ORIE 6590 Project Report

Yueying Li, Yujia Zhang

April 2021

1 Introduction

Ride-hailing services have seen a rapid rise in the past decades. Companies such as Uber, Lyft, and Didi Chuxing allow passengers to request rides through an app and match them with drivers that fulfill their requests. This matching problem is intrinsically complicated due to many reasons, including

- spatial complexity: cars and passengers are distributed across different regions;
- temporal complexity: fulfilling a passenger request, as well as routing an empty car to pick up a passenger, involves a time duration and requires the planner to foresee future impact of the decisions;
- scalability: when there are many cars to be arranged and many passenger requests, matching them becomes a huge combinatorial problem that is in most cases intractable;
- parameter uncertainty: traffic parameters (travel time, passenger arrival rates) vary over different times of the day; moreover, the planner may only have partial knowledge of the parameters.

As such, the operations research community has adopted approximation methods to solve this problem. One example is [1], which uses a queueing network model and derives a fluid-based approach that is optimal in the asymptotic "large-market" regime, under the assumption that the centralized planner knows the traffic parameters.

In the recent years, the ridehailing problem has also been tackled using reinforcement learning (RL) approaches. Here, the problem is modeled as a Markov Decision Process (MDP) and the planner seeks to make smart decisions so as to maximize the reward it accumulates in the long run. The development in RL theory, such as methods for policy optimization, can then be deployed to improve the policy the planner takes.

Along this line, [2] builds on the work in [1] and adopts an RL approach to tackle the problem of trip assignments to cars across a collection of regions. In particular, they assume a centralized planner without knowledge of traffic parameters assigns trips to available cars in the system. Their approach alleviates the combinatorial challenge by turning each collection of trip assignments into a *sequential* decision making problem, making the decision for one car at a time. They then use proximal policy optimization (PPO) [7] to train the agent, yielding a policy that outperforms [1]. Performance is measured by the fraction of ride requests fulfilled over the entire time. We also use this metric to measure performance in our report.

Goals for the project:

- Understand the model in [2] and implement it as a custom gym environment;
- Implement the PPO algorithm in [2] and observe its performance;
- Investigate the value provided by empty-car-rerouting by disallowing it in the environment and comparing the results;
- Implement RL algorithms other than PPO and observe their performance. Along the way, also observe how different design details (e.g., hyperparameters) affect the training and outcome.

2 Model dynamics

Overview We model a five-region traffic network with a fixed horizon and number of cars. A centralized planner has the information of the location of all cars as well as the passenger arrivals in each new epoch (i.e., time-of-day). Each passenger has an identical "patience" time they are willing to wait to be picked up. At each epoch, the centralized planner assigns actions to some cars in the system through a "sequential decision making process" (SDM). Below we give descriptions for the model parameters, state and action space, reward function, and details of the SDM. Lastly, we mention a few details in the implementation of the simulator.

2.1 Model Setup

Parameters The model uses the following parameters:

- **Horizon:** $H = 360$. This is the total number of epochs in a "working day", which are divided evenly into three periods, which we denote q, each with length 120.
- **Number of regions:** $R = 5$.
- **Number of cars:** $N = 1000$.
- **Passenger arrival rates:** λ. This is a two-dimensional array in $\mathbb{R}^{3\times R}$, where $\lambda[q,i]$ represents the expected number of passengers arriving to region i in period q.
- **Distribution of ride requests:** P. This is a three-dimensional array in $\mathbb{R}^{3\times R\times R}$, where $P[q,i,j]$ represents the probability that a request starting at region i in period q has j as its destination. By construction, $\sum_j P[q,i,j] = 1$ for all q, i.
- **Travel times:** τ. This is a three-dimensional array in $\mathbb{R}^{3\times R\times R}$, with $\tau[q,i,j]$ representing the travel time from region i to region j in period q.
- **Patience time:** $L = 5$. The maximum time a passenger is willing to wait for a matching.

The values of λ, P, τ are identical to those listed in the appendix of [2].

