UNIVERSITY OF CHICAGO

Distributed Reinforcement Learning

by

Naga Nithin Manne

https://github.com/nithinmanne/ada-project

December 2020

Abstract

In this project, I want to use Reinforcement Learning to train an agent to play Atari games. The learning model is a deep Q-network using a convolutional neural network to train on the raw pixel values of the game screen and output the value function estimating future rewards. Along with this implementation, I will be trying to use distributed computing to parallelize this learning process.

Contents

Chapter 1

Introduction

Reinforcement learning is an area of machine learning in which a software agents takes actions in an environment in order to maximize the cumulative reward obtained. It differs from other types of machine learning such as supervised or unsupervised learning as instead of needing a dataset to train on, the agent generates its own dataset by taking actions on the environment, and learns from it to maximize the reward. Figure 1.1 shows this cycle.

The environment in reinforcement learning is usually modelled as a Markov Decision Process, which is a probabilistic model that describes how the environment behaves when applying a particular action on a given state. This would result in a new state and the agent receives a reward corresponding to this state change. The goal of reinforcement learning is to effectively create a policy that tells the agent the best move that it can take while maximizing the cumulative reward generated through the its interaction with the environment.

There are many algorithms which can be used to generate an optimal model for a given environment. Q-learning is a model-free reinforcement learning algorithm that can be used to

FIGURE 1.1: Reinforcement Learning

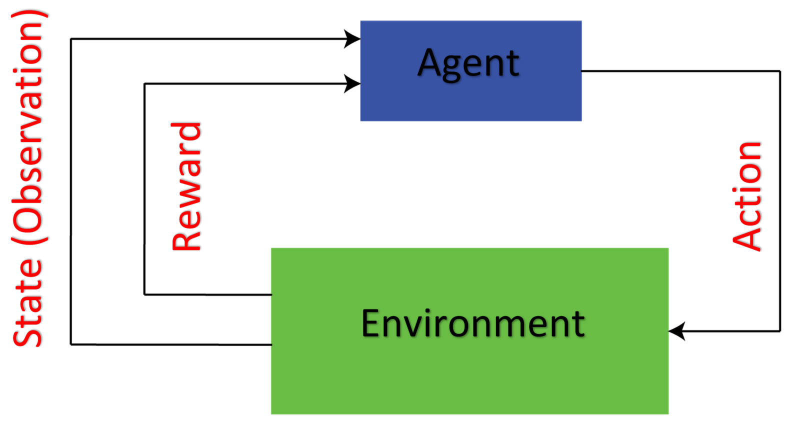
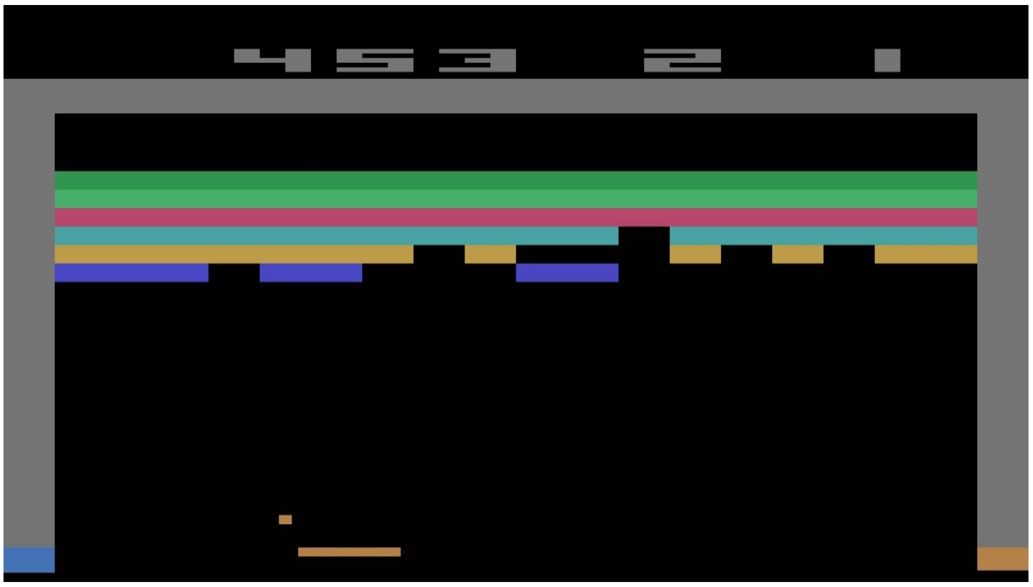


