangling braids and to compare the results of both algorithms. The idea is to use multi-agent (two competing players) based approach to tackle the problem of untangling braids. We interface the braid untangling problem with the OpenAI Gym environment, a widely used way of connecting agents to reinforcement learning problems. The results provide evidence that the more we train the system, the better the untangling player gets for both approaches at untangling braids. The comparison of both approaches produces interesting results, where Q- learning performs better while dealing with braids of shorter length, whereas DQN performs slightly better while dealing with braids of longer length.

Introduction

Braids are mathematical objects from low-dimensional topology which can be successfully encoded with sequences of letters and, therefore, studied using algebra or, as we do in this study, using some computer-scientific approach. A braid on n strands consists of n ropes whose left-hand ends are fixed one under another and whose right-hand ends are fixed one under another; you can imagine that the braid is laid out on a table, and the ends of the ropes are attached to the table with nails. Figures 1, 2, 3 show examples of braids on 3 strands.

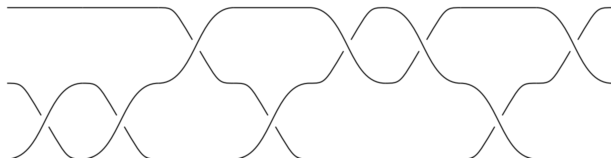

Figure 1: Braid $aabaBBAB$

Few of the interesting application of braids, e.g., in the field of biology they can be used to examine the ability of enzymes to add or remove tangles from DNA; in chemistry, they allow us to describe the structure of topological stereoisomers, or molecules with the same atoms but different configurations; whereas in physics, graphs used in braids to create interesting models for examining the way in which particles interact. In the context of the current study, untangling braids with reinforcement learning is important as a step towards using reinforcement learning to solve a more general mathematical problem known as the *word problem in groups*; making progress in this direction is generally useful for training computers to prove theorems and conduct mathematical research.

Copyright © 2021by the authors. All rights reserved.

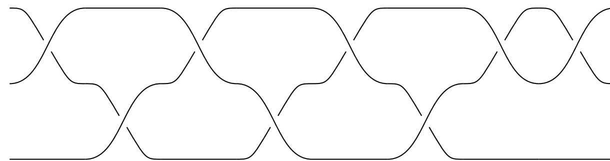

Figure 2: Braid $baBABaBb$

Two braids are *equivalent* to one another if they can be transformed into one another by shifting and twisting the middle parts of the ropes (without touching the ends of the ropes). For example, the two braids in Figures 1, 2 are equivalent to one another, although it is difficult to see it. They are also what is called *trivial* braids, in the sense that they are equivalent to the braid without any intersections of ropes, shown in Figure 3.

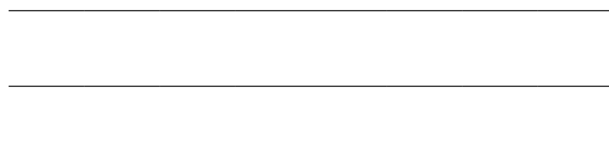

Figure 3: The trivial braid without intersections

Now let us explain how braids can be represented conveniently in the computer. A braid is considered as a sequence of its simple fragments; for braids on 3 strands, these are the fragments shown in Figure 4, which we denote by $A, a, B, b, 1$ (and which in mathematical papers are usually denoted by $\sigma_1, \sigma_1^{-1}, \sigma_2, \sigma_2^{-1}, 1$).

Using this convenient notation, we can now say that the braids in Figures 1, 2 are $aabaBBAB$ and $baBABaBb$. This notation is useful not only for describing braids, but also for checking if two braids are equivalent. Indeed, it is known that two braids are equivalent if and only if one can be transformed to the other using rules called the second

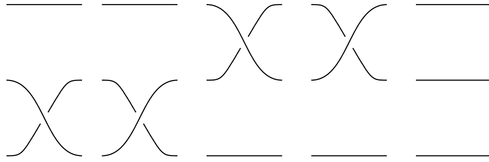

Figure 4: Braid fragments $A, a, B, b, 1$

Reidemeister move and the third Reidemeister move. The *second Reidemeister move* is the rule stating that Aa and aA are equivalent to 11, and Bb and bB are also equivalent to 11. (An algebraist studying braids in the context of group theory would also add that 11 is equivalent to 1; however, we felt that the performance of our AI will be best if we omit this non-essential rule). The *third Reidemeister move* is the rule stating that ABA is equivalent to BAB. A recent study (Gukov et al., 2021) uses RL to untangle knots using a version of Reidemeister moves known as Markov moves. The novelty of our approach is the use two agents: one for tangling and one for untangling.

