Playing Geometry Dash with Convolutional Neural Networks

Ted Li
Stanford University
CS231N
tedli@cs.stanford.edu

Sean Rafferty
Stanford University
CS231N CS231A
seanraff@cs.stanford.edu

Abstract

The recent surge in deep learning has resulted in a significant improvement in artificially intelligent game-playing agents. Combined with the recent improvements in computer vision, various game-playing agents that have human-like or better performance have been created. In this paper, we will design and train various AI to play Geometry Dash, a rhythm based action platformer that revolves around the player navigating through a side-scrolling stage filled with dangerous obstacles. While the actual gameplay is relatively simple and the types of obstacles are limited, Geometry Dash's main challenges come from being able to recognize and react to different patterns and variations of the simple obstacles and react accordingly in very short windows. We present our findings and the advantages and disadvantages of different models and variations.

1. Introduction

Geometry Dash is a rhythm based action platformer that revolves around a player navigating their block through a side-scrolling stage filled with obstacles. Players control their block as it travels through a stage by timing their jumps. Depending on the state of the block, the jump input may cause the block to jump, double-jump, invert gravity, or do nothing. By using visual and audio cues, players must then correctly time the use of their jump button in order to travel to the end of a level. Without crashing or touching any spikes.

Our project focuses on designing an artificial intelligent game-playing agent for Geometry Dash. Using only screenshots of the game as our inputs, we create a model that can play through a game, similar to how a human would play the game. In this paper, we present our approaches, problems encountered, and the success of our different models.

2. Background/Related Work

2.1. Reinforcement Learning

Reinforcement learning is a framework which consists of an environment and an agent [9]. The agent interacts with the environment by performing actions which mutate the current state and yield an immediate reward. It is then the task of the agent to learn how to maximize the discounted long-term reward by taking actions at each step. This optimization typically that the agent be able to observe the state in some way. In Atari games, the observed state is typically raw frames from the game [4, 5]. This is intuitive to us since this is also the state that we observe as humans. However, raw frames are not the only way to observe state; for instance we observe sound from the game whereas the agent does not. Furthermore, Atari emulators can also supply the RAM instead of images from the game, which is a completely unintuitive but potentially useful way to observe state [1]. Furthermore, state for a particular environment may or may not be fully-observable. For instance, if I take a picture of a bouncy ball in the air and show it to you, you may not be able to tell if it is falling down or bouncing up into the air. DeepMind converted these types of partially-observable games to fully-observable games by stacking multiple (grayscale) frames together as inputs. This effectively converts the color channel into a time channel and allows the agent to learn time-based features such as velocity. Recently, Hausknecht and Stone have used recurrent neural networks on Atari games with similar success [2].

There are many approaches to reinforcement learning [4, 4, 3], but they all share a common mathematical formulation of how the agent interacts with the environment. The environment is modeled as a Markov Decision Process (MDP); there is a set of states and a set of actions, and performing a certain action in a certain state yields a probability distribution of transitions to other states and a probability distribution of rewards. If the state is not fully-observed then this MDP becomes a partially-observed MDP (POMDP), in which we propose a probability distribution of the states we could be in instead of knowing for sure. We

will not discuss MOMDPs and instead assume they have been converted to MDPs. Given an MDP we can calculate the discounted long-term reward of performing a particular action in a given state as follows:

$$Q(s,a) = E_{s',r\sim a}\left[r + \gamma \max_{a'} Q(s',a')\right] \tag{1}$$

Where s is the current state, a is the proposed action, s', r are the resulting state and reward respectively, $\gamma \in [0,1]$ is the discount and controls how much we care about the future, and a' varies over all the actions we could take in the resulting state. This recursive definition is known as the Q-value.

Reinforcement learning is an attractive approach as it allows the agent to learn from environment directly without any supervision.