State space The state has the following components:

- **Current epoch:** an integer taking values from 0 to H;
- **Cars:** an iterated two-dimensional array, whose $[i,k]$ entry corresponds to the number of cars with region i as their destination and will arrive k time steps later. We say a car is available for (requests originating from) region i if its destination is i and it will arrive in no more than L time steps. Using such (destination, time-till-destination) tuples to represent the cars state has the advantage of making the problem scalable, as the centralized planner can focus its attention on the cars available for one origin of interest, rather than arranging every car on the map;
- **Passengers:** a two-dimensional array, whose $[i,j]$ entry is the number of ride requests from region i to region j made in the current time epoch;
- **"Do-nothing" cars:** a two-dimensional array similar to the cars component; it records the number of cars that do not undergo a change in destination in the decision process for the current epoch. We will elaborate on this in our discussion of the transition dynamics later.

2.2 Action Space, Transition Dynamics, and Reward

Actions and transition dynamics We discuss the action space and transition dynamics together because they are closely dependent on each other. The action space is the space of size-two tuples where each entry takes values in $\{1, \cdots, R\}$. An action (i,j) represents assigning a trip with origin i and destination j.

At each time epoch, actions can be taken for each car that is within time L away from the region it is heading to. This set of cars, called "available cars", can be assigned new trips. Once such assignments are made, the time progresses to a new epoch, where cars move forward one unit of distance closer to their destinations, the passenger requests are reset, and new passenger arrivals are sampled.

Deciding actions for a large set of cars at each epoch is difficult to achieve all together. The sequential decision making process (SDM) combats this challenge by imposing an order on the cars awaiting trip assignments at each epoch; taking actions in sequence reduces the complexity of the decision-making.

An SDM is run at each epoch on the available cars. We present the SDM algorithm for one epoch in Algorithm 1. It loops over all the regions and focuses on the cars available for each region o (as an "origin"). Actions (trip assignments) are made sequentially to one car at a time. An action can fall into three categories:

- Matching of a ride request: as passenger matching has priority, a ride request starting from o is matched as long as there are cars available at o;
- Empty car rerouting: the car is idling at o and the action relocates it to another region;
- "Do-nothing": the action does not change the destination of the car.

Algorithm 1: Generating atomic actions in the SDM process for one epoch

```
Result: Transition dynamics in the SDM process for one epoch
initialization;
for each region o do
    get number of cars available for dispatch at o;
    if there are such cars then
        if there is a request starting at o then
            match car to passenger;
            car status is updated; passenger matrix is updated;
        else
            for each available car at o do
                if car is idling at origin o and action places it to $d \neq o$ then
                    empty-car routing; cars status is updated;
                else
                    "do-nothing"; car is excluded from available cars pool and placed into
                      "do-nothing" pool;
                end
            end
        end
        exclude cars with assigned trips from the available cars pool;
    end
end
```

Note that each step of the SDM takes action on *one car*. At each step, the change in car destination is reflected in the cars component; if a passenger request is taken, it is subtracted from the passenger matrix; if a car is assigned "do-nothing", the "do-nothing" component in the state is updated correspondingly. It is necessary to separately keep track of the "do-nothing" cars, because otherwise they will still be considered available based on distance. The SDM for each epoch lasts until all available cars at this epoch receive their trip assignments, at which point the time progresses to a new epoch.

Reward An important metric for evaluating the performance of ridehailing systems is the fraction of passenger requests fulfilled. Thus, we assign a reward of 1 to each matched request, and a reward of 0 to an empty rerouting task or a do-nothing task. At the end of the simulation, we can directly compute the fraction of requests fulfilled from the cumulative reward.

2.3 Simulation details

We next mention a few details in the implementation of the simulator.

First, although the model imposes a hierarchy in the sense that there are multiple steps in the SDM of one epoch, the action implemented at any SDM step (whether it is in the middle or at the end of the current SDM) is always one of the 25 possible trip assignments. As such, it is not necessary to explicitly implement a large "action vector" for each epoch. Instead, the simulator runs an MDP whose transitions occur on the scale of one SDM step; in each iteration, action (i.e., trip assignment) is taken on only *one car*, resulting in the change of status of this particular car and possible fulfillment of one passenger request. Only upon the entrance of a new epoch are all cars moved one unit of distance forward and new passenger arrivals sampled. In other words, the simulator "flattens" the hierarchy in the dynamics constructed in the paper.