Recent success of reinforcement learning (RL) in many interesting problems, such as playing the game of Go (Silver et al., 2017), playing card games (Brown and Sandholm, 2019), and autonomous driving (Shalev-Shwartz et al., 2016) mostly involve the participation of more than one single agent/player, and such problems are modeled as multi-agent RL (MARL) problems. MARL addresses the autonomous agents that operate in a common environment, each of which aims to optimize its own long term return by interacting with the environment. Multi-agent systems can be placed in different groups e.g., *cooperative*, *competitive* depending on the types of settings. In particular, in the cooperative setting, agents collaborate to optimize a common long-term return; while in the competitive setting, they have opposite goals with reward of one agent is the loss of the other.

We experimented with braids with three strands using (Q-Learning and Deep Q-Learning), where following transformations are allowed, Aa=aA=11; Bb=bB=11; A1=1A; a1=1a; B1=1B; b1=1b, ABA=BAB. We approach the problem of untangling braids on 3 strands as a game played between two players, player 1 (the tangling player) and player 2 (the untangling player). Player 1 starts with an untangled braid as in Figure 3 with specific length of the input and applies *Reidemeister moves* to tangle the braid. For example, braids in Figures 1, 2 were produced by player 1 after approximately 150 games against player 2. Once player 1 has created a tangled braid after a fixed number of steps, that would be the input for the player 2 (untangling player); the task of player 2 is to apply *Reidemeister moves* to reach a fixed target output, that would be all 1's (untangled state). We use OpenAI Gym (Brockman et al., 2016) an interface which provides a number of environments to implement reinforcement learning problems. The benefit of interfacing with OpenAI Gym is that it is an actively developed interface which allows to add environments and features useful while training the model.

The paper is organized as follows: in the following section we discuss the Background taking into consideration basics towards Q-Learning and Deep Q-Learning, state-of-the-art, setting up the environment to actions and rewards. In **Section 3**, we mention about the experimental details and results.

Background

In this section, we formally highlight the important concepts for the understanding and development of the project, and also highlight some of the relevant work in the domain of reinforcement learning specifically for games.
Reinforcement learning is the training of machine learning models to make a sequence of decisions, where the agent learns to achieve a goal in an uncertain, potentially complex environment(Kaelbling et al., 1996). In RL, there is a game-like situation, where the computer employs trial and error to come up with a solution to the problem. Basically, during the whole learning process, the agent gets either rewards or penalties for the actions it performs. The overall goal is to maximize the total rewards.

Q-Learning

$$\underbrace{\text{New } Q(s,a)}_{\substack{\text{New}\\\text{Q-Value}}} = \underbrace{Q(s,a)}_{\substack{\text{Current}\\\text{Q-Value}}} + \underset{\substack{\text{Learning}\\\text{rate}}}{\alpha} \Big[\underbrace{R(s,a)}_{\text{Reward}} + \underset{\substack{\text{Discount}\\\text{rate}}}{\gamma} \overbrace{\max Q'(s',a')}^{\substack{\text{Maximum predicted reward, given}\\\text{new state and all possible actions}}} - Q(s,a) \Big]$$

As discussed in our previous work (Khan et al., 2021), Q-learning makes use of the Bellman equation. Where the first term, Q(s, a) is the value of the current action in the current state, alpha is the learning rate that controls how much the difference between previous and new Q-value is considered. Gamma is a discount factor, which is used to balance between immediate and future reward. The updates occur after each step or action, and ends when an episode is done (reaching the terminal point). The agent will not learn much after initial episodes, but eventually with enough exploring (steps and episodes) it will converge and learn the optimal Q-values.

Algorithm 1: Q-learning (Sutton and Barto, 2018)

```
Initialize Q(s,a), for all s ∈ S+, a ∈ A(s).
foreach episode do
    Initialize S;
    foreach step of episode do
        Choose A from S using policy derived from
          Q (e.g., -greedy);
        Take action A, observe R, S';
        Q(S,A) ← Q(S,A) + α[R +
          γ max_a Q(S',a) − Q(S,A)];
        S ← S';
        until S is terminal
    end foreach
end foreach
```

The pseudo code mentioned in the **algorithm 1**, the Q-table is initialized to all zeros indicating that the agent

doesn't know anything about the world. Then as the episode begins, the agent performs an action from the given state currently it resides, and observe the next state with respective reward, the agent remains in the new state and repeat the process until a terminal state is reached. Instead of selecting actions based on the maximum future reward it selects an action at random. Acting randomly is important because it allows the agent to explore and discover new states that otherwise may not be selected during the exploitation process.