2.2. Q-learning

2.2.1 Overview

In Q-learning we learn to approximate the Q-function given above. Note that if we had access to the true Q-function for our game then we could always perform optimally in our game by always choosing the action that maximizes the Q-function for the current state. This is a model-free approach; we learn the optimal action for a given state without learning the explicit mechanics of our game (we do not define the MDP). We use a collection of experiences to update our Q-function. Each experiences consists of the initial state, the action taken, the resulting reward, and the resulting state (s, a, r, s'). We then generate a "true" Q-value (q) and the estimated Q-value $(\hat{q})$ and compute the difference between them, known as the temporal difference error (δ). Finally, we pass the temporal difference error to a loss function (typically Huber to clip gradients) and then backpropogate the gradients.

$$q = r + \gamma \max_{a'} Q(s',a') \tag{2}$$

$$\hat{q} = Q(s,a) \tag{3}$$

$$\delta = |q - \hat{q}| \tag{4}$$

2.2.2 Double Q-learning

The estimates provided by Q can be noisy and optimistic. To combat this we will use *double Q-learning* [10]. We implement this by keeping a frozen copy of the *online Q-function* k called the *target Q-function* which copies weights from the periodically. We use the online Q-function to choose the best action to take when computing the Q-value on the right-hand-side of q but use the target Q-function to compute the Q-value of this action. This takes the following form:

$$q = r + \gamma Q_{target}(s', \arg\max_{a'} Q_{online}(s',a')) \tag{5}$$

$$\hat{q} = Q_{online}(s,a) \tag{6}$$

2.2.3 Dueling Q-function

Although we are learning the Q-function, we are primarily interested in the advantage that a particular action could give us in any given state. In the current formulation, the state value and action-advantage value are combined and there is no explicit sharing of a state-value between the many actions. We will remedy this with a dueling Q-function [11]. A naive formulation is as follows:

$$Q(s,a) = V(s) + A(s,a) \tag{7}$$

However, since A is an advantage it should sum to zero over all actions for a particular state. We formulate A as follows to force this property:

$$A(s,a) = \tilde{A}(s,a) - \frac{1}{|A|}\sum_{a'} \tilde{A}(s,a') \tag{8}$$

2.2.4 (Prioritized) Experience Replay

Recall that we earlier state that we update the Q-function using collections of experiences (s, a, r, s'). Until now we have not discussed how we obtain this collection. While playing the game we collect a *replay memory* of past experiences. Every few steps we take a few batches from this replay memory and train on it. This is known as *experience replay*, and can be thought of as a dataset for our game that is automatically collected and update [4, 5]. Experience replay is crucial for efficiently learning from observations.

Note that some experiences in the replay memory will be rare and useful examples for our model while others may be useless frames or states we have already learned how to solve. Rarely sampling the useful examples and commonly sampling useless frames will reduce or training speed. If we had some sort of value of usefulness for each example then we can use weighted sampling with this value to increase training speed. This is the motivation behind prioritized experience replay [7], which has been shown to increase training speed in most cases. The metric for usefulness is the temporal-difference error (δ) and can be interpreted as how surprised the model is by a certain training example.

2.3. Asynchronous Advantage Actor-Critic (A3C)

Asynchronous advantage actor-critic (A3C) methods are the current state-of-the art for reinforcement learning [3]. A3C is one the best methods for parallelizing the training of reinforcement learning agents. In these methods, many

workers play the game simultaneously and update a central parameter server periodically. This allows for scalable parallelism as well as increased exploration which has been shown to greatly decrease training time. Unfortunately, this method is incompatible game with the framework for Geometry Dash and would require significant implementation time to enable so we will not be pursuing it.

2.4. Imitation Learning

Imitation learning is another approach commonly used for the task of playing games. Unlike reinforcement learning, which typically has a period of exploration and then learns its optimal policy, imitation learning begins with a near-optimal strategy provided by an *"expert"* that it learns to imitate. While imitation learning hasn't shown as much success as reinforcement learning models and their variants, imitation learning models are typically simpler, faster to train and converge, and easier to tune. However, since the states in Geometry Dash depend on previous states and actions, the data and labels provided for our imitation learning models are not I.I.D., and as such, the model may struggle with robustness [6]. Because there are so many combinations of obstacles in Geometry Dash, this can cause issues since small changes in the state and action space may lead to even more divergent states that our expert has never trained on and thus the agent has not seen before. One way that agents have overcome this is by using imitation learning simply to pre-train weights for a model before it is fed to a Deep-Q Network, such as the case of AlphaGo [8].