FIGURE 1.2: Screenshot of Atari Breakout



learn the policy. Q-learning doesn't require the actual probabilistic model of the environment, but actually learns the policy by means of exploration, which is randomly exploring the environment to learn about the state changes and rewards associated with any particular state and action, and then exploits this information by choosing the set of actions that maximize the cumulative reward that's generated by the agent over the course of its interaction with the environment

The Q function in Q learning refers to the maximum reward for a given state by taking a particular action. An optimal policy here with the Q function would be to simply take the action that gives the maximum possible rewards and recursively keep going. This function is generated by a mixture of exploring the environment and exploiting the already known Q fucntion values. There is a balance between this exploring and exploiting, and there are very simple techniques like ϵ-greedy method where you explore with a probability ϵ.

In my project, the environment that I will be using is an Atari 2600 game called Breakout. Its a simple game, where the goal is to break bricks by hitting a ball with a paddle without letting the ball fall below the screen. The state of the environment that the agent will see here is actually the raw image of the screen just any player would, without any other information provided. There are four actions that the agent can take on the environment here, which are to do nothing, move left, right or to fire. The fire action is actually just a special action that used to start the ball, and not used while the ball is actually moving on the screen. Figure 1.2 is a high resolution screenshot of this game.

The state of this Atari game is an $210x160$ RGB image. Clearly, as this state space is extremely large($(255^3)^{210*160}$ which is about 1.78×10^{242579}), the Q function is too complex to be modeled exactly, as the Q function essentially requires a value for each possible state. So, to solve this problem, I will actually be using neural networks. The main idea here to use neural networks to actually estimate the same Q function. This algorithm is called Deep Q-learning, and in this we use deep convolutional neural networks on the raw pixel images to estimate the Q values. We train our network using experience replay where we keep a large collection of prior actions and corresponding results and randomly sampling this data to train.

The next part of this problem that I wanted to explore is to use distributed computing to speed up the reinforcement learning process. Since reinforcement learning in general, is quite a CPU intensive process, as it requires generating the dataset which needs to be as large as possible, and this is practically CPU bound. So, an easy way to scale this data generation could be to have the worker computers simulate the environment and generate create the replay buffer, then this data can be transferred to the master node, where it uses this buffer to train on and update the model. This model can then be re-transferred back to the worker computers to generate more data.

Chapter 2

Literature Review

The main paper that I used as inspiration for this project is Playing Atari with Deep Reinforcement Learning from Deepmind[6], which is actually the first paper to use deep neural networks to estimate the Q functions on Atari games. I found this paper actually really interesting, and always wanted to implement it. In this paper, they used convolutional neural networks to train raw pixels to estimate the Q function. When it came out, this algorithm was actually best performing algorithm ever on any of these games, and it did that without having any specific learning algorithm for the games, that is the network is generic enough to learn any type of game.

Because deep Q learning actually suffers from an overestimation problem, there were variants like the Double Q-learning[8] used to rectify this problem and eliminate the bias.

RLlib[4] is the library I used to achieve distributed computing, this paper is also very interesting, as they applied many distributed computing techniques to various reinforcement learning algorithms to measure the performance. Though DQN didn't have a high degree of parallelism, there were many other algorithms such as Ape-X[3] that performed much better and converged faster.

There were a few more variants of distributed RL like IMPALA[2] and Advantage-Weighted Regression[7] which I read but didn't actually implement. There were also some multi-agent RL algorithms[5] which could be parallelized.

Chapter 3

Detailed Description

3.1 Environment

Like I mentioned earlier in Section 1, the environment that I'm using is an Atari game called Breakout. This is a simple game where the goal is to break bricks with a ball using the paddle. This environment is from a library called OpenAI Gym[1]. This library is a very popular library for Reinforcement Learning and provides many such environments. It also exposes a very simple API to implement pre-processing.

The main functions that can be performed on the environment are reset and step. Reset resets the environment to its initial state, and step is used with an action to generate the observation and reward, which in this case is a $(210 \times 160 \times 3)$ RGB image and an integer reward for each block broken in the step.

3.2 Pre-Processing

The environment that I'm using to train on is the Atari game Breakout. The state of this environment is the current pixel values of the screen. This is a 210×160 RGB image. This image is still unnecessarily too complex, and would require more computing to train on and predict using the neural network.