Furthermore, we make the simulation consistent with the fact that the SDM loops over the regions one by one, each time focusing on ride requests with a certain region as the origin. To achieve this, we add an auxiliary component to the state vector that tracks the region currently being considered as the "origin" in the SDM. The simulator operates under the assumption that the origin of the action (trip assignment) aligns with the current SDM origin. (This constraint is supposed to be satisfied on the agent's part, who only has the freedom to choose the destinations of trip assignments. We implement this in all of our agents, whether under random sampling or with policy training.)

Lastly, for each SDM, we maintain an "SDM clock" that keeps track of how many steps are left in the current SDM. The value of the SDM clock is initialized with the number of available cars in the system at the start of epoch.

3 Methodology

3.1 Proximal Policy Optimization

[2] uses proximal policy optimization [7] to train the centralized planner. We do not repeat their equations here. Mathematical details can be found in their original paper.

Algorithm 2: Original PPO algorithm for policy improvement in [2]

```
Result: Policy
initialize policy network with random weights $\pi_{\theta_0}$;
for each policy iteration $j = 0, \cdots, J$ do
    Run $K$ episodes of Monte Carlo simulations using current policy $\pi_{\theta_j}$;
    Obtain a stochastic estimate of value function $\hat{V}$ for the states visited in the simulation;
    Learn function approximator $V_\psi$ to $\hat{V}$;
    Use $V_\psi$ to estimate advantage for each state-action pair visited in the simulation;
    Update policy $\pi_{\theta_j}$ to maximize the PPO surrogate objective function;
end
```

3.2 Deep Q Learning

In addition to PPO, we implement deep Q learning [6] on the same environment. Q-learning is based on Bellman equation

$$Q(s,a) = r(s,a) + \max_{a'} \mathbb{E}_{s' \sim P(\cdot|s,a)}[Q(s',a')],$$

where the Q-value $Q(s,a)$ represents the expected cumulative return of taking action a at the current step and following the optimal policy afterwards. Since the expected value is hard to compute for large state and action spaces, Q-learning uses a stochastic version of the Bellman equation where the expectation is replaced by samples of s'. We then update the Q-values using some learning rate α, treating the stochastic version of the RHS of the Bellman equation as the target:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r(s,a) + \max_{a'} Q(s',a')).$$

When the Q functions are represented by neural networks, we generally seek to minimize the error between the left hand side and right hand side. Note that both expressions are *approximations* to the true Q-value, i.e., the target is not a "ground truth" baseline as in supervised learning.

For the sake of stability, we keep two networks, one for the main Q value Q_θ and one for the target $Q_{\bar{\theta}}$. At each training iteration, the main Q network is optimized to approach the one-step lookahead value predicted by the target Q network. The target is then updated periodically with the current main Q network's weights.

To increase the statistical power of the process, we use experience replay where we sample a "replay buffer" $\mathcal{B}$ randomly from the memory at each iteration. Sampling the replay buffer shuffles the experiences and decreases the sequential correlation between the transitions close to each other in history.

For $L(\cdot,\cdot)$ being some measure of difference, we have the following optimization problem for deep Q-learning:

$$\min_\theta \sum_{s_t,a_t,s_{t+1}\in\mathcal{B}} L\left(Q_\theta(s_t,a_t),\ r(s_t,a_t)+\gamma\cdot\max_{a'} Q_{\bar{\theta}}(s_{t+1},a')\right)$$

where $\bar{\theta}$ is set to θ periodically at some preset frequency. Moreover, justified by the Bellman equation, we can define an n-step version:

$$\min_\theta \sum_{s_t,a_t,s_{t+n}\in\mathcal{B}} L\left(Q_\theta(s_t,a_t),\ \sum_{i=0}^{n-1}\gamma^i r(s_{t+i},a_{t+i})+\gamma^n\cdot\max_{a'} Q_{\bar{\theta}}(s_{t+n},a')\right).$$

3.3 Neural network implementation and tuning

The state value and policy networks in PPO and the state-action value network in DQN all take state as an input. In our implementation, they share similar neural network architecture.

We follow the setup in [2] and process the "epoch" component of the state vector using an embedding layer. Embedding is a technique for representing categorical variables, with the most common example being word embedding in text processing [3]. An embedding maps each categorical variable to a point in low-dimensional space, which is a more compact representation than representing each value using one-hot encoding in high-dimensional space. We hypothesize that the role of embedding here is to "tone down" the effect of different epoch values that make otherwise similar states (in terms of cars and passengers) look very different to the neural network.