3. Gameplay Framework

Unlike the Atari games commonly used in modern reinforcement learning research [4, 5], Geometry Dash does not have an emulator that allows us to easily interface with the game and run it at discrete time-steps. Since we wanted our model to be compatible with an OpenAI Gym environment, we had to implement a *step* function which would take an action, emulate a step in the game, and return the resulting state, reward, and whether our state is terminal. Additionally, this would allow for us to have higher precision with our predictions and avoid synchronization problems.

3.1. State-based Gameplay

In order to allow for this, we made the game discrete by injecting a DLL into the application which causes all Windows API calls that return a time to instead return a fabricated timestep that increments the time by the inverse frame rate every step. To input actions and capture screenshots of frames, we used traditional Windows API calls. To detect terminal states, we checked for the game over overlay. All of this functionality was then wrapped into a python library so that it could be used easily.

3.2. Data Collection

Using this framework, we collected data in two ways. For training our reinforcement learning agents, we collected the frames of our gameplay as we trained our models and also used them for prioritized experience replay. Since our reinforcement learning agents were constantly learning, their actions would change and as such, we could constantly be in different states, collecting multiple playthroughs of each level where each playthrough ended when the in-game state was terminal. When training our imitation learning model, we only collected one playthrough for each level. Since we're training an agent to play like an expert, we simply recorded an expert level playthrough of each level in the game and used that data to train our agents. In order to collect the data, we again discretized the gameplay, but also kept track of what input we were feeding to the game during each frame. This raw data was then preprocessed and fed to our imitation learning models.

However, one major drawback that resulted from our data collection method, was that the built-in anti-cheat mechanisms present in Geometry Dash online levels prevented us from using our environment. As such, we were limited to only the default levels that come with the game.

4. Approach

For both of our models, the framework was similar. We start with raw pixel data extracted from the game as a screenshot. These images are preprocessed and then fed into our convolutional layers in order to generate features. These features were then either fed into the deep reinforcement learning agent or the imitation learning agent.

4.1. Image Preprocessing

In order to avoid losing valuable features and allow for our model to be as end-to-end as possible, we minimized the image preprocessing done for our models. For all of our images, we downscaled our images by a factor of 4. Afterwards, we sliced the sides of our image off so that the final image would be a square. Thus, the overall resize took our images from $[720, 480]$ to $[120, 120]$. We also grayscaled our images in order to reduce the parameters and also simplify the model. Another step of preprocessing was the in-game textures for many of the blocks. These textures have no impact on the game and provide no extra information for the agents, and thus would only slow down the training of our models.

4.1.1 Input Windows

An interesting aspect of Geometry Dash is the variable successful input windows. Depending on the size and shape of the obstacles, the window to input a jump can be any-

where from 2 to 10 frames. However, if the input is a single frame outside of the correct window, it will result in a terminal state. This causes some problems with our imitation training model. If an expert is asked to jump over the exact same obstacle twice, even if they play the game frame-by-frame, it is possible that they do not jump from the exact same position each time - in fact, based on the song's speed and the game's frame rate, it's possible that the player's square will never be the same distance from two identical obstacles. As such, this leads to discrepancies in our expert training model. For example, if an expert jumps while 12 pixels away from an object, but not at 10 or 14 pixels away, and later jumps over the same obstacle, but at 14 pixels away and not at 12, we now have two identical input images (states) with opposite actions taken by the expert. To avoid this issue, we remove the 10 frame window around each action that the expert chooses. While this can lead to some issues with precision during gameplay, it helps the model avoid training on inconsistent data.

4.1.2 Past Timesteps

We also experimented with concatenating multiple timesteps of our model in give the agent information about past states.

4.1.3 Image Augmentation

Because we were limited by the amount of data we had access to, we augmented our expert-training data. The designs of all Geometry Dash levels come from black-and-white textures for platforms and obstacles. These textures are then overlaid with colors that constantly change during gameplay. While these colors don't add much to the actual gameplay, they do add another hurdle for our model to train on, and as such, we artificially changed colors in our data and trained on those modified images as well.

4.2. Feature Generation

For our Feature Generation, we used convolutional neural networks. The layers are as follows:

1: `INPUT`: $120 \times 120 \times 1$
2: `CONV2D`: 8×8 size, 4 stride, 32 filters
3: `RELU`: $\max(x_i, 0)$
4: `CONV2D`: 4×4 size, 2 stride, 64 filters
5: `RELU`: $\max(x_i, 0)$
6: `CONV2D`: 3×3 size, 1 stride, 64 filters
7: `RELU`: $\max(x_i, 0)$

We also experiment with adding batch normalization layers in between our convolutional layers. These features are then flattened and fed into either the reinforcement learning model or the imitation learning model.