So, to optimize for this, I will firstly convert the image to grayscale, which now reduces the dimensionality of the observation(state) space to 210×160. Another easy pre-processing step is to scale down the image to a smaller size. This is done because, the image is now smaller, which means that the not only is the data occupied by it much smaller, but the training and prediction phase on this image can be considerably faster. This pre-processing step also doesn't necessarily remove any information at all from the image. This is because, even with the smaller size the main elements of the image, which is the blocks, the paddle and the ball are still clearly visible, and the RGB color space and bigger resolution is only good for aesthetics.

The next pre-processing step is to normalize the actual color intensity values of the images. That is, the 0-255 uint8 representation, which is usually used to represent the white intensity, is scaled down to a floating point number between 0 to 1. This is usually done as many neural network models, or corresponding activation functions sometimes require the inputs to fall in this range for optimal performance by the network layer.

The last pre-processing step on the observation is called frame stacking. In this step, as part of each observation, rather than the agent receiving the current image of the screen as the observation, it receives a stack of the last 4 images of the screen that were generated by the environment. Though this now increases the dimensionality of the state space by a factor of 4, this has an added advantage, which is that this lets the network train on what is short sequence of events rather than a still image. This can have a practical advantage of the network being able to train movement through the picture.

3.3 Replay Memory

Replay Memory is actually an important part of reinforcement learning. In deep Q learning, we use the replay memory to sample observations from which we train on. Since we use it for training, it will be more effective if we can keep the replay memory as large as possible.

The choice of data structure is actually very important for this memory, as there are certain operations that are done very frequently on this data store, and it would extremely beneficial to choose a data structure to specifically optimize for these operations. The operations that are done frequently on this are adding an element to the end and removing an element from the front, and randomly sampling a small batch of observations from it. This sample size is generally much smaller than the underlying data structure.

Some of the built-in data structures that can be used would to use a Python list or deque. The list is an variable length array data structure. Randomly sampling a small batch of data from it is $O(\text{size of batch})$ as its essentially backed by an array. Adding data to the end is also $O(1)$ as the cost is amortized. But the other frequent operation is to remove from the front of the list. This operation on a list in Python is actually $O(n)$, which would make if very inefficient to use with a large observation store. Another data structure is the Python deque, this is actually a double ended queue. Since its a queue, removing from the beginning and adding to the behind is $O(1)$. But randomly sampling any data on a queue is (n), which would make it impractical for the sampling operation.

But since the size is fixed, the obvious choice is to use a ring buffer. This would make removing and inserting very fast, as that's the exact reason cyclic buffers are made for. The next operation of sampling on it is also very efficient and is $O(\text{size of data})$. This is because its essentially backed by a fixed size array. In my implementation, I actually used a Numpy array as the backing data store. This is because its more efficient to pre-allocate memory, mainly on large memory stores, and it also doesn't have any cost to move the array. Its also more convenient to randomly sample dataset using Numpy slicing, than it is on a Python list.

3.4 Neural Network

The neural network forms the backbone of deep Q learning. As mentioned earlier, deep Q learning involves using a deep convolutional neural network to estimate the Q function on the environment. The network I used consists of 5 layers in total, with 3 convolutional neural networks, and and 2 more fully connected layers. I used the ReLU activation function for the network. For the loss and the optimizer for this network, I used the Huber loss, and Adam which is just a variant of stochastic gradient descent.

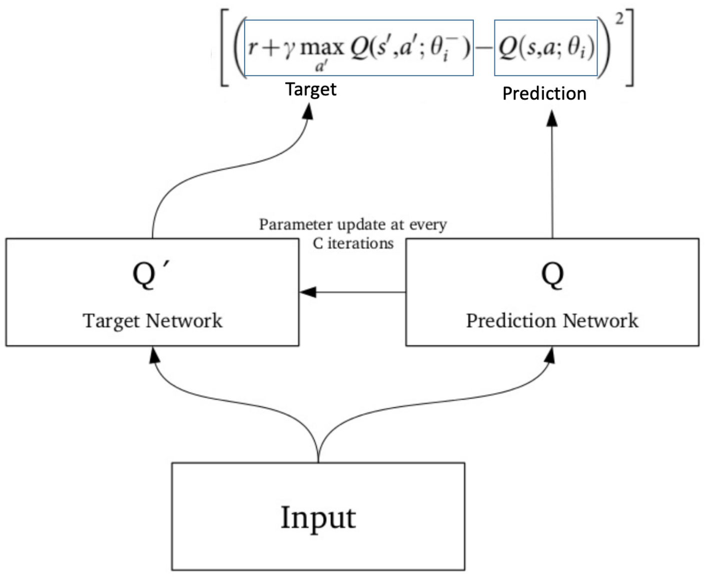
To implement this neural network, I used Keras to directly create the layers. Since the main focus of my project is to explore the Reinforcement Learning part of this implementation, I decided to use the Keras high level API, rather than use TensorFlow to implement the convolutional kernel, and the neural network layers.