DQN

A core difference between Deep Q-Learning and Q-Learning is the implementation of the Q-table. DQN replaces the regular Q-table with a neural network. Rather than mapping a state-action pair to a Q-value, a neural network maps input states to (action, Q-value) pairs.

One of the property of DQN is that the learning process uses 2 neural networks. These networks have the same architecture but different weights. After every fixed number of defined steps, the weights from the main network are copied to the target network, resulting towards more stability in the learning process. Both the main and target network maps input states to output actions. These output actions actually represent the model's predicted Q-value. The action that has the largest predicted Q-value is the best known action at that state.

After choosing an action, it's time for the agent to perform the action and update the main and target networks according to the Bellman equation. Deep Q-Learning agents use Experience Replay to learn about their environment and update the main and target networks. The main network weights are then copied to the target network weights every fixed number of defined steps.

Experience Replay is the act of storing and replaying game states (current state, action, reward, next state) that the RL algorithm is able to learn from. The use of Experience replay updates the parameters of the algorithm using saved and stored information from previously taken actions. It learns in small batches to avoid inaccurate dataset distribution of different states, actions, rewards, and next states that the neural network will see. Importantly, the agent doesn't need to train after each step.

From the pseudo code mentioned in **algorithm 2**, we first Initialize main and target neural networks, also an empty replay memory D. The agent selects and executes actions according to an Epsilon Greedy Strategy. The algorithm modifies standard Q-learning to make it suitable for training large neural networks without diverging. It uses experience replay in which we store the agent's experiences at each time-step (current state, action, reward, next state) in a dataset, pooled over many episodes (where the end of an episode occurs when a terminal state is reached) into a replay memory. During the inner loop of the algorithm we apply mini batch updates to sample experiences, drawn at random from the pool of stored samples. DQN performs rollouts in an environment, collects data and then uses this data to train, by performing Stochastic gradient descent on the Mean Squared Error Loss of the predicted Q values for a given state. Slowly, after many iterations of training, both neural networks will approximate the optimal Q values.

Algorithm 2: DQN Algorithm (Mnih et al., 2015)

Initialize replay memory D to capacity N
Initialize action-value function Q
Initialize target action-value function $\tilde{Q}$
while *not converged* **do**
    Choose A from S using policy derived from Q (e.g., -greedy);
    Agent takes action a, observes reward r, and next state s';
    Store transition in the(s,a,r,s',$done$) in the experience replay memory D
    if *enough experiences in D* **then**
        Sample a random $minibatch$ of N transition from D
        for *every transition* (s_i,a_i,r_i,s_i,$done_i$) ***in*** $minibatch$ **do**
            if $done_i$ **then**
                $y_i = r_i$
            else
                $y_i = r_i + \gamma max_{a' \in A} \tilde{Q}(s'_i, a')$
            end if
        end for
        Calculate the loss $\mathcal{L} = 1/N \sum_{i=0}^{N-1} (Q(s_i, a_i) - y_i)^2$
        Update Q using the SGD algorithm by minimising the loss $\mathcal{L}$
        Every C steps copy weights from Q to $\tilde{Q}$
    end if
end while

State-of-the-Art

Currently there is a lot of research going on towards building hybrid solutions (Khan et al., 2019b,a) integrating Deep Learning with rule-based reasoning. In our current research we use rules discussed in Section1 to implement a Reinforcement learning based solutions taking into account Q-Learning and Deep Q-Learning.

Q-learning was first proposed by Watkins in 1989 (Watkins and Dayan, 1992), since then it has become the popular option for reinforcement learning-based agents, however it is less effective for the complicated and high state space problem. Whereas, comparatively the use of DQN is more of a recent phenomena proposed by (Mnih et al., 2015) which is more effective for high state space problems. The extensions of DQN include double DQN (Van Hasselt et al., 2016), dueling DQN (Wang et al., 2016), deep recurrent Q-network (Hausknecht and Stone, 2015).