4.3. Reinforcement Learning

We used a Deep Q-network (DQN) with dueling Q-layers, double Q-functions, and prioritized experience replay. The dueling Q-layers are as follows:

State value (V)

1: `INPUT`: Flattened features
2: `DENSE`: 256 features
3: `RELU`: $\max(x_i, 0)$
4: `DENSE`: 256 features
5: `RELU`: $\max(x_i, 0)$
6: `DENSE`: 1 feature

Action advantage (A)

1: `INPUT`: Flattened features
2: `DENSE`: 256 features
3: `RELU`: $\max(x_i, 0)$
4: `DENSE`: 256 features
5: `RELU`: $\max(x_i, 0)$
6: `DENSE`: 2 feature

Q-value (Q)

1: `DUEL` $V(s) + A(s, a) - \text{mean}_{\hat{a}} A(s, \hat{a})$
2: `ARGMAX`

We implemented an OpenAI gym Env wrapper for Geometry Dash so that we could use the OpenAI baselines reinforcement learning framework [1] [1]. Doing so significantly de-risked our approach and allowed us to focus on particular issues associated with Geometry Dash and how we could augment our model to solve them, rather than on the implementation and debugging of a framework. We do not use any exploration in our training. Geometry Dash is interesting in that there are only two actions at every step. Since we are using a binary softmax classifier, negatively reinforcing one behavior implicitly positively reinforces the other. Thus, exploration is unnecessary. Geometry Dash is also interesting in that you must solve a consecutive series of obstacles to progress through the level. Hence, a small chance of failing a certain obstacle quickly compounds and makes it incredibly likely that you would have failed after a certain number of obstacles, even if your model was otherwise perfect.

1: initialize replay memory M to capacity N
2: initialize Q_{target}, Q_{online} with random weights
3: choose a level in the emulator
4: **for** episode in episodes **do**
5: restart the level
6: s := initial state
7: **while** s is not terminal **do**
8: $a = \max_a Q_{online}(\phi(s), a)$
9: execute a and receive image s' and reward r

[1] Code at `https://github.com/openai/baselines` was used as a starting point for reinforcement learning. The environment for Geometry Dash was built by us

if s is terminal **then**
$q = r$
else
$q = r + \gamma Q_{target}(s, \arg\max_{a^*}(s, a^*))$
end if
$\hat{q} = Q_{online}(s, a)$
$\delta = |q - \hat{q}|$
store (s, a, r, s') in M with weight δ
weighted sample minibatch (s, a, r, s')
if s is terminal **then**
$q = r$
else
$q = r + \gamma Q_{target}(s, \arg\max_{a^*}(s, a^*))$
end if
$\hat{q} = Q_{online}(s, a)$
$\delta = |q - \hat{q}|$
perform gradient descent on $Huber(\delta)$
end while
end for

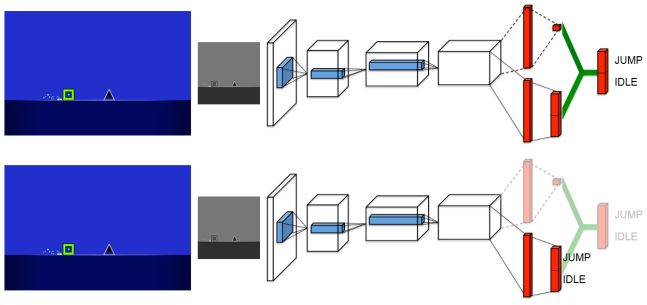
4.4. Imitation Learning

Our imitation learning agent was modeled as a classification task. The final output is determined from a softmax followed by argmax over the two possible actions. The layers are as follows:

1. `INPUT`: Flattened features
2. `DENSE`: 256 features
3. `RELU`: $\max(x_i, 0)$
4. `DENSE`: 256 features
5. `RELU`: $\max(x_i, 0)$
6. `DENSE`: 2 features
7. `SOFTMAX`
8. `ARGMAX`

Note that this is essentially the same as the action advantage layer from our reinforcement learning model.