Then, we normalize the rest of the state components (using the mean and standard deviation from Monte Carlo simulations) and concatenate the normalized entries with the embedded epoch representation. This constitutes the first hidden layer of our neural network. Note that this process naturally implies the following relation:

$$\text{size of state vector - 1 + size of embedding layer = size of first hidden layer.}$$

In our implementation, the state vector has size 456 and the embedding size is 6. As a result, the first hidden layer is of size 461. We then use two more hidden layers, each of size 44 and 5, connecting them with ReLU activation functions, before outputting. We use ReLU instead of tanh for activation since the back-propagation for ReLU results in less gradient explosion. For PPO, the state value network outputs a scalar; the policy network outputs a 25-dimensional vector representing the probability of taking each action. For DQN, the state-action value network outputs a 25-dimensional vector representing the Q value at the input state and each action.

For PPO, we mainly follow [2] and use the same values for the other hyperparameters. A few details worth mentioning include:

- We changed the initialization of the model weights from random to incremental, meaning that we transferred what the model learned from the last episode to the current.
- We implemented early stopping based on KL divergence between the current policy model and the previous policy model. We found a blog post by John Schulman [4] that derived an unbiased estimator for KL divergence. Namely, a sample-based estimate of KL divergence between two distributions q and p is given by

$$\widehat{KL}[q,p]=\frac{1}{n}\sum_{i=1}^{n}(r_i-1)-\log(r_i),$$

where $r_i = \frac{p(x_i)}{q(x_i)}$ and $x_i \sim q$. We follow the way Stable Baselines3 implements PPO [8] and computes a KL divergence estimate between the previous and current policy model, and evaluate whether this estimate meets the stopping criterion used in [2].

- For policy training, we also added an L2 regularization of the policy network parameters to prevent overfitting.
- The main challenge we faced while trying to recreate the results from PPO in [2] is the large size of the simulation data. Though the problem is finite-horizon, and the storage and time complexity in theory scales linearly with the number of episodes, each simulation takes a different number of MDP iterations due to stochasticity in the dynamics, often on the order of tens of thousands of time steps per episode. For many times, our optimization runs were terminated because the system went out of memory to store the simulation data. We are aware of the tradeoff in choosing the number of episodes to simulate. On one hand, not having enough episodes results in a noisy value function estimate; on the other hand, having a large number of episodes poses a challenge for data storage, which we are seeking to resolve right now.
- We also considered improvements in learning rate scheduling. Because the default learning rate is quite small and results in a slow convergence with our model, we decided to use a step decaying scheduler. The scheduler decreases the learning rate by 0.2 after 20 steps.

  Besides, we also experimented with the following cosine schedulers: It has the effect of starting with a large learning rate that is relatively rapidly decreased to a minimum value before being increased rapidly again. The resetting of the learning rate acts like a simulated restart of the learning process and the re-use of good weights as the starting point of the restart is referred to as a "warm restart" in contrast to a "cold restart" where a new set of small random numbers may be used as a starting point.

$$\eta_t = \eta^i_{min} + \frac{1}{2}\left(\eta^i_{max} - \eta^i_{min}\right)\left(1 + \cos\left(\frac{T_{cur}}{T_i}\pi\right)\right)$$

  where η^i_{min} and η^i_{max} denotes the range of learning rates, and T_{cur} shows how many epochs have passed since the last restart.

  The intuition of the scheduler is to utilize the fact that our learning in each epoch starts from a better policy model, so setting a warm start can be helpful under the initialization. [5]

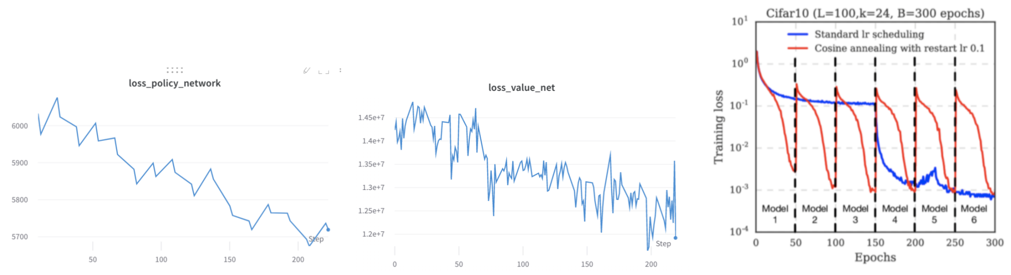

Figure 1: Loss of PPO in policy net and value net over training iterations and the illustration for the cosine scheduler.