RL has had extensive success in complex control environments like Atari games(Mnih et al., 2013), Sokoban

planning (Feng et al., 2020). It is also applied to games where there is real time strategy (RTS) such as bots (Wender and Watson, 2014), another reinforcement learning based approach (Amato and Shani, 2010) chooses from a set of predefined strategies in turn based strategy based games. In such approaches the training process is separated into several stages, each of them responsible for different aspects of the game (such as combat, movement and exploration). Other works in strategic fighting games (Graepel et al., 2004) map the possible states of the game based on low-level formations, such as distance between the fighters and health points. The reward function used are simple: a positive reward is granted every time the agent strikes the opponent and a negative reward is given when the agents gets hit. Moreover, Q-learning and DQN have been widely applied to dynamic treatment regimes (DTR) (Chakraborty and Moodie, 2013; Tsiatis et al., 2019), where the goal is to find sequential decision rules for individual patients that adapt to time-evolving illnesses.

A very recent study (Gukov et al., 2021) introduced natural language processing into the study of knot theory, and they also utilize reinforcement learning (RL) based algorithms to find sequences of moves and braid relations that simplify knots and can identify unknots by explicitly giving the sequence of actions. Another study (Grannen et al., 2020) proposed HULK a perception-based system that untangles dense overhand and figure-eight knots in linear deformable objects from RGB observations. It exploits geometry at local and global scales and learns to model only task-specific features, instead of performing full state estimation, to enable fine-grained manipulation.

Environment Settings

To use the Q-Learning and DQN algorithms, it is necessary to setup the environment with which the agent interacts by performing certain actions while being in a specific state. We have used OpenAI Gym to setup the environment which focuses on the episodic setting of reinforcement learning, where the agent's action chains are broken down into a sequence of episodes. Each episode begins by randomly sampling the agent's initial state and continues until the environment reaches a terminal state. We experimented with different settings of the environment. For example, *reward shaping* which is an effective technique for incorporating domain knowledge into reinforcement learning (RL) algorithms. Using reward shaping the system would like to accumulate positive rewards as much as possible, whereas negative rewards encourage to reach a terminal state as quickly as possible to avoid accumulating penalties. The rewards mentioned in the table 1, table 2 are of the type reward shaping. In the second setting, we explored *reward propagation*. In this setting, rewards are assigned at the end of episode, e.g. as the game is played between the two agents, a positive reward is associated to the final move for the agent which wins the game, and a negative reward for the agents final move that loses the game. In the final setting, we experimented with hybrid approach reward shaping and propagation, where agent1 is trained using reward shaping and agent2 using reward propagation.

Action	Reward
CARET_MOVE	0
ROTATE_TRUE	0
ROTATE_FALSE	0
REPLACE_TRUE	1
REPLACE_BACK	-1
REPLACE_FALSE	-1
ROTATE_REPLACE	0

Table 1: Reward associated for each action for agent2

Action	Reward
CARET_MOVE	0
ROTATE_TRUE	0
ROTATE_FALSE	0
REPLACE_TRUE	-1
REPLACE_BACK	1
REPLACE_FALSE	-1
ROTATE_REPLACE	0

Table 2: Reward associated for each action for agent1

In this paper, we present results of the reward shaping, as they were more encouraging as compare to the other two settings we discussed.

Inside the environment for our use-case; we use caret of a fixed length 5 which moves back and forth over the string. The length of the caret represents a sub-string, where the sub-string represents the state of the agents. The caret moves over five characters at a time in the whole string and the five characters inside the string would represent the state of the agent. It is not feasible to consider the whole string, as for each character of the string, there are 5 options: ['a', 'A', 'b', 'B', '1']. This setting would result in large state-space representation, and not feasible for Q-learning problem, as Q-table is constructed for each state-action pair.

Actions and Rewards

The table 1, shows the rewards associated with each action for braids with 3 strands for the untangling agent, whereas, table 2 shows the reward associated for each action of the tangling agent. As we have already discussed all such actions that bring us closer to the target output value will have the higher rewards and all such actions which takes us away from the target output will have lesser rewards.

The following actions does the following, $action_replace$, replaces (Aa to 11, aA to 11, Bb to 11, bB to 11), $action_replace_back$ replaces (11 to Aa, 11 to aA, 11 to Bb, 11 to bB), $action_rotate_replace$ moves the position of the strings (ABA to BAB, BAB to ABA), $action_rotate$ moves the position of the strings (aA to Aa, aA to Aa, bB to Bb, Bb to bB). The choice of the reward selection is inspired from few of the works recently published (Gawłowicz and Zubow, 2018; Mendonça et al., 2015).

Experiments and Results

All the experiments were performed in a system with Nvidia Geforce GTX 1080Ti with 3584 cores running at

1583 MHz frequency on 11GB onboard GDDR5X memory. We have used CUDA version 10.2 to compile the code. The code was implemented in python using gym library. To measure the performance, we utilize the metrics provided by OpenAI Gym interface, namely *rewards over episodes* of a particular environment. In each episode there are two players (player1 = tangling player, player2= un-tangling) the first player starts with a fixed length of the input tries to tangle the braid during the fixed number of defined steps applying the transformations discussed in Section 1, that tangled state is the input for the second player which again applies the same transformations to un-tangle the braid.

To implement the Q-Learning algorithm the idea is to find the optimal action-selection policy using a Q function. Our goal is to maximize the value function Q. The Q-table helps us to find the best action for each state. It helps us to maximize the expected reward by selecting the best of all possible actions. Q(state, action) returns the expected future reward of that action at that state. This function can be estimated using Q-Learning, which iteratively updates Q(s,a) using the Bellman equation. Initially the agents explore the environment and update the Q-Table, over the period of time agents will start to exploit the environment and start taking better actions. We ran the experiments for the braid length from 7 to 11 to observe the results for 1000 and 10000 episodes respectively. The choice of hyper-parameters selection was looked from some of the work in the literature (Gelana Tostaeva, 2020).

To implement the DQN algorithm we use the pyTorch library, it is an open source machine learning library. The idea behind the DQN algorithm is to use function approximators (i.e. Neural Networks) in order to approximate the action-values *Q-values* for any given state. The DQN algorithm has 2 such networks: One is the *train network* (i.e. the one that is being trained), and the other one is the *target network*, which is what the *train network* is trying to approximate. By using the bellman equation, we approximate Q(s, a) with the *train network* and Q (s', a') with the *target network*. This is essentially what DQN does: It performs rollouts in an environment, collects data and then uses this data to train, by performing gradient descent on the Mean Squared Error Loss of the predicted Q values for a given state and the same values but by substituting Q(s, a) (for the action 'a' that was taken) with the bellman equation value. These training iterations take place every fixed number of episodes e.g. in the current case after (n=100) episodes, and after those the *target network* copies the weights of the *train network*. Slowly, after many iterations of training, both neural networks approximate the optimal Q values. During evaluation, for any given state, the algorithm will pick the action that corresponds to the highest Q value, and this is how the agent interacts with the environment. DQN uses the replay buffer, the replay buffer is filled at each step during training in an episode. For example, if the maximum number of steps in an episode are 20, and agent trains every 100 episodes, data stored in the replay buffer will be 100 *20 = 2000. To balance exploration and exploitation, we are using the epsilon-greedy strategy. We first promote full exploration by setting epsilon =1 and update it after each episode to slowly decrease it to 0.05. We ran the experiments for the braid length from 7 to 11 to observe the results for 1000 and 10000 episodes respectively.

Input length	**ep=1000 steps=20**	**ep=10000 steps=20**	**ep=1000 steps=100**	**ep=10000 steps=100**
7	44.3%	71.9%	60.8 %	78.6%
8	32.1%	65.2%	35.6%	71.7%
9	32.3%	57.2%	33.7%	58%
10	25%	58.1%	21.9%	50.9%
11	25.2%	55%	26.9%	47.9%

Table 3: probability of player2 of winning the game, ep=episodes using Q-Learning

Input length	**ep=1000 steps=20**	**ep=10000 steps=20**	**ep=1000 steps=100**	**ep=10000 steps=100**
7	18.9%	75.4%	36.9%	81%
8	15.6%	60.27%	26%	78.4%
9	10.6%	74.2%	19.7%	79.7%
10	10.%	61.9%	13.6%	77.1%
11	9.5%	54.5%	11.9%	75.9%

Table 4: probability of player2 of winning the game, ep=episodes using DQN

In order to evaluate the performance of the RL agents, we run them on braids with different lengths. It is observed from Table 3 and 4, for lesser number of training episodes and larger length of the input the probability of the tangling player to win the game is more, whereas when we train the system for higher number of episodes the probability of the un-tangling player to win the game is more times at the end of training. Looking at the respective tables 3 and 4, it can be observed the untangling player for Q-Learning performs better with lesser number of episodes and lesser number of maximum defined steps, on the other hand with 10000 episodes and maximum number of steps e.g., 100 the performance of DQN agent is more stable and better then as compared to the Q-learning agent. To sum up Untangling player for Q-learning agent performs better when dealing with lesser number maximum defined steps agent can take to perform transforms, whereas the performance of DQN agent becomes more stable while dealing with higher number of episodes and more steps agent can take to untangle the braids.

