# Battle Royale Exploration for Card Game ZSY

**Leon Lin**

## Abstract

To learn a 2-player card game without significant recorded data, an agent must play against other agents to produce data, making its training environment different from its test environment. However, it also allows for different models to be simulated in tandem and tested against each other, improving data efficiency and exploration.

I leveraged this exploration and simulation efficiency is to try out more, different models and aggregation models. The result is an agent that seems to have acheived human level performance on the card game ZSY.

## 1. Introduction

争上游 (ZhengShangYou, or “Competition Upstream”) is a Chinese card game that is part strategy, part luck. Each player is dealt about 18 cards; they get rid of cards by matching patterns; the player who gets rid of all their cards first wins.[1] I aim to train a deep learning agent to have an above 50% win rate against humans in a 2-player version of this game.

There are three main challenges to learning this game. First, it is stochastic with a large state space. With 2 players each being dealt 18 cards, there are about 151 trillion possible initial states[2]. Second, it’s partially observed. A player can only see their own cards. Third, the test environment (playing against a human) cannot be used to train it because of the volume of data required.

[1] Full rules can be found here: https://github.com/leonl0000/ZSY2/blob/master/Writeups/ZSY%20Rules.pdf

[2] Exactly 151,632,049,354,500, calculated recursively with this code: https://github.com/leonl0000/ZSY2/blob/master/utils/one_shot_code.py

I have previously attempted this but only achieved about a 40% win rate against a human player (me). It was a rather shallow network (3 layers), fully connected, and didn’t use advanced RL methods like fixed targets. Building upon this work, there are many improvements to be tried. A CNN could be better at picking up the patterns in the cards, an RNN could be better for making use of the history of moves than what I did previously (just summing them up).

## 2. Data

The data are be obtained by having agents play against each other in a simulator I wrote.

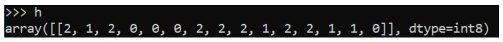
In designing the simulator, I wanted to represent the game in a way that captures its complexity without being unworkably memory intensive. During gameplay, it is useful to represent hands as counts of cards because they can be simply added to or subtracted from to represent taking an action. However, this obscures the fact that having two of a kind is fundamentally not just twice having a single: having a pair allows for different kinds of patterns to be formed.

Thus, during gameplay, the hands, the moves, and the history of moves are all represented by counts. During learning, they are converted to a stack of one-hot encodings of how many there are of each card. With 15 values of cards (ordered 3-K, A, 2, Black Joker, Red Joker), and 5 possible quantities (0,1,2,3, or 4) for each value, this results in a 5x15 array to represent any hand of cards.

See the figure below for an example of these representations.

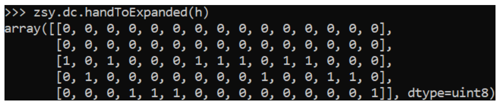
In the game, to take an action is to play some number of cards from a player's hand. So, I decided to represent a state-action pair (hand + move) as a concatenation of the remaining cards in a player's hand after a move and that move for a 5x30 array. A player can only observe what cards the opponent *has* played, not the cards the opponent has remaining. So, I represent the history of the game with two more hands of cards: the sum of the cards played by the player and the sum of the cards played by the opponent, and attached those histories to each state-action pair by concatenation into a long, 5x60 array.

*A hand of cards*



```
>>> h
array([[2, 1, 2, 0, 0, 0, 2, 2, 2, 1, 2, 2, 1, 1, 0]], dtype=int8)
```

*1x15 array representing the counts of each value of card*



```
>>> zsy.dc.handToExpanded(h)
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0],
       [0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1]], dtype=uint8)
```

*5x15 array with each column being a one-hot encoding of whether there are 0, 1, 2, 3, or 4 of a card*

### 2.1. Replay Buffer

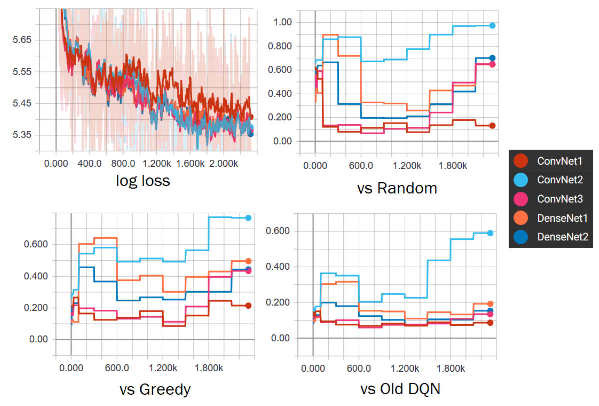
In the previous project, I made several models with different hyperparameters and had them running against themselves: they would train for some number of epochs on 100,000 games of data, then dump that data and simulate another 100,000 games against themselves, and repeat. After each simulation, they would test themselves against a random and a greedy agent (that I describe in more detail in section 3), and I periodically manually killed off the worst performing models based on those tests.

That was highly inefficient. This time, I have made a replay buffer inspired by Mnih et al. (2013). With a maximum storage 250k games and each game lasting ∼25-35 turns, the buffer maxes out at about 7 million data points. Instead of having each model having its own buffer, many models are initialized and play against each other to contribute to the same buffer. As these simulations continue, I'll eliminate the worse performing agents until only a few remain. This kind of "battle royale" approach takes advantage of the PVP nature of the game to combine hyperparameter/network structure search with increased efficiency for the simulations. On the one hand, this does introduce more off-policy learning to the agents; on the other, it's another method of exploration and it allows the amount of available data to scale with the number of agents: I would not have been able to try out nearly as many different kinds of agents if they each needed their own replay buffers.

## 3. Experiment 1: Buffer Size

At first, I made the buffer to store 1 million games played by a random agent, but after testing that out on two basic agents I found that the models' win-rates decreased significantly after 2 epochs, and further that they performed better when trained on only a 250k subset of the data.

To analyze this more formally, I instantiated 5 agents (2 fully connected and 3 with convolutional layers), broke up the 250k games into 2.4k batches of 4096 points, trained the agents over those batches, stopping at batches $0, 1, 3, 10, 100, 300$ and every 300th batch after that to test the agents performances against 3 static agents.



*Exp. 1: Buffer Size. Losses plotting with .9 smoothing*

As games are episodic with bounded rewards, I decided to use Monte Carlo gains as the targets for the Q network. That is to say, each 5x60 state-action-history pair is assigned a value of 1 if it resulted in a victory for the agent and a 0 if it did not. Losses are log losses against these values.

Unlike Mnih et al. (2013), I decided to use feed each state-action pair into the network to get a Q value. This is because, unlike with the Atari games, the action space is enormous, variable, and discrete. With the different patterns available to play, there a total of 9075 possible actions in the game, with typically 5-30 of them being available on any particular turn.

The 5 agents were either fully connected or CNNs. The Dense nets have 3 layers of size 300, 40, and 1 while the Conv nets had 2 layers of 3x3 valid convolutions (32 in the first and 64 in the second) followed by a size 100 and 1 fully connected layers. The activations were mixes between ReLu and Leaky ReLu, and the output layer had a sigmoid activation. The full definition of each specific network can be found in the configs.csv files for each experiment in the github repo.[3]

All training was done with the TensorFlow's implementation of the Adam Optimizer with no change to its default values. The three agents it was tested against are:

- The **Random Agent** takes completely random moves
- The **Greedy Agent** always tries to get rid of its lowest valued cards as quickly as possible, never passing if it can take a move.
- The **Old DQN Agent** is the agent from the previous project, weights frozen.

Each test was with 1000 simulated games. Treating the true winrate as a binomial variable, the maximum standard deviation for the sampled winrate across 1000 games is $1.58\%$.

For most of the agents, the winrate grew very fast at first and levelled off around a quarter of the data. One agent did have a significant growth in winrate near the end, but others did not. As such, I decided to initialize the replay buffer with 100k random games and set the maximum to 250k games.

## 4. Experiment 2: Round Robin

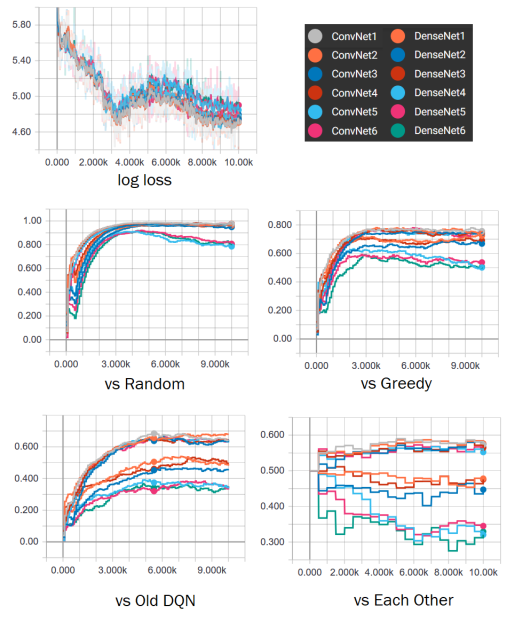
As a stepping stone to the battle royale idea, I needed to get a sense of just *how* different different agents were to set the thresholds of killing of models.

In my second experiment, I initialized 12 agents, 6 DenseNets and 6 ConvNets. The DenseNets had 3-4 layers; the ConvNets each had 2 layers of 3x3 valid convolutions, some had a third layer that was a 1x3 valid convolution (as the size of the 5x60 input was reduced to 1x56xC in the previous two layers), and some had an additional 1x1 convultion layer before the two dense layers. As before, the full definitions of each model can be found in the configs file in the Github repo. Each model tracked 4 statistics in addition to their weights: their winrate against Random, their winrate against Greedy, their winrate against the Old DQN, and their average winrate against each other (hereon called the ”VS” winrate for brevity). Each epoch consisted of a training phase and a simulation phase.

[3]https://github.com/leonl0000/ZSY2/tree/master/Experiments

**Algorithm 1** Round Robin Algorithm

```
Init buffer with 100,000 random games
Init $A$ with 12 agents
for e in NumEpochs do
  for i in range(500) do
    sample = random 4096 points from buffer
    for Agent ag in $A$ do
      Train Agent ag sample
      if i%100 == 0 then
        Test Agent ag for 100 games against the 3 static agents
        Update Agent ag's winrate against the 3 static agents
  for i in range(500) do
    Agent ag, ag' = 2 random agents from A weighted by their "VS" winrates
    Simulate 100 game between ag and ag'
    Add new games to buffer
  for Agent ag in A do
    Update ag's "VS" winrate
```

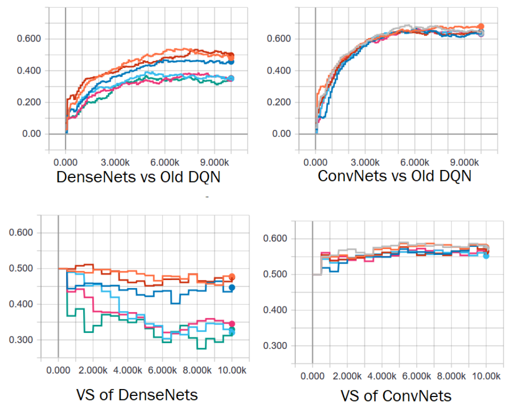


*Exp. 2: Round Robin. Losses shown with .9 smoothing*

In the training phase, 500 samples of 4096 points are taken from the buffer without replacement, and each model is trained on those points. With 250k games in the buffer, this means that on average 8 points are taken from each game (which have 30 points, one from each turn in the game). At each 100 iterations, the models were tested against the static models, and their winrates were updated exponentially with $\alpha = .9$.

In the simulation phase, 500 sets of 100 games are played, for a total of 50k games, replacing 1/5 of the total buffer. For each set of 100 games, 2 agents randomly selected based on the VS winrate, and after all the games are played their VS winrates are updated exponentially with $\alpha = .5$.

What was immediately noteworthy was how the ConvNet Agents performed significantly better than the DenseNets. See figure 6 for the VS winrates separated by the DenseNets on the left and the ConvNets on the right.



*Winrates separated by kind of agent*

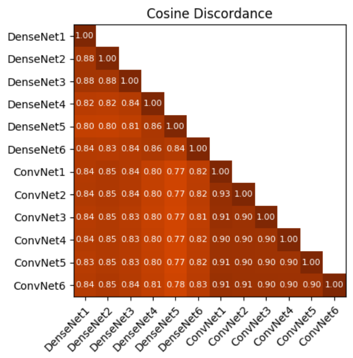
By every metric, all of the ConvNet Agents were better than all of the DenseNet Agents. As this game is heavily dependent of patterns of cards, this seems to confirm my hypothesis that a ConvNet would be good a picking up on those patterns.

Another thing of note is that, though the winrates were very different, the losses were almost identical: fitting the data well is not a great indicator of how the model would perform in practice, even though the data was refreshed every epoch. Also, several agents had declining winrates near the end, suggesting some kind of overfitting. Every ConvNet Agent ended up beating the old DQN, all around epoch 4 of 20. Two of the DenseNet Agents did, but it was during the middle of their run, and then their winrates dipped below 50% by the end.

## 5. Experiments 3 & 4: Cosine Discordance and Aggregation Models

While the DenseNets had quite differing winrates, the CNNs have such similar winrates against each of the static agents. That raises the question: do they have the same policies? Even though they have different structures and different initializations, they were trained on the same data and had very similar losses, so it's entirely possible that the networks are functionally identical.

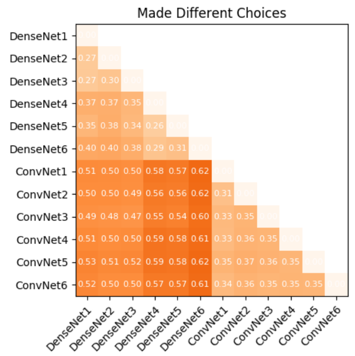
To test this, I ran an experiment to find a metric hereto referred to as "Cosine discordance". I ran a round robin simulation simulation for all of the agents: in each game, only the pair of agents playing made the final decisions, but every agent recorded their Q values for every action for every state. Then, for all states that had more than one legal action, I found cosine similarity between the vectors averaged them. Additionally, I recorded the proportion of instances that two agents would have made different choices (i.e. that action with max Q value was different). The results were suprising: even though the final test performances were similar, the choices they made were different, indicating that they were making different kinds of mistakes.



*Cosine similarity between the vectors of Q values*

*Table 1.* Final results of some Round Robin models + Aggregate models

| MODEL | AVERAGE WINRATE |
|---|---|
| MIN6COMBOAGENT | 0.652 |
| MEAN6COMBOAGENT | 0.649 |
| CONVNET2 | 0.546 |
| CONVNET1 | 0.535 |
| ... | |
| MAX9COMBOAGENT | 0.478 |
| OLD DQN | 0.477246 |
| DENSENET3 | 0.462029 |
| DENSENET1 | 0.453333 |



*Portion of times the top valued action differed between 2 agents*

The ConvNets were more similar to each other than the DenseNets were, and the ConvNets as a group were more different from the DenseNets as a group than from each other. Even though the Conv nets were more similar, they still made different choices from each other about a third of the time.

This suggests that the agents are learning different policies and that there is potential merit in a aggregator model. So, I made 3 kinds of aggregators: using the minimum Q value as from several models, the average Q value, and the maximum Q value, and tried testing them with the top 3, 6, and 9 models. See Table 1 for the results of a 300 games/match-up round robin of the 12 trained agents, 9 aggregator models, and 3 static agents, and Table 2 for the winrates of the agents against the static agents.

*Table 2.* Top performance of each kind of model

| TOP MODEL VS | RANDOM | GREEDY | OLD DQN |
|---|---|---|---|
| MIN6COMBOAGENT | 0.993 | 0.773 | 0.690 |
| CONVNET2 | 0.956 | 0.743 | 0.623 |
| DENSENET3 | 0.933 | 0.643 | 0.440 |

The aggregation made shockingly large improvements to the winrates of the agents, with an around an 5% gap between the best aggregating agent and the best single agent. Only one of the aggregator models (Max9ComboAgent) performed worst than some non-aggregator models.

## 6. Experiment 5: Battle Royale

Another takeaway from the Round Robin experiment was that the winrates for various models were different, but not so wildly different that using those to weight the model sampling for simulation probably doesn't make much difference. For this final experiment, I doubled the number of agents to 24 (running 2 seeds of each original agent) and started with the same 100k random games buffer as before. Based on the results of the Round Robin experiment I defined these two formulas for exploration and killing off agents:

$$\epsilon = \frac{1}{ep+2} * \sqrt{\frac{|A_0|}{|A|}}$$

$$\text{Kill\_Threshold} = 0.5 * (1 - .75^{ep * \frac{|A|}{|A_0|}})$$

Where $ep$ is the epoch number (starting from 0), $|A|$ is the number of remaining agents and $|A_0|$ is the initial number of agents. Exploration decreases as the number of epochs increases, but also increases slightly when there are fewer agents. The Kill Threshold starts at 0 and exponentially approaches 50% as the number of epochs increase, but also decreases as there are fewer agents remaining. Any agent with a VS winrate below the Kill Threshold is killed. Each time an agent is killed, the VS winrates of the remaining agents drops because the remaining agents are tougher opponents.

In this experiment, I only tested the agents against static agents at the end of every epoch and did not weigh the number of games simulated between them by their VS winrates.

**Algorithm 2** Battle Royale Algorithm

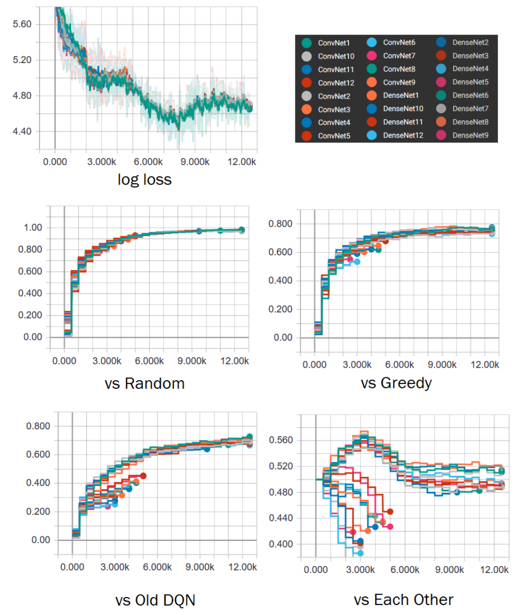
```
Init buffer with 100,000 random games
Init $A$ with 24 agents, 12 with duplicate models
for e in NumEpochs do
    $\epsilon = \frac{1}{ep+2} * \sqrt{\frac{|A_0|}{|A|}}$
    Kill_Threshold $= 0.5 * (1 - .75^{ep * \frac{|A|}{|A_0|}})$
    for i in range(500) do
        sample = random 4096 points from buffer
        for Agent ag in $A$ do
            Train Agent ag sample
    for i in range($|A|$) do
        for j in range(i, $|A|$) do
            Simulate $50000 / \frac{|A| * |A| + 1}{2}$ games between
            $A_i$ and $A_j$
            Add games to buffer
    for Agent ag in $A$ do
        Update ag's "VS" winrate
        Test Agent ag for 200 games against the 3
        static agents
        Update Agent ag's winrate against the 3 static
        agents
        if ag's VS winrate < Kill_Threshold then
            Remove ag from $A$
```

*Table 3.* Final test of Battle Royale models with Aggregate models

| MODEL | AVERAGE WINRATE |
|---|---|
| MIN6COMBOAGENT | 0.695 |
| MIN3COMBOAGENT | 0.695 |
| ... | |
| CONVNET2 | 0.574 |
| CONVNET3 | 0.573 |
| ... | |
| CONVNET11 | 0.530 |
| OLD DQN | 0.4640 |
| DENSENET3 | 0.436 |
| DENSENET5 | 0.435 |
| ... | |
| GREEDY | 0.336 |
| DENSENET11 | 0.333 |

*Table 4.* Top of each kind of Battle Royale model + Aggregate models

| EACH MODEL VS | RANDOM | GREEDY | OLD DQN |
|---|---|---|---|
| MIN6COMBOAGENT | 0.978 | 0.777 | 0.754 |
| CONVNET2 | 0.966 | 0.726 | 0.628 |
| DENSENET3 | 0.962 | 0.617 | 0.426 |
| DENSENET9 | 0.896 | 0.511 | 0.244 |



*Results of experiment 5, 10 agents remained at the end*

The top model from the Battle Royale experiment only slightly beat of the top performing model in the Round Robin experiment, but the aggregator model was very significantly better. Likely, this is due to there being more models to aggregate from.

## 7. Playing Against Humans

In parallel with the Round Robin experiments, I built a simulator with a GUI component in Unity in order to test these agents against humans. At the time of releasing the game, I had only run the Aggregator experiment for 100 rounds and the found that the ComboAgent using the minimum Q of the top 9 agents performed the best, so I sent out the game with that model. At the time of writing this report, I have gotten the following gameplay results back, highest to lowest scoring human, in Table 5.

It is an simple game to learn for humans, with one player commenting ”the tutorial was SUPER helpful and it’s very easy to pick up!!”. The top result is from a player who has played ZSY for a number of years, but the second result is from one who had only just started. The third is me, and the rest were new to the game. The is available to play against on PC and Mac right now[4]

One thing to note that may skew these results is that humans also learn. Player 1, for example, commented that she lost more to the Agent in the beginning than at the end.

[4]PC build : https://drive.google.com/file/d/1auOngmTju8Pg8WQKNkGNUvjCXRWxkJOA/view?fbclid=IwAR18pKu3wfDANUOuBwu8d6ozCdQSf1V8C7v9_bL5lqGMwP8AGO0nQs7D-lg
Mac build : https://drive.google.com/file/d/1ZQ_-JJ__go3fGu9P5BJSzQ3dYoFxi0rY/view?fbclid=IwAR13No7aV-8KxD9XrLvxBW6z3uXUSghWqXsuYzit_wKFdt0z4ZanSt6sm_k

*Table 5.* Winrates of humans against the Min9ComboAgent

| PLAYER | WINS : LOSSES | WINRATE | $\sigma$ |
|---|---|---|---|
| 1 | 27:32 | 45.8% | 6.5% |
| 2 | 29:35 | 45.3% | 6.2% |
| 3 | 16:24 | 40.0% | 7.5% |
| 3 | 12:20 | 37.5% | 8.6% |
| 4 | 20:36 | 35.7% | 6.4% |

## 8. Conclusion and Next Steps

Without more widescale testing, I cannot say for certain that the Agent has achieved human level performance. The variance in the data is huge, but the results look promising. The previous DQN beat me 40% of the time. For this agent, the ratio was reverse. Further, as it is not a very widely played game, there is no true consensus on what is an 'expert' player.

However, with the various methods I attempted in this paper, I made significant improvements to the performance of the previous model, beating it roughly 3:1. Probably the two most important factors were adding a conv net model and an aggregator model. I expected a conv net to be more effective, but the sheer gap between that and the fully connect models was a suprise. The aggregator model was a complete surprise in how much of a difference it made: I assumed they may raise the winrates by 1 or at most 2 percent, but they made such a huge difference.

It's hard to say how much the Battle Royale method for exploration helped the models actually explore more options, and it certainly seems, from experiment 2, that models can still overfit in some way even with the exploration. But, it did help try out many more different models than I otherwise would have. In experiments 2 and 5, even though training involved many models and the simulation only involved 2 at a time, the simulation stage of each epoch still took longer than the training phase. If not for this method of pooling simulation results, I would not have had enough different models to try an aggregator model with.

There are still many things I wanted to try that I didn't get the chance to because I prioritized the things I tried in this paper. First of all, I rewrote the simulator to store each data point in the history instead of just the summed histories and had the agents sum the histories on the fly. This was because I wanted to try out an RNN agents that didn't just sum over the histories but treated each move an an individual event, but I did not write an RNN agent. Secondly, the Unity game was written at the very end and I didn't write in code to send game results to a server: players just manually reported to me their scores. With this game written, it is possible to collect human data to potentially train against. Also, with the game written, it would be possible to better visualize different tricky games by replaying them visually rather than looking over a bunch of numbers representing cards.

Finally, though simulation takes much longer than training, the GPU usage is much lower, so potentially it would not increase simulation time too much to use an aggregation model for simulation to get better simulation results. But since Experiments 4 and 5 were written and run in parallel, I didn't have the chance to apply what I learned about the aggregator models to the Battle Royale.

## 9. References

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning.