Using Deep Reinforcement Learning with Humans while Training for Agar.io

Alex Zuzow
University of Texas at Austin
azuzow@utexas.edu

Stephane Hatgis-Kessell
University of Texas at Austin
stephane@utexas.edu

Abstract

Deep Reinforcement Learning has achieved superhuman performance in a variety of video games when using high dimensional image inputs. We utilize one such algorithm, the Deep Q-Network, to play the online multiplayer game Agar.io. We train the model against real humans, and are able to achieve better than random performance. We also investigate various state representations, discounting factors, as well as gradient clipping. Our agent is able to successfully learn a decent game-playing strategy, but our analysis shows there is a lot left to be desired. A video description of our project can he found here, and the code with instructions on running it is available here.

1. Introduction

Reinforcement Learning (RL) has had much success in achieving superhuman performance on a variety of challenging video-games [8]. In this paper we explore the use of Deep Q-Networks, an off policy RL algorithm, to play the online multiplayer game Agar.io. While training, the agent plays against humans in real time.

RL algorithms traditionally struggle with high-dimensional states such as images, which are frequently used with video-games. Constructing image features for linear function approximators may require significant computational resources and expert task knowledge, with many of these feature representations still being unable to represent the state space adequately. One major limitation of the linear form is the inability to capture interactions between features [9]. Instead, recent developments have shown the efficacy of using Deep Neural Network (DNN) function approximators for RL tasks. Using DNNs may remove certain convergence guarantees and require large amounts of training data, but allows us to do away with assuming feature linearity. We utilize such a function approximator in our work, and implement various techniques to increase the probability of convergence and improve learning stability.

We apply our approach to the Agar.io game, training our model against real humans. We explore various state space representations, and compare model performance to other human players.

2. Related Work

There has been extensive work on applying reinforcement learning techniques to play various video games, both in training a single agent and multiple agents. We extend and investigate these techniques to the online, multi-player web game Agar.io, where all trained RL models are forced to compete with real humans.

2.1. Q-Learning

Q-Learning [9] is a common RL method that learns to predict the expected reward after taking an action, a_t in state s_t at time step t. The Q estimate target of an (s_t, a_t) pair is updated using the transition $(s_t, a_t, r_{t+1}, s_{t+1})$ as follows:

$$Target(s_t, a_t) = r_{t+1} + \gamma max_a(Q(s_{t+1}, a))$$

Deep Q-Network (DQN) have been introduced to play Atari games [8]. The Q-Network is parameterized as a deep neural network, with raw pixel images as the input. These images are then passed through a series of convolutional and linear layers. The Q target, shown above, is used for back-propagation to update the Q function. Because of its success in Atari, we use a DQN model in this paper.

2.2. Improving Convergence and Stability

The combination of function approximation and off-policy learning inherent with DQNs removes convergence guarantees and enables the deadly triad [9]. To increase the likelihood of convergence, Double Q-Learning was introduced [5], which learns both a policy Q-Network and a target Q-Network. While training, the

policy network is used to find the best action to taken in state s_t, which in turn is used to update the target network. Similar to [8], we use this methodology to encourage convergence in Agar.io.

Gradient clipping with gradient descent has also been shown to converge faster than using a fixed learning rate [12]. We implement gradient clipping when updating the Q function to incur better performance, but also explore the affect of not using gradient clipping.

Additionally, game-play dynamics such as direction and speed can be difficult to capture in images. Frame-stacking [8] can be used to ensure that the raw-pixel state space is Markovian and captures certain domain-specific attributes. Similarly, we use this technique in the context of Agar.io but modify it for better simulation performance.

The experience replay mechanism [7] can be advantageous to training a DQN [8]. With experience replay, transitions are stored in a buffer and then randomly sampled at every step t to update the DQN model. We include this in our model to improve performance and increase stability in the Agar.io domain. A variation of experience replay, prioritized experience replay (PER), has been used to stabilize learning in Agar.io [1]. PER prioritizes transitions to be sampled from the replay buffer based on the temporal difference error:

$TDE_i = Target(s_t, a_t) - Q(s_t, a_t)$

Where transitions in the replay buffer are sampled in accordance to this error. We do not investigate PER in this paper, but doing so is an interesting direction. Various state space representations have been explored in the Agar.io domain [1], as well as how these representations affect divergence when only training an agent to eat pellets. Using grid-representations with CNNs and larger, 128x128 images were shown to outperform smaller, 42x42 images. We do not investigate the affect of image size on performance, and instead use 128x128 images in accordance with this work.