Conclusion

In this study the main focus is the comparison of the results using Q-Learning and DQN for the problem of untangling of braids. The problem of untangling of braids was approached as a competitive game between two players, where the tangling agent starts with a fixed length of input and applies certain rules to tangle the braid, that tangled braid is the input for the untangling agent which again applies the rules to untangle the braid, ultimately if second agent successfully untangles the braid it wins the round or vice versa. We observe the more we train the model, the more is the probability of the second agent to win the game. The results shows the performance of

Q-Learning is slightly better when compared with DQN for the untangling of braids. In the future we would like to do a more comprehensive comparative analysis of other RL based approaches (Proximal Policy Optimization (PPO), Asynchronous Advantage Actor Critic (A3C), Trust Region Policy Optimization (TRPO)) with the mentioned approaches in this paper.

References

C. Amato and G. Shani. High-level reinforcement learning in strategy games. In *AAMAS*, volume 10, pages 75–82, 2010.

G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

N. Brown and T. Sandholm. Superhuman ai for multiplayer poker. *Science*, 365(6456):885–890, 2019.

B. Chakraborty and E. Moodie. *Statistical methods for dynamic treatment regimes*, volume 2. Springer, 2013.

D. Feng, C. P. Gomes, and B. Selman. Solving hard ai planning instances using curriculum-driven deep reinforcement learning. *arXiv preprint arXiv:2006.02689*, 2020.

P. Gawłowicz and A. Zubow. ns3-gym: Extending openai gym for networking research. *arXiv preprint arXiv:1810.03943*, 2018.

Gelana Tostaeva. Introduction to q-learning with openai gym. https://medium.com/swlh/introduction-to-q-learning-with-openai-gym-2d794da10f3d, April 2020.

T. Graepel, R. Herbrich, and J. Gold. Learning to fight. In *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, pages 193–200. Citeseer, 2004.

J. Grannen, P. Sundaresan, B. Thananjeyan, J. Ichnowski, A. Balakrishna, V. Viswanath, M. Laskey, J. E. Gonzalez, and K. Goldberg. Learning robot policies for untangling dense knots in linear deformable structures. In *Conference on Robot Learning (CoRL)*, 2020.

S. Gukov, J. Halverson, F. Ruehle, and P. Sułkowski. Learning to unknot. *Machine Learning: Science and Technology*, 2(2):025035, 2021.

M. Hausknecht and P. Stone. Deep recurrent q-learning for partially observable mdps. In *2015 aaai fall symposium series*, 2015.

L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

A. Khan, L. Bozzato, L. Serafini, and B. Lazzerini. Visual reasoning on complex events in soccer videos using answer set programming. In *GCAI*, pages 42–53, 2019a.

A. Khan, L. Serafini, L. Bozzato, and B. Lazzerini. Event detection from video using answer set programing. In *CILC*, pages 48–58, 2019b.

A. Khan, A. Vernitski, and A. Lisitsa. Untangling braids with multi-agent q-learning. *arXiv preprint arXiv:2109.14502*, 2021.

M. R. Mendonça, H. S. Bernardino, and R. F. Neto. Simulating human behavior in fighting games using reinforcement learning and artificial neural networks. In *2015 14th Brazilian symposium on computer games and digital entertainment (SBGames)*, pages 152–159. IEEE, 2015.

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.

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): 529–533, 2015.

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

D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

A. A. Tsiatis, M. Davidian, S. T. Holloway, and E. B. Laber. *Dynamic Treatment Regimes: Statistical Methods for Precision Medicine*. CRC press, 2019.

H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

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

S. Wender and I. Watson. Combining case-based reasoning and reinforcement learning for unit navigation in real-time strategy game ai. In *International Conference on Case-Based Reasoning*, pages 511–525. Springer, 2014.