3.5 Learning Algorithm

This is the crux of the Deep Q learning algorithm. It actually contains two neural networks, a prediction network and a target network. Now, both these networks are created in the same way as in Section 3.4. The prediction model is the main model which is trained at each training step, and is the also the network that's used to predict the next action based on the current state. The target is a frozen snapshot of the prediction network that is updated very infrequently. Its used to prevent instabilities, as otherwise, any one change can spiral out of control as its using itself to learn on. In DQN, since we never have an exact value function, we actually need to use the model itself on the next state to find the target action. Figure 3.1 shows a simple block diagram of the DQN learning process.

Another important process of this algorithm and reinforcement learning in general is the balance between exploration and exploitation. That is, the agent needs to sometimes explore, possibly making mistakes in order to learn more about the environment and become better. Exploitation here refers to only using the prediction model and take the best possible action.

FIGURE 3.1: Simple Block Diagram for DQN


$\left[\left(r+\gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]$
Target
Prediction
Parameter update at every C iterations
Q′
Target Network
Q
Prediction Network
Input

There are many strategies to choose between exploration and exploitation, and the strategy that I used is called ϵ-greedy. Its a simple strategy where the agent decides to explore rather than exploiting with a probability of ϵ. Another small variation to this strategy is having a decaying ϵ. This means that the ϵ will start out much higher. Clearly this has the benefit of encouraging more exploration initially and slowly reduces. This is actually very useful, as initially the completely untrained model wouldn't know anything and probably just predict the same action every time, which would probably generate a lot of useless observation.

I exclusively used Numpy and TensorFlow operations in the learning code, as using pure Python functions are sometimes very slow. In fact, I didn't use any Python loops in the code for the learning implementation, and instead using a mix of Numpy slicing and other Numpy/TensorFlow functions.

In DQN, since the predictions are very noisy, we actually use the maximum action values as an estimate of the maximum expected action value. But this is susceptible to overestimation, as this max action would could also be the action with the most noise, which gets picked up. There is a variant of DQN called Double Q-learning, where try to separate the estimation part from the actual choosing part. That is, instead of just selecting the maximum action, we instead use the prediction model to choose which action to actually use while training. Since these will be independent, that bias would now not exist. There is also another way to implement Deep Q learning, by having both networks predict and learn, using one for training the other and vice versa. Since both these are independent, the bias would not exist anymore.

Also initially, for a few turns, we exclusively use exploration to generate a lot of data, before we even start training. Another implementation detail is that the update of the training net which I'm doing every 1000 steps, is done by simply copying the weights from the prediction network to the target network.

3.6 Scaling

The next major part of my implementation is scaling the model using distributed computing. Since reinforcement learning algorithms usually take a long time to converge to the optimal strategy, usually in the order of hundreds of millions of steps, and also since a major part of the learning process, the environment simulation and data generation, is CPU bound, I wanted to use distributed computing to try and speed up this part.

Most of the implementation of this scaling is done using a library called RLlib.
RLlib is an open-source library for reinforcement learning that offers both high scalability and a unified API for a variety of applications. RLlib natively supports TensorFlow, TensorFlow Eager, and PyTorch, but most of its internals are framework agnostic.

RLlib is a part of Ray, which is an open-source framework for distributed computing in general for Python. The main implementation of the learning algorithm is the same, but there are a few caveats to watch out for when implementing it using RLlib.