2.3. Gradient Descent

Gradient descent is a first-order optimization algorithm where the gradient of an objective function is calculated, negated, and then minimized in order to find the minimum of the function. Following this gradient to find the objective functions minimum involves stepping into the direction of the negated gradient, with the step-size being constant. Using a constant step-size, however, is a major limitation; this can increase training time and lead to missing the optima. Root Mean Squared Propogation (RMS-Prop) addresses this problem by maintaining a moving average of the square of all previous weight and bias gradients, and dividing the current weight and bias gradient by the square root of these maintained sums [6]. The moving average for the gradient of the model weights, S_{dw}, and the model bias, S_{db}, are given by:

$S_{dw} = \beta S_{dw} + (1 - \beta)(dW^2)$

$S_{db} = \beta S_{db} + (1 - \beta)(db^2)$

Where β is the discounting factor for the history of gradients. With a step size α and a small stability constant ϵ, the weight and bias updates are then given as follows:

$W = W - \alpha \frac{dW}{\sqrt{S_{dw}}} + \epsilon$

$B = B - \alpha \frac{dW}{\sqrt{S_{dw}} + \epsilon}$

We utilize RMS-Prop in the Agar.io domain based on its success in Atari [8].

2.4. Huber Loss

Following the Atari DQN [8], we also used the Huber loss function because of it's lack of sensitivity to outliers. The loss is given as follows:

$$l(\hat{y}, y) = \begin{cases} 0.5(\hat{y} - y)^2 & |\hat{y} - y| < \delta \\ \delta(|\hat{y} - y| - 0.5\delta) \end{cases}$$

2.5. Policy Gradient Methods

Policy gradient methods have been applied to the Agar.io domain and shown promising results when using a continuous action space [11]. Because of lack of compute, however, we decided to discretize the action space and utilize Q-Learning instead. We seek to build previous work [11] by training an RL agent against real humans.

2.6. Multi-Agent Reinforcement Learning

Interesting social behaviours such as cooperation and competition have been learned using multi-agent deep reinforcement learning [10]. In the Atari domain,[10] trained cooperative agents A and B such that if A incurs reward r_t at time step t then so does B. With this convention, the Q-function for agents A and B are identical, and the optimal cooperation strategy is part of the Nash equilibrium of the game [13]. An alternative framework is to allow agents A and B to have their own reward functions, and then to average the reward received by both agents at each step [13].

For competitive agents A and C, if A incurs reward r_t then C incurs reward $-r_t$. This forms a zero-sum game [10] [13].

Additionally, complex competitive behaviours have been shown to arise in simple environments through training RL agents with self play [2].

We would have liked to explore the use of such multi-agent methods in Agar.io, but due to lack of time, fell

short of this goal. Instead, the human opponents are assumed to be part of the Agar.io environment instead of being explicitly modeled.

3. Agar.io Domain

The Agar.io game presents a unique challenge to RL agents, allowing agents to play against real humans while learning complex behaviours. In this game, a player seeks to acquire mass by eating pellets or other smaller players. Thus, an agent must learn to avoid larger opponents, find food, and follow and trap smaller opponents. The game ends when the player is absorbed by another player.

Throughout training, our agent is connected to the Agar.io servers so that it is playing against real humans at all times. Each time step corresponds to approximately 1 second in the game.

3.1. MDP Formulation

We assume that the Agar.io is a deterministic Markov Decision Process (MDP). Because each action in Agar.io not only influences the immediate next state and reward, but future states, rewards, and actions, this is a sequential decision making problem. The MDP is defined by the tuple (S, A, T, γ, D_0, R). S is the set of all possible states, and A is the set of all possible actions. $T : S \times A \times \rightarrow S$. is the transition function, which is assumed to be deterministic. γ is the discount factor, which remains constant throughout. D_0 is the start state distribution, which is uniformly random. R is a reward function, $R : S \times A \times S \rightarrow \{0, -1\}$. The human opponents in Agar.io are treated as part of the environment, which may remove the state representations Markovness.