For DQN, we use the following hyperparameters and schemes:

- We decay the learning rate for the Q-network over time. In particular, we adopt a scheme that decays the learning rate by γ every n_{LR} epochs (using `torch.optim.lr_scheduler.StepLR`). We choose $\gamma = 0.2$ and $n_{LR} = 20$.
- We use the Huber loss to measure the difference between the main and target Q-networks:

$$L(x, y) = \frac{1}{n}\sum_i z_i$$

where

$$z_i = \begin{cases} \frac{1}{2}\frac{(x_i-y_i)^2}{\beta} & |x_i - y_i| < \beta \\ |x_i - y_i| - \frac{1}{2}\beta & \text{otherwise} \end{cases}$$

We use the default value $\beta = 1$ in `PyTorch.SmoothL1loss`.

- We choose to update the target Q-network every 200 steps.
- For multi-step Q-learning, we tried both $n_{steps} = 5$ and 10.
- We kept a replay buffer of size 512, sampled from a memory of size 4096. The minimum size for the memory is set to be 1024 before a replay buffer can be sampled.
- We use ϵ-greedy, where we tried two schemes for decaying ϵ decay ϵ at step t using

$$\epsilon_t = 0.2 \cdot \exp(-t/30000)$$
$$\epsilon_t = 0.8 \cdot n^{-0.6}.$$

  The first is very conservative while the second explores more in early iterations.
- We implemented early stopping in the following way: once the loss of the main Q-network with respect to the current target Q-network is less than a threshold value, we terminate the current episode and start training on a new episode. Based on empirical observation of the training performance of the Q-network, we choose to start training on a new episode if the average loss of the last 100 iterations is below 0.02.

4 Results

Preliminary results are given in Table 1. Due to limited access to high-performance computing resources, we have not been able to conduct experiments for various hyperparameter settings. This effort is in progress as we submit this report.

Our code can be found at: https://github.com/yl3469/RL-Ride-Hailing.

4.1 Randomized actions

With the constraint that the starting region of the action aligns with the current region considered in the SDM, as discussed in Section 2.3, we first try a randomized policy that samples the action destinations randomly from the five regions. Over 100 iterations, such policy results in 54.53% request filled if empty car rerouting is allowed, and 50.18% requests filled if not.

We then implement policy networks initialized with random weights on the environment. Over 100 iterations, this results in similar performance as the random-sampling policy, as can be seen in Table 1. (We would like to explore, more generally, the theory of whether a neural network with random weights is in expectation similar to a random choice function with the same output dimension.)

Both of them serve as baselines to which we can compare the performance of our trained policies.

4.2 Policy training

Currently, we have some preliminary results in training.

As mentioned earlier, we are still struggling with data storage problem when running PPO. Our best PPO results show 53% requests fulfilled (see Figure 3).

The problem with data storage is not as serious in DQN, since we train as we go and use a memory and replay buffer of limited size. We first tried multi-step DQN with 5 and 10 steps under the first ϵ-greedy decaying scheme. The performance is much worse than the random baseline. We suspect it is due to insufficient exploration particularly in the early phases of training, where the Q networks are noisy. We then tried 10-step DQN under the second ϵ-greedy scheme. We

	Empty rerouting	No empty rerouting
Braverman OR baseline	84%	
Feng et al. performance	87%	
Randomized policy	54.53% (6.22%)	50.18% (6.44%)
Policy NN, randomized weights	53.30% (2.21%)	50.16% (0.66%)
PPO	54.32%	
5-step DQN (first ϵ-greedy decaying scheme)	16%	
10-step DQN (first ϵ-greedy decaying scheme)	23.01% (0.31%)	
10-step DQN (second ϵ-greedy decaying scheme)	60.04% (0.60%)	

Table 1: Comparison of performance: average fraction of requests filled (standard deviation).

trained for 10 episodes (with early stopping described above) and evaluated for 8 episodes. This resulted in much better performance, with about 60% of the requests filled. The improvement is presumably due to better exploration early on in the training. The loss of the Q-network over 10 training episodes and the performance of the trained network over 8 evaluation episodes is given in Figure 2.