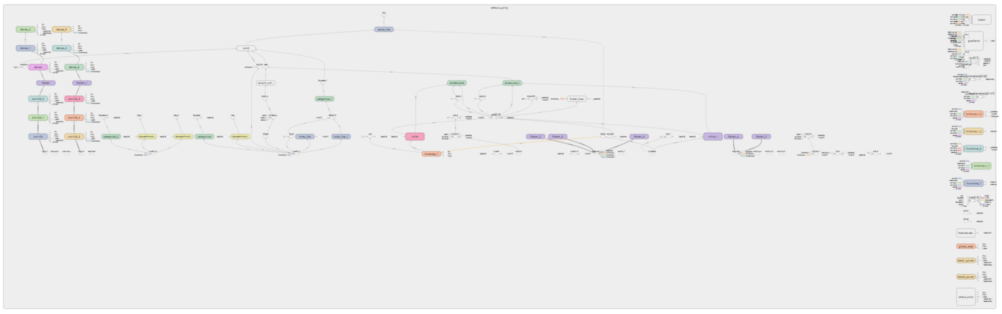
3.6.1 Environment

RLlib supports OpenAI Gym environments without any compatibility layer, and it actually pre-processes the environment observations internally

3.6.2 Model

RLlib also supports TensorFlow/Keras and PyTorch models out of the box. TensorFlow is supported by the library creating the tensor graph in TensorFlow and then sending this evaluation graph to the worker nodes. Due to this, I was actually able to use the exact same model without much changes. The only change it to implement my model as a subclass of a corresponding TensorFlow model class in RLlib, and implementing the corresponding required API.

FIGURE 3.2: TensorFlow graph visualized using TensorBoard



3.6.3 Policy

The RLlib policy is the main algorithm that's implemented to tell RLlib what to do. Since RLlib uses distributed programming, it requires a strict way in which the code should be implemented. This makes it so that it takes of all the distributed part, without requiring any other specific code.

The main component of the policy is the implementations of the function to build loss of the model. RLlib actually distributes the model across all the nodes, generates the data and returns a batch of the training samples to train the model with. Like I mentioned earlier, the library actually generates a complete TensorFlow graph of the full network, including the loss computation, and the optimizer steps. This requires that all the APIs that can be used in implementing this be pure Tensor operations, i.e., not normal numerical/Numpy operations, because the code is not executed with actual data, but rather executed with a TensorFlow placeholder input, which generates the actual graph.

I was able to extract this graph by putting a break point and visualize it using TensorBoard. Unfortunately it was not possible to download the actual SVG of the graph, but I included a screenshot in Figure 3.2. The figure is probably impossible to read clearly, but in it I was able to see the examine and see the two models which are on the left, and the logic for calculation loss which is in the middle. This graph actually even helped me debug some problems which were caused broadcasting. On the right were implementations of separate parts like the optimizer and the loss function.

Implementing these policy was probably the hardest part of this project, as the documentation for RLlib was not very clear about its APIs, and I actually needed to examine the source codes of TensorFlow and RLlib to implement it. The actual logic of the function to generate loss is the exact same thing as Learning Algorithm in Section 3.5. However, instead of actually training on the model, I just return the losses.

3.6.4 Execution Plan

This part actually tells the workers and the master node what to do. It defines the sequence of execution of each step in the process. Since RLlib is focused on Reinforcement Learning, it has APIs here for most of the required framework. The procedure here is create the replay buffer, on all the worker nodes and them dispatch the model so that they can generate the observations. Then these observations are sent back to the master node, there the training is done normally on the neural networks using the loss function defined in Section 3.6.3. After training, the model parameters need to now be sent back to the workers to complete training.

The target network update should also be defined here, RLlib actually provides an API to do this as well, but the implementation was actually very obscure and it required adding a special attribute to the policy which generates the TensorFlow graph needed to update the weights.

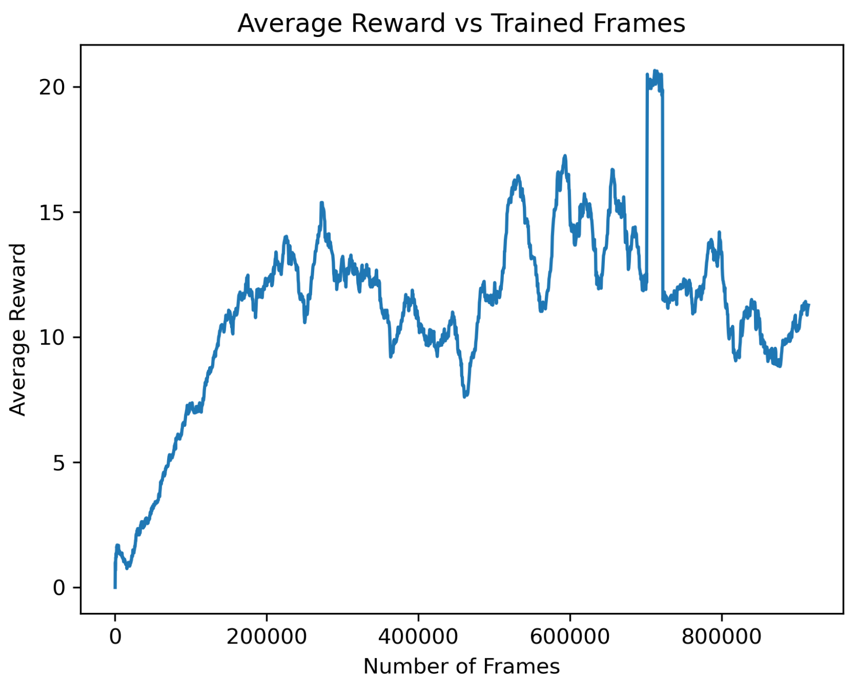
Chapter 4

Results & Future Work

It was actually very interesting to watch the agent learn on the environment and see it play the game. The difference was very clear, when the agent stops moving completely randomly and slowly tries to follow the ball. It was also very fascinating to watch the agent figure out ways to min-max the score by hitting the ball to the side and taking it to the top gaining a large amount of reward.

However the performance of my model was not as good as the performance achieved by the Deepmind[6] paper, as the performance they achieved seemed to be much higher than what

FIGURE 4.1: Average reward over time while training



I was able to run at. In fact, they used a replay memory of over 225GB and trained for about 200 million frames. My model took about 10 hours to train to a million frames, and it seemed infeasible to train for that long. The performance of my model is however still very good, and it was still able to get very high rewards, albeit slightly lower.

Regarding the distributed computing of the reinforcement learning algorithm, the parallelism present is in the generation of data, but since this environment is a relatively quick to generate observations, and DQN updates the model frequently, the parallelism didn't achieve as much speed up as I anticipated.

I was also experimenting with several other reinforcement learning algorithms, that are better suited for distributed computing. One of the other models that I found very interesting was Ape-X(Distributed Prioritized Experience Replay)[3]. In this implementation, for the experience replay, instead of randomly sampling the observations from the replay memory, they used a prioritized sampling. This lets the model converge much faster as it trains with data of higher quality. And this process is also extremely CPU intensive as its required to sort the entire dataset, and can hence achieve a higher degree of parallelism.

In the future, I actually want to extend this algorithm, and actually try to train on more complex games. As a gaming enthusiast, it was actually very interesting to see the different kinds the approaches that reinforcement learning algorithms come up while trying to min-max the reward.

Bibliography

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. *CoRR* abs/1606.01540 (2016). arXiv:1606.01540 http://arxiv.org/abs/1606.01540

[2] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. 2018. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. *CoRR* abs/1802.01561 (2018). arXiv:1802.01561 http://arxiv.org/abs/1802.01561

[3] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. 2018. Distributed Prioritized Experience Replay. *CoRR* abs/1803.00933 (2018). arXiv:1803.00933 http://arxiv.org/abs/1803.00933

[4] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Joseph Gonzalez, Ken Goldberg, and Ion Stoica. 2017. Ray RLLib: A Composable and Scalable Reinforcement Learning Library. *CoRR* abs/1712.09381 (2017). arXiv:1712.09381 http://arxiv.org/abs/1712.09381

[5] Yiheng Lin, Guannan Qu, Longbo Huang, and Adam Wierman. 2020. Multi-Agent Reinforcement Learning in Time-varying Networked Systems. arXiv:2006.06555 [cs.LG]

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR* abs/1312.5602 (2013). arXiv:1312.5602 http://arxiv.org/abs/1312.5602

[7] Xue Bin Peng, Aviral Kumar, Grace Zhang, and Sergey Levine. 2019. Advantage-Weighted Regression: Simple and Scalable Off-Policy Reinforcement Learning. *CoRR* abs/1910.00177 (2019). arXiv:1910.00177 http://arxiv.org/abs/1910.00177

[8] Hado van Hasselt, Arthur Guez, and David Silver. 2015. Deep Reinforcement Learning with Double Q-learning. *CoRR* abs/1509.06461 (2015). arXiv:1509.06461 http://arxiv.org/abs/1509.06461