3.2. State Space

We use frames of size 128x128. A state consists of 3 of these frames stacked on top of each other. Stacking the frames encodes information such as the agents direction and speed. These frames are then passed through several convolutional layers, which extract the state feature that serves as the input for the Q-function. We investigate the affect of using two RGB and gray-scale images. While the RGB image representation leads to a significantly larger state space, such color information may encode information about opponents. We sought to explore this. These frame representations are shown in figure 1.

3.3. Action Space

The Agar.io action space is continuous, however, we wanted to investigate the performance of a DQN. Consequently, we discretized the action space into set of 20 different degrees of direction the agent may move in. The sum of these degrees of movement add up to 360, providing the agent with a full range of motion. Increasing the size of the action space could decrease the DQN's performance. More possible actions entails more exploration needed, which could slow down learning. We found that 20 possible actions did not limit the agents range of motion and did not require excessive exploration.

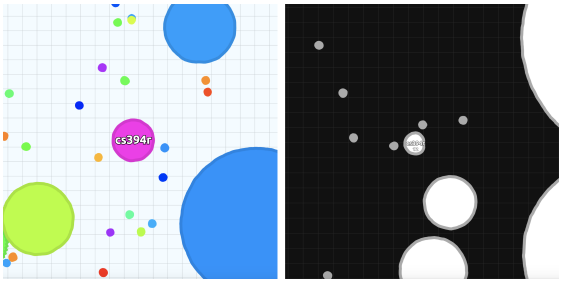

Figure 1: (Left) The RGB frame, with the center purple circle being the agent, the smaller circles being food, and the bigger surrounding circles being opponents. (Right) The grayscale frame, with the center circle being the agent.

3.4. Reward Function

The agent receives a reward of 0 for being alive and a reward of -1 for dying. Such a reward function incentivizes the agent to stay alive as long as possible. A discount factor, $\gamma = 0.999$ is used so as to better assign credit to previous transitions, however, in section 5.2 we illustrate the affects of using different discount factors.

4. Methodology

4.1. Connecting to Agar.io

We used Selenium and ChromeDriver to open Agar.io and control the agent using mouse movements. This enabled us to connect to the Agar.io servers and train the agent using real human opponents. Images of the screen were captured, resized, and pre-processed using OpenCV [4].

4.2. Model Architecture

A Double Q-Network [5] was implemented to encourage stability and convergence. Both the policy and target networks have the same architecture and the same initial weights. The target networks parameters were updated with the policy networks parameters every 5 episodes. We found that this worked best in terms of minimizing the loss function while maintaining stability.

Unless otherwise specified, all gradients are clipped between $[-1, 1]$. As shown in section 5.2, we found this was paramount to convergence.

The Q-networks input is a state and the outputs are the predicted Q-values for each of the 20 actions. This differs from the standard Q-function parameterization, where the Q-function input is a state and action pair, and the output is the predicted Q-value for that pair [9]. Instead, the parameterization we used avoids the need for multiple forward passes when acting greedily over the Q-function [8].

Batch updates of size 64 were used throughout training, which was small enough to avoid long periods of inaction while the agent was deployed in Agar.io. The DQN model parameters were updated using RMS-Prop [6], with a step size of $\alpha = 0.01$ and a gradient history discount factor of $\alpha = 0.99$. These were the Pytorch defaults and they worked well for our domain. The Huber loss function was used because of its ability to handle outliers well.

Exploration is paramount for the model to learn better Q-value predictions. To encourage exploration early on, we used ϵ-greedy, where ϵ was decayed linearly from 0.9 to 0.05 over the course of 200 episodes, and then remained at 0.05 thereafter.

A key problem with using a DNN function approximator is that this method assumes the underlying data distribution remains unchanged. The Q-function update for a current state, however, relies on the Q-functions output for the next state, which essentially creates a moving target. This underlying distribution is non-stationary, which is a problem addressed by the experience replay mechanism. Our experience replay mechanism had a maximum size of 10000 transitions, which we found was the maximum we could accommodate in memory.

The model architecture used is described below, with the various design decisions being chosen in accordance to the Atari DQN implementation [8]. The exception to this is that our state space consists of consists of 3 frames stacked on top of each other instead of 4. This was done to reduce lag time while executing a step in Aagar.io.

The DQN model consists of 5 convolutional layers and one linear layer. Each convolutional layer is followed by batch normalization in order to increase training stability. For each convolutional layer, a rectifying linear unit activation function is used, as well as a kernel size of 5 and a stride of 2. The final linear layer has no activation function. If the pre-processed image is in gray scale, then input to the network is a 3x128x128 image. Otherwise the input is a 3x3x128x128 image. The output of each of the 5 convolutional layers has the following number of channels, in order: 16, 32, 64, 64, 128. The last linear layer is fully connected and has an input of size 128 and an output of size 20, which is equal to the number of actions.

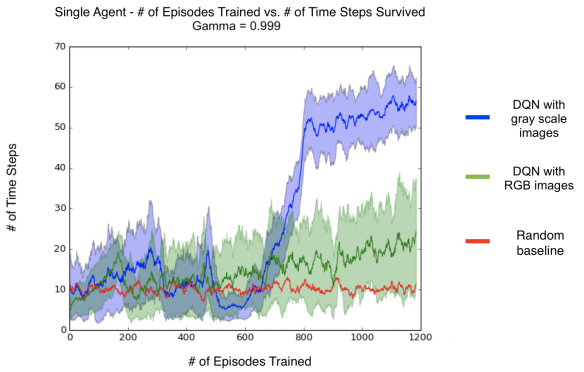

Figure 2: The above graph shows the number of time steps survived by each model during training, where these values are averaged over 3 separate runs. The standard deviation across the 3 runs is shown by the shaded regions. A rolling window of step size 20 is used to further average these results for the sake of clarity.

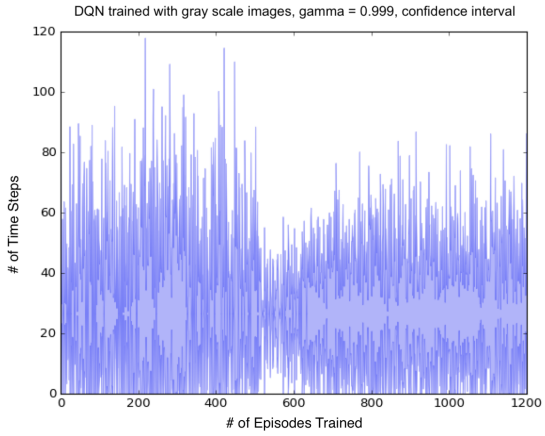

Figure 3: The above graph shows the 95% confidence interval for the best performing model, the DQN trained on gray scale images, across 3 runs.

5. Results and Discussion

5.1. State Representation

First, we compare the performance of the single-agent DQN model with a state space comprised of RGB images and a state space comprised of gray scale images. For these experiments, $\gamma = 0.999$. Because a negative reward is only given upon episode termination, we measure the agents performance as the number of time steps it survives, with a time step being roughly 1 second. The two different models are compared against an agent that selects actions uniformly. Figure 2 illustrates the results,

Table 1: The mean and standard deviation of the number of time steps survived by each agent across 10 games is shown below.

Model	DQN with RGB Images	DQN with Gray Scale Images	Human 1	Human 2	Random Baseline
Mean	18.2	51.8	98.7	602.4	11.4
STD	22.6	18.3	20.5	15.2	28.1

where each models performance is averaged across 3 runs. Table 1 shows how the models performs against real humans (the authors of this paper) and a random baseline over a set of 10 episodes after training is complete.

It is evident that there is high variance between runs, likely because the models are being trained while playing real humans. Therefore, for each episode and for each run, the agents are likely encountering different humans with different skill levels. One possible area of improvement is to increase the number of runs used to evaluate each model at training and test time. Because of lack of time and computational resources, however, we were unable to do that.

There is also a high variance both between and within individual human players, as shown in table 1 for humans 1 and 2. This highlights not only the difficulty of Agar.io but the large variety of possible game-playing strategies. The variance within players points to the affect that different opponents may have. We treated the human opponents as part of the environment. This is a limitation however, and one that could be overcome by attempting to model the human opponents and treat them as other agents.

From our results it is clear that the best performing model is the DQN trained on gray scale images (figure 2). It is evident, however, that this model has a high variance and may not always perform well (figure 3). When observing this models behaviour, it consistently learns to go into the corners of the Agar.io board to avoid encountering other agents. These corners also have food, although usually less than other areas. While this strategy increases the number of time steps survived, this is not ideal game playing behaviour. Instead, a good player should increase its odds at survival by eating food and other players. Most human players accomplish this by staying near the middle of the board, where there are plentiful amounts of food and other players. The agent may have learned to avoid the center because there is also a greater risk of being eaten there, which is hinted at by the high standard deviation amongst human players shown in table 1. This is a clear shortfall of our approach, and one that can be attributed to the sparse reward function used. If, instead, we had rewarded the agent for eating food or other players explicitly, then perhaps we would see better strategies arise. Consequently, this agent performs better than a uniformly random agent but does not achieve near-human performance. The significant gap between the agent and the humans performance, shown in table 1, illustrates room for learning more complex strategies. Besides using a dense reward function instead of a sparse one, this can be encouraged through exploration as well. We used annealing ϵ-greedy to encourage exploration early on, but after the 200th episode the ϵ value is very low. This partly disables the ability to explore different strategies later on. Intrinsic motivation has shown promising results in exploration for continuous state and action spaces [3], so that may improve results.

When training the model on RGB images, only slightly better than random performance is achieved, and the standard deviation of number of time steps survived only decreases slightly. This suggests that no real strategy is learned, only a weak notion of avoiding other players. The large state space dimensionality, combined with function approximation and off policy learning, induced a scenario where convergence was difficult to achieve. One possible other shortcoming of our approach here is the size of the Q-Network used. We could have experimented more with different model architectures, especially considering that the RGB image state space is significantly larger than the gray scale image state space but the same network is used.

Despite the dissimilarities between episodes, mainly the different human opponents that are present, the DQN is still capable of learning better than random behaviour. Training the model against more consistent opponents, such as other pre-trained agents, may increase the agents performance when introduced to human opponents. This could be a more promising avenue to explore.

5.2. Discounting

Next, we explored the affect of different discounting rates on the DQN model trained on gray scale images. The results are shown in figure 4. We found that a discount rate of $\gamma = 0.999$ worked best, likely because of the reward sparsity. We found that when $\gamma = 0.9$, the agent learns to occasionally avoid being eaten by bigger players, but is unable to learn the long-horizon strategy of moving to the edge of the board where there are less

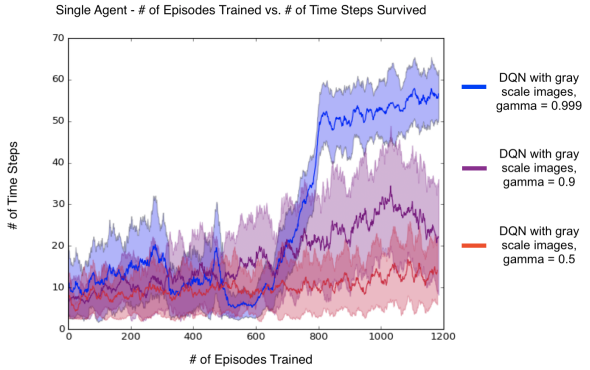

Figure 4: The above graph shows the affect of discounting for the DQN model trained on gray scale images. The results are averaged over 3 runs, and the standard deviation is shown by the shaded regions. A rolling window of step size 20 is used to further average these results for the sake of clarity.

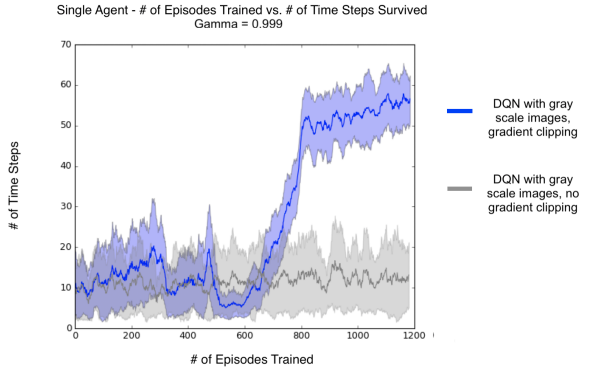

Figure 5: The above graph shows the affect of gradient clipping for the DQN model trained on gray scale images. The results are averaged over 3 runs, and the standard deviation is shown by the shaded regions. A rolling window of step size 20 is used to further average these results for the sake of clarity.

opponents. When $\gamma = 0.5$, the agent performs on par with the random baseline. This highlights the importance of a low discounting rate for Agar.io when using a sparse reward.

5.3. Gradient Clipping

Additionally, we wanted to look into the importance of gradient clipping for Agar.io. For only this experiment we turned gradient clipping off, and found that there was a stark decrease in model performance (figure 5). This highlights the problems induced by the Deadly Triad, where the combination of function approximation, bootstrapping, and off-policy learning can make convergence difficult to achieve. In particular, we found that exploding gradients caused our model to diverge, but the simple gradient clipping technique was capable of alleviating this problem.

6. Conclusion

We implemented a DQN to play the online game Agar.io, where our model was trained while playing real human players. Because of the Deadly Triad, achieving convergence proved to be rather difficult. Through a combination of gradient clipping, Double Q-Learning, experience replay, and experimentation with various discount rates and state representation, we were able to achieve consistent improvement in the models performance. We found that through using gray scale frames, with multiple frames stacked on top of each other, as well as $\gamma = 0.999$, our model was able to learn a decent game playing strategy that performed better than random. The model, however, failed to perform anywhere near as well as a human player. We largely attribute this to the sparse reward we were using, and believe that with a dense reward more complex behaviour could emerge.

Besides this, there were many shortfalls with our approach. We did not explicitly account for the human opponents in the game, but rather assumed them to be part of the environment. This may have removed the Markov property from the state representation. Modeling the human opponents, or incorporating multi-agent RL, could be an interesting extension. Additionally, while we used a DQN with a discrete action space, policy gradient methods may outperform DQNs in a variety of video-game domains and thus could be more apt to play this game. We also observed a high variance both for the DQN agent and for human players, indicating that measuring our models performance over more runs would be beneficial. A better exploration method, such as intrinsic motivation, could replace our annealing ϵ-greedy algorithm and lead to better results as well.

Despite these limitations, we show that a DQN is capable of learning reasonable game playing behaviour when trained against real humans. That being said, inducing convergence was rather tricky, and required both hyper-parameter tuning and various techniques being implemented. This points to the difficulty of applying DQNs to a video-game domain with a high dimensional state space.

References

[1] Nil Stolt Ansó, Anton Orell Wiehe, Madalina M Drugan, and Marco A Wiering. Deep reinforcement learning for pellet eating in agar. io. In *ICAART (2)*, pages 123–133, 2019.

[2] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. *arXiv preprint arXiv:1710.03748*, 2017.

[3] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. *Advances in neural information processing systems*, 29, 2016.

[4] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[5] Hado Hasselt. Double q-learning. *Advances in neural information processing systems*, 23, 2010.

[6] Thomas Kurbiel and Shahrzad Khaleghian. Training of deep neural networks based on distance measures using rmsprop. *arXiv preprint arXiv:1708.01911*, 2017.

[7] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3):293–321, 1992.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[9] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[10] 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.

[11] Anton Orell Wiehe, Nil Stolt Ansó, Madalina M Drugan, and Marco A Wiering. Sampled policy gradient for learning to play the game agar. io. *arXiv preprint arXiv:1809.05763*, 2018.

[12] Jingzhao Zhang, Tianxing He, Suvrit Sra, and Ali Jadbabaie. Why gradient clipping accelerates training: A theoretical justification for adaptivity. *arXiv preprint arXiv:1905.11881*, 2019.

[13] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of Reinforcement Learning and Control*, pages 321–384, 2021.