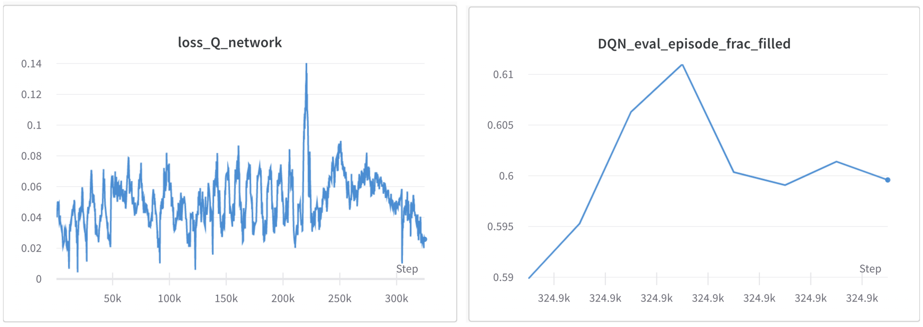

Figure 2: Performance of DQN

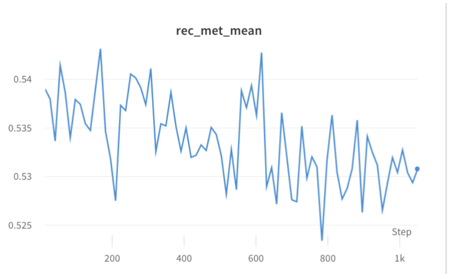

Figure 3: Performance of PPO over episodes (note that multiple episodes are shown for each single policy iteration), the best performance is around 10th policy iteration

We hope to continue conducting experiments on both PPO and DQN under different hyperparameter settings.

5 Conclusion and discussion

So far, we have understood the model in [2], implemented as a gym environment, and constructed a codebase for PPO and DQN. We are working on hyperparameter tuning to produce better training results. Due to time constraints, we did not implement many different approaches. There are a few more things we would like to explore in the future:

- Dueling Q networks [9]: This approach uses two streams, sharing the same state representation architecture, to learn the state-value function $V(s)$ and advantage $A(s, a)$ separately;

it then combines the two and outputs an estimate for $Q(s, a)$. This results from the insight that the choice of action may not matter at some "safe" states, making the estimation of $A(s, a)$ less important. The dueling structure places priority on $V(s)$ and on $A(s, a)$ for states where the choice of actions matters. It has been shown to achieve good performance on Atari games, and we are curious how it will perform in the ridehailing environment.

- Hyperparameter tuning using Bayesian optimization: Any RL algorithm involves several hyperparameters whose affect on training/evaluation performance is unclear and are difficult to tune. Bayesian optimization treats the loss as a black-box function of the hyperparameters and uses Gaussian process regression and acquisition functions to search for the optimal hyperparameter configuration(s). We would like to implement this for our RL algorithms.
- Robustness: The model makes the fairly strong assumption that the centralized planner observes everything. We wonder whether the SDM process and the policy optimization approaches will be robust to perturbations, such as when the planner observes the state with noise or only has partial observation of the state. This of course requires more modeling.

Lastly, we would like to thank our instructors, Dr. Jim Dai and Dr. Qiaomin Xie, and our TA's, Sean, Lucy, and Mark, for making the course an interesting and rewarding experience.

References

[1] Anton Braverman, Jim G Dai, Xin Liu, and Lei Ying. Empty-car routing in ridesharing systems. *Operations Research*, 67(5):1437–1452, 2019.

[2] Jiekun Feng, Mark Gluzman, and JG Dai. Scalable deep reinforcement learning for ride-hailing. *IEEE Control Systems Letters*, 2020.

[3] Cheng Guo and Felix Berkhahn. Entity embeddings of categorical variables. *arXiv preprint arXiv:1604.06737*, 2016.

[4] John Schulman. Approximating kl divergence. http://joschu.net/blog/kl-approx.html, 2020.

[5] Ilya Loshchilov and Frank Hutter. SGDR: STOCHASTIC GRADIENT DESCENT WITH WARM RESTARTS. Technical report.

[6] 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.

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[8] Stable Baselines3. Source code for stable_baselines3.ppo.ppo. https://stable-baselines3.readthedocs.io/en/master/_modules/stable_baselines3/ppo/ppo.html#PPO, 2020.

[9] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.