Figure 1. Deep Q-network (upper) and classifier (lower)

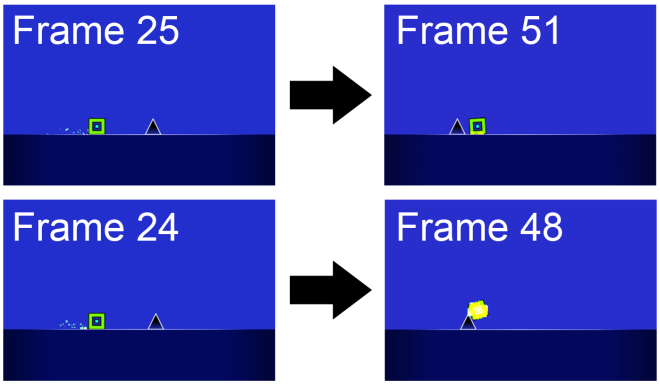


5. Experiments

6. Reinforcement Learning

We allowed our agent to play overnight with several different reward structures. We first tried rewarding -100 on collision and 0 on all other frames. The model quickly learned to repeatedly jump and pray that it would clear the obstacles and receive points. We observed that the inputs to the game are relatively sparse; we should really only be jumping once per obstacle and otherwise idle. To encourage this, we provided a reward of 1 for idling, a reward of 0 for jumping, and a reward of -100 for colliding with objects. The agent jumped only when necessary and learned to clear a few obstacles but eventually got stuck.

Figure 2. Jumping one frame earlier causes a collision



We hypothesize that the biggest issue with our game is in discriminating between two extremely similar frames that require different inputs to clear an obstacle. For example, if we jump in frame 25 we successfully clear the obstacle by frame 51. However, if we jump just a frame earlier in frame 24 we will collide with the obstacle. From frame 24 to frame 25, the cube moves just 8 pixels closer to the obstacle. Since we scale each frame down by a factor of four, this difference becomes just two pixels. The window in which a jump is valid is 12 frames, or 96 pixels (24 pixels when scaled down). Note that this is the simplest example and there are obstacles in the game which have much smaller windows (a few frames).

Reinforcement learning further complicates this issue. We observed that when an agent jumps too early, it will tend to jump even earlier and earlier. Consider what happens with the reward signal when the agent jumps too early and collides with an obstacle. The negative reward propagates back to the frame where the agent jumped too early and negatively reinforces that behavior. However, that reward keeps propagating back to even earlier frames, negatively reinforcing the good behavior of waiting. Given the sparse and binary nature of credit assignment in our problem, we wondered whether the reinforcement learning framework was not suitable for our task. This was the motivation of experimenting with supervised learning using mostly the same model. If architecture works in the classification framework, then the model is powerful enough and there is an issue with the reinforcement learning framework. Otherwise, the model is not strong enough and would need

modification.

7. Imitation Learning

In order to train our model, we used the Adam Optimizer with a learning rate of $1e-3$ and calculated loss using binary cross-entropy. Because most of the game is spent not pressing the jump button, our classification labels were very biased. In order to make sure our model learns to jump, we weighted the loss on jump labels. In nearly all of our levels, approximately 2% of the level is spent jumping, so we increased the loss on all jump predictions by 50 fold.

Because level 1 was the simplest of all of the levels, we trained our model on levels 2-10 and then validated on level 1. We used the default model shown above and trained for 2 or 5 epochs. In addition, we used augmented data for the weighted and sliced model, where we swapped the R, G, and B channels before converting to grayscale, in order to simulate different colored backgrounds.

Table 1. Default Parameter Experiments

Weighted	Sliced	Augmented	Dropout	Epochs	No Action %	Jump %	loss
No	No	No	No	2	1.00	0.00	0.084
Yes	No	No	No	2	0.98	0.13	2.648
Yes	Yes	No	No	2	0.86	0.80	1.222
Yes	Yes	Yes	No	2	0.90	0.77	0.617
Yes	Yes	Yes	Yes	2	0.97	0.60	1.938
No	No	No	No	5	0.98	0.20	0.120
Yes	No	No	No	5	0.99	0.10	3.920
Yes	Yes	No	No	5	0.98	0.63	2.431
Yes	Yes	Yes	No	5	0.94	0.60	1.243
Yes	Yes	Yes	Yes	5	0.98	0.59	2.150

Table 2. We tabulate the results from some of our experiments above. The weighted column indicates that we weighted our jump losses since they are a minority of the actions. Sliced indicates that we removed a 10 frame window from each jump to avoid discrepancies. Dropout indicates whether 50% dropout was used on the dense layers. No Action % indicates the percentage of expert no-actions that were predicted by the agent. Jump % indicates the percentage of expert jumps that were predicted by the agent. Loss is the overall loss.

Table 1 shows the results of these experiments. The experiments without weighting are misleading, since only 2% of the samples are jumps, which is why the model favors doing almost nothing and has such a low loss. Slicing is shown to be helpful in reducing the loss and especially in increasing the correct jump predictions, which supports the concern that slight variations in the expert's jump timings may create discrepancies that the model cannot handle. However, augmenting our data with naive color swapping proved to be somewhat unsuccessful, which may be due to still not having enough colors to work with. We also see that the models begin to overfit by epoch 5. Dropout is shown to be slightly harmful. Although not shown here, the model was still able to perform well on the training set with dropout so the model was still powerful enough even with this regularization.

Below are two links to two of our imitation learning agent playing Geometry Dash. The first is the agent successfully overfitting to level 1 by training and evaluating on the same level and then playing. As expected, our predictions are both strong and correct. The second is the agent trained on levels 2 through 10 and evaluated on level 1 without weighting or slicing. However, there is also an expert issuing jumps whenever the agent does not correctly output a jump, so that we can continue through the level and show its performance. As shown by earlier experiments, the unweighted and unsliced model heavily favors not jumping, but shows promise as there is typically a low signal whenever it needs to jump. Interestingly, it is very confident in predicting jumps when the block is on a pillar, most likely because not only is the pillar very easily identifiable, but there are not many variations of that obstacle. In addition, we see that if sometimes gets mixed signals when different obstacles are combined, such as when it has to jump from a platform to another while also avoiding spikes.

Overfitting: `https://www.youtube.com/watch?v=1mtXeL_hhvw&feature=youtu.be`

Validation without weighting or slicing: `https://www.youtube.com/watch?v=nWl8hmlWeTI&feature=youtu.be`

8. Conclusions and Future Work

Geometry Dash is a deceptively challenging game for artificial intelligence agents but there is still hope in solving it. There are three key issues that make it difficult. First, an agent must discriminate between nearly identical frames while still generalizing to new levels. Second, the agent must solve every obstacle in sequence in order to complete the level. Third, there are not many levels and within each level there is a strong skew towards idling. Reinforcement learning was having trouble solving the first issue due to noisy credit assignment and a large delay between action and consequence. Imitation learning largely addressed the first concern, but since we did not achieve nearly 100% accuracy on both classes without overfitting to the level itself, our agent never completed a level when tested on the emulator; this highlights the second issue. We successfully addressed the third issue by weighting our training samples by the inverse frequency of their associated label. We also attempted to address this issue with priority experience replay, but never had much luck with reinforcement learning in general.

We believe their are still ways to improve the performance of our classifier. First, recall that each obstacle has a window in which you must jump. We attempted to press jump in the middle of this window while collecting data. There are two issues with this approach. First, we were not always perfect in knowing when the middle of the window was. Second, there is value in collecting jumps for multi-

ple valid frames within near the center of the window rather than just one. Since we have only a few levels and not many jumps per level, we could use all the extra data we could get. Second, the background color of the level changes over time. It is likely that this is implemented by cycling the hue in the HSV color-space. If different hues produce different intensities in greyscale, then this is extra information that is useless to the model and may cause overfitting. We could become hue-invariant by instead inputting images in the HSV colorspace and only looking at the SV channels. This could help our model ignore differences which do not matter, and thus pay extra attention to those which due.

References

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

[2] M. Hausknecht and P. Stone. Deep recurrent q-learning for partially observable mdps. *arXiv preprint arXiv:1507.06527*, 2015.

[3] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *arXiv preprint arXiv:1602.01783*, 2016.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[6] S. Ross and J. A. Bagnell. Efficient reductions for imitation learning. *AISTATS*, 2010.

[7] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[8] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. v. d. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

[9] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[10] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *arXiv preprint arXiv:1509.06461*, 2015.

[